VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

BEZPEČNOST PROTOKOLŮ BEZKONTAKTNÍCH ČIPOVÝCH KARET

SECURITY OF CONTACTLESS SMART CARD PROTOCOLS

DISERTAČNÍ PRÁCE

PHD THESIS

AUTOR PRÁCE — Ing. Mgr. MARTIN HENZL

AUTHOR

ŠKOLITEL — doc. Dr. Ing. PETR HANÁČEK

SUPERVISOR

BRNO 2016

Abstrakt

Tato práce analyzuje hrozby pro protokoly využívající bezkontaktní čipové karty a představuje metodu pro poloautomatické hledání zranitelností v takových protokolech pomocí model checkingu. Návrh a implementace bezpečných aplikací jsou obtížné úkoly, i když je použit bezpečný hardware. Specifikace na vysoké úrovni abstrakce může vést k různým implementacím. Je důležité používat čipovou kartu správně, nevhodná implementace protokolu může přinést zranitelnosti, i když je protokol sám o sobě bezpečný. Cílem této práce je poskytnout metodu, která může být využita vývojáři protokolů k vytvoření modelu libovolné čipové karty, se zaměřením na bezkontaktní čipové karty, k vytvoření modelu protokolu a k použití model checkingu pro nalezení útoků v tomto modelu. Útok může být následně proveden a pokud není úspěšný, model je upraven pro další běh model checkingu. Pro formální verifikaci byla použita platforma AVANTSSAR, modely jsou psány v jazyce ASLan++. Jsou poskytnuty příklady pro demonstraci použitelnosti navrhované metody. Tato metoda byla použita k nalezení slabiny bezkontaktní čipové karty Mifare DESFire. Tato práce se dále zabývá hrozbami, které není možné pokrýt navrhovanou metodou, jako jsou útoky relay.

Abstract

This thesis analyses contactless smart card protocol threats and presents a method of semi-automated vulnerability finding in such protocols using model checking. Designing and implementing secure applications is difficult even when secure hardware is used. High level application specifications may lead to different implementations. It is important to use the smart card correctly, inappropriate protocol implementation may introduce a vulnerability, even if the protocol is secure by itself. The goal of this thesis is to provide a method that can be used by protocol developers to create a model of arbitrary smart card, with focus on contactless smart cards, to create a model of the protocol, and to use model checking to find attacks in this model. The attack can be then executed and if not successful, the model is refined for another model checker run. The AVANTSSAR platform was used for the formal verification, models are written in the ASLan++ language. Examples are provided to demonstrate usability of the proposed method. This method was used to find a weakness of Mifare DESFire contactless smart card. This thesis also deals with threats not possible to cover by the proposed method, such as relay attacks.

Klíčová slova

Bezpečnost, Čipová Karta, Bezkontaktní Komunikace, Model Checking, ASLan++, Formální Verifikace, Protokol

Keywords

Security, Smart Card, Contactless Communication, Model Checking, ASLan++, Formal Verification, Protocol

Citace

Martin Henzl: Security of Contactless Smart Card Protocols, disertační práce, Brno, FIT VUT v Brně, 2016

Security of Contactless Smart Card Protocols

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením pana doc. Dr. Ing. Petra Hanáčka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Martin Henzl

April 14, 2016

Poděkování

Chtěl bych poděkovat svému školiteli doc. Dr. Ing. Petru Hanáčkovi za odborné vedení a spolupráci při výzkumu. Dále děkuji všem, kteří mi poskytli cenné podněty a rady. Děkuji své rodině a přítelkyni za trpělivost a podporu.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Contactless smart cards and devices equipped with near-field communication (NFC) technology are used in many modern applications worldwide. Most of these applications require high level of security. Typical contactless technology applications are in the fields of payment systems [1][2][3][4], electronic tickets and vouchers [5][6][7][8], loyalty programs [9], access control systems [10][11], passports [12][13] and ID cards [14]. There are hundreds of projects worldwide [15], which use NFC devices for small payments, and the number is growing. Contactless cards are more convenient for the user to perform transactions than contact cards; however, they yield new vulnerabilities due to the radio interface. Proper use of these technologies provides high level of security; however, some applications, especially for access control, may be developed by developers that are not security experts, so they can contain vulnerabilities.

Development of secure hardware is very expensive and very slow compared to development of software. Protocols used in security sensitive systems are usually very secure and sometimes even formally verified. The software implementation is usually developed faster than hardware and is usually not formally verified, which makes it the weakest link. The software implementation is as important as other parts of the system. The protocol must be carefully implemented using secure hardware in proper way, which is very difficult, thus there is space for potential mistakes leading to vulnerabilities. People writing such applications have to be perfectly aware of all weaknesses of the particular card type in order to implement the system properly. An automated tool for vulnerability search in contactless communication applications would help them to verify their implementation on particular device.

When designing and verifying security protocols using informal techniques, some

security errors may remain undetected. Formal verification methods provide a systematic way of finding protocol flaws. The protocol is specified in a formal way and the correctness of security properties is proved or disproved using formal methods and mathematics.

The motivation for this work is a massive spreading of new contactless technologies and development of many applications sometimes by developers that are not security experts. Due to the high number of systems using contactless technology worldwide and the possibility of gaining high financial profit from compromising such a system, there are efforts to find vulnerabilities in these systems on both sides, attackers are trying to compromise a system, while developers are trying to fix vulnerabilities and improve security.

1.2 Goals

The high level goal of this thesis is to investigate security of contactless smart card protocols and to find methods of improving security of these protocols.

This thesis is concerned with contactless smart card protocols, which are protocols, such as payment protocols, that use contactless smart cards to store some data, values, cryptographic keys, and to perform cryptographic operations. End users usually use these personalized cards for payments, access control, loyalty programs, etc. The focus here is on contactless smart cards which differ from smart cards with contact interface mainly in two aspects. Firstly, the contactless smart cards are usually simpler due to the power limitations, so they can be modeled more easily. Secondly, the contactless interface introduces threats due to the fact that all communication is wireless. These threats, which are not applicable for smart cards with contact interface, must also be considered when investigating security of contactless smart card protocols.

If we try to understand what security issues can occur in such a protocol, we have to look not only at one level of the communication, such as the RF link, or the high level protocol definition. We have to investigate possible vulnerabilities at all levels.

The focus in this thesis is on the high level attacks on the protocol level. Possibility of these attacks will be analysed and a method of semi-automated vulnerability finding using formal methods will be proposed.

The formal model can be created from the protocol definition or extracted from the eavesdropped communication. Unwanted states that constitute an attacks must also be specified. After analysing the protocol and creating the model including the attack states, the formal analysis methods, such as model checking, can be used.

However, not all kinds of attacks are covered by the proposed method, such as attacks specific to the contactless interface. One of the attack types that are not covered by the method, the relay attack, is investigated separately. A minor part of this thesis is therefore dedicated to relay attack investigation and countermeasure proposal.

Relay attack is one of the most dangerous attacks against contactless devices, because there is no practical countermeasure to it. There are so called distance bounding protocols; however, they are implemented only in some devices, keeping the rest of devices unprotected. In this thesis the relay attacks will be investigated and if possible, a countermeasure to relay attacks performed over network will be proposed.

1.3 Contribution

The contribution of this thesis is twofold. Two areas of contactless smart card security were researched:

- finding high level attacks on the protocol level
- countermeasure to relay attacks

Chapter 2 provides introduction to contactless technologies and smart cards, and analyses threats for smart cards, which are physical attacks, logical attacks, side-channel attacks, and most importantly, threats specific to contactless communication, such as relay attack. This chapter also provides introduction to security API and protocols, and formal methods that can be used for security protocol analysis.

Chapter 3 shows the state of the art in contactless smart card security and overviews the attacks and techniques used to find new types of attacks and the verification processes of protocols. Besides published attacks on Mifare Classic and Mifare DESFire, the focus is on security API attacks and protocol attacks, which provide an insight to methods that can be used to find vulnerabilities, such as formal methods.

Main focus in this thesis is on proposing a method of finding high level attacks on the protocol level. Chapter 4 is dedicated to the description of the proposed semi-automated method of finding vulnerabilities in contactless smart card protocols using formal verification methods. The focus is on contactless smart cards, because they are simpler than smart cards with contact interface, often only memory cards providing encrypted communication.

Chapter 5 provides a formal model and describes a method to create models in ASLan++ language, which is an input format for various model checkers. The ASLan++ model contains smart card, protocol, and attack definitions, which are sufficient for a model checker to be able to find vulnerabilities.

Results of experiments with three sample verifications are shown in chapter 6 to demonstrate the usability of the method. By using this method, the developer can iteratively fix the vulnerabilities found by the model checker and secure the application. The proposed method was also used to reveal a not yet published weakness of the Mifare DESFire smart card. Although the experiments are based on Mifare DESFire types of cards, the method can be used on other smart cards as well, even on more complex cards with operating system.

Although formal verification methods are useful for finding vulnerabilities on the protocol level, the usability of this technique on other attacks on contactless smart cards is limited. Chapter 7 provides investigation of relay attacks and, in particular, relay attacks performed over network. A method is proposed to prevent real-world attacks that induce delays significantly longer than the delay caused by the time travelling longer distance. A possible countermeasure to the overclocking attacks is also proposed, and results from tests on real hardware are provided.

Chapter 2

Background

2.1 Contactless Technologies

There are two main types of contactless devices involved in security sensitive systems, contactless smart cards and near-field communication (NFC) devices. Both use the same communication interface, similar to the interface used in Radio Frequency Identification (RFID) tags. This thesis is focused on contactless smart cards, but many things discussed in this work can also be applied to the NFC technology.

The contactless communication is based on mutual inductance of two coils. The active device (reader) creates the alternating electromagnetic field, which provides the passive device (card) with energy due to the electromagnetic inductance. Data are transferred from the reader to the card by amplitude modulation of the carrier (which also provides the energy to the card). The modified Miller coding with 100 % modulation or Manchester coding with 10 % modulation are used. The data transfer in the direction from reader to card is based on load modulation. The card sends the information by changing its power consumption. These changes can be detected on the reader and the data can be extracted. The Manchester coding with 10 % modulation is used in this direction. The contactless devices work at a frequency of 13.56 MHz and the maximal operational distance is about 10 cm, but may differ for individual card types and NFC devices. The data transfer rate ranges from 106 kbps to 424 kbps. The speed of 106 kbps is the minimum that every contactless card must support and at which every communication begins. All details about the communication can be found in [16].

NFC is the standard designed for handheld devices such as cell phones or PDAs. This technology provides such devices with the capability of contactless smart cards. Such equipped device can be used as a smart card in ticketing, banking, or other

application, or as a contactless smart card reader to communicate with a real card, or it can establish a communication channel with any other NFC enabled device. The working frequency of NFC is also 13.56 MHz. The device may run third party software which uses the NFC interface. The possibility of running arbitrary and potentially malicious code implies the emphasis on security. Security threats are similar to threats of contactless smart cards. Except for the essential communication interface, the NFC device employs the secure element (SE), which is a hardware security chip similar to the smart card. The SE is used for storing sensitive data, such as keys. It can be part of the cell phone, part of the SIM card or it can be in form of a removable secure memory card. There are attacks on NFC devices that exploit vulnerabilities in the NFC architecture and implementation, there is even a possibility to infect the NFC device with the malicious code wirelessly [17].

2.2 Smart Cards

Smart cards are plastic cards equipped with an integrated circuit. Contact cards are compliant with ISO/IEC 7816 [18], which describes the physical parameters of these cards and their contacts and the communication protocols. Contactless smart cards are compliant with ISO/IEC 14443 [19]. The simplest type of card is a memory card which contains just a non-volatile memory. More complex cards are microprocessor cards equipped with the arithmetic logic unit (ALU), which enables them to perform more complex operations. Cryptographic smart cards have in addition a cryptographic coprocessor, which can be used for encryption/decryption, key generation, and pseudo-random number generation. Cryptographic smart cards communicate with the environment via security API, which is designed to not reveal any sensitive data to a potential attacker. Cryptographic smart cards are used for many purposes which require storing of sensitive data, such as cryptographic keys. Asymmetric key pairs can be loaded into the card or can be generated directly on the card and the public key can be consequently extracted. Cards are designed in the way that private keys never leave the card. So if the user generates the key pair on the card, he can be sure that no one had access to the private key in past and will never have in the future. However, there are some attacks that can reveal some information about the secret key. The threats will be briefly outlined in 2.3.

Contactless smart cards have been developed from contact smart cards by adding the contactless interface similar to the interface used in RFID tags. Contactless cards are more convenient for the user to perform transactions than contact cards; however,

they yield new vulnerabilities due to the radio link interface.

There are various RFID standards for communication on various frequencies and designed for communication on various distances. RFID tags are read-only cards with limited memory typically communicating at 125 kHz. Another type is the Gen 2 UHF Card that operates at 860 MHz to 960 MHz.

This thesis is focused on cryptographic contactless smart cards with read and write protected memory. These cards are compliant with the ISO/IEC 14443 standard and use the frequency of 13.56 MHz. Most contactless smart cards provide data transfer encryption, which is essential for many applications. Some contactless smart cards also provide other cryptographic operations such as on-card key generation.

The contactless smart card is often called the Proximity Integrated Circuit Card, abbreviated PICC. Its communication counterpart, the terminal, is usually called the Proximity Coupling Device, abbreviated PCD.

In terms of smart card operating system, there are two categories of cards [20]: smart cards with fixed file structure and smart cards with dynamic application system. The selection of a smart card operating system depends on the application that the card is intended for.

Smart cards with fixed file structure are usually used as a secure computing and storage devices. Files and permissions are defined in advance by the issuer, which is sufficient in applications that do not change frequently. The benefit of this card type is that they are cheap compared to the cards with more sophisticated operating systems. These types of microprocessor cards are the most common smart cards globally. One of the most widespread contactless smart cards are cards from the Mifare family, such as Mifare Classic, which is a memory card with data encryption, or its successor Mifare DESFire. Both Mifare Classic and Mifare DESFire MF3ICD40 has some security vulnerabilities, which will be described later.

Smart cards with dynamic application operating system are more complex and enable developers to build, test, and deploy applications that will run on a card. The two main operating system standards for smart cards are JavaCard and MULTOS. Both support the Java language, in both cases a Java compiler translates the source to Java classes. For JavaCard the classes are converted to JavaCard bytecode, for MULTOS, the classes are converted to MULTOS Executable Language (MEL) bytecode. The bytecode can be executed by the virtual machine on the card. Another type of programmable smart card is BasicCard, which can be programmed using Basic programming language and which is much simpler and cheaper than Java and MULTOS cards. Each type is suitable for different type of application.

2.2.1 Mifare Classic

Mifare Classic cards are contactless smart cards equipped with more powerful chips than classical RFID tags, these cards provide the unique identification number and cryptographically protected memory, which allows such cards to be used in access control, ticketing, and payment systems. The Mifare Classic card is compliant with ISO/IEC 14443A up to part 3. Part 4 of this standard (high-level protocol) differs from the Mifare implementation, the Mifare uses its own proprietary stream cipher CRYPTO1 to secure the communication layer [21]. The CRYPTO1 algorithm is used for data encryption and for the mutual authentication between the card and the reader. The card is designed by the NXP Semiconductors and the implementation details were kept secret.

The security features of Mifare Classic card are UID, pseudorandom number generator (PRNG) and proprietary encryption algorithm CRYPTO1. The UID is set in the factory and cannot be altered. It is used in the anti-collision procedure and for identification purposes. The PRNG is used for authentication and the CRYPTO1 for encrypting the communication after successful authentication.

Before reading any data from the card, the reader has to be authenticated for the particular sector of memory. The Mifare Classic uses a mutual three pass authentication protocol. After this protocol both parties are authenticated and both believe that they share the same secret key. Each sector has two access keys, one of these secret keys has to be used for authentication. Different privileges can be set for these keys.

The set of commands that Mifare Classic supports is small, the commands that are supported are $Read$, $Write$, $Increment$, $Decrement$, $Restore$, $Transfer$.

Mifare Classic was very popular because of its low price until series of attacks were performed [22][23][24][25][26], revealing serious vulnerabilities in proprietary encryption algorithm CRYPTO1 and the PRNG, diminishing security of this card. The card is now obsolete, superseded by the more advanced smart card, the Mifare DESFire.

2.2.2 Mifare DESFire

Mifare DESFire is a successor of the cryptographically weak Mifare Classic. The most significant improvement is that the proprietary encryption algorithm CRYPTO1 was replaced with standard 3DES and AES encryption algorithms. From the mathematical perspective, the 3DES and AES algorithms are sufficiently secure. However, even the strong mathematically secure algorithms can be attacked using side-channel

analysis or there might be vulnerabilities on the protocol level.

Mifare DESFire is used in several large payment and public transport systems around the world, such as the Czech railway in-karta [5], the Australian myki card [6], or the Oyster card used in London [7]. Other applications of this smart card include mobile payment and access control systems.

Although the concept presented in this thesis does not depend on any particular hardware, for demonstration purposes we have chosen Mifare DESFire card.

Mifare DESFire MF3ICD40 [27] is the oldest type of DESFire, using the 3DES algorithm. The next version, Mifare DESFire EV1 [28], supports 128-bit AES and random ID and is Common Criteria certified at level EAL 4+. The newest version, Mifare DESFire EV2 [29] is the most advanced Mifare smart card, employing additional security functions, such as proximity check against relay attacks. Mifare DESFire EV2 chip can hold as many different applications as the memory size supports and new applications can be loaded after the card has been deployed into the market.

The Mifare DESFire MF3ICD40 [27] is equipped with two, four, or eight kilobytes of memory, which is organized using a flexible file system. This file system allows a maximum of 28 different applications on one Mifare DESFire. Each application provides up to 32 files. Every application is represented by its 3 bytes Application IDentifier (AID). The card is protected against cloning with the fixed 7 byte UID, programmed into each device during production. The UID cannot be altered and ensures the uniqueness of each device.

The communication starts with a mutual three pass authentication between Mifare DESFire and PCD depending on the configuration employing either 56-bit DES (single DES, DES), 112-bit 3DES (triple DES, 2K3DES), 168-bit 3DES (3 key triple DES, 3K3DES) or AES. The level of security of all further commands is set during the authentication. The following options are supported for data transfer during the session on Mifare DESFire EV1:

- Plain data transfer
- Plain data transfer with cryptographic checksum (MAC): backwards-compatible mode: 4 byte MAC, DES/3DES/AES based mode: 8 byte CMAC
- Encrypted data transfer (secured by CRC before encryption): authentication with backwards-compatible mode: A 16-bit CRC is calculated over the stream and attached. The resulting stream is encrypted using the chosen cryptographic

method. All other authentications based DES/3DES/AES: A 32-bit CRC is calculated over the stream and attached. The resulting stream is encrypted using the chosen cryptographic method. [28]

2.2.3 Java Card

Java Card [30] is a technology that allows Java Card applets (small Java applications) to be run securely on smart cards. Billions of Java Cards are used as SIM cards, payment cards, ID cards, e-Passports, and more.

Java Card applets are small programs, that can communicate with the terminal and with each other. The applet is a state machine which reads incoming commands, processes them, and responds back by sending data and/or status. Java Card technology enables multiple applications to be deployed securely on a single smart card. Applets can be either pre-loaded on the card, or loaded into a smart card once it has been issued.

Java Card run-time environment performs bytecode verification as part of the applet loading procedure. The applet is first run and verified before it is loaded to the card. The verified applet is signed and installed on the card. Java Card only performs lightweight bytecode verification on the card. Once the applet is verified and signed, it cannot be changed. New verification would have to be performed in case of application change.

Java Card technology was designed to store sensitive information securely on the card. Java cards offer hardware acceleration of symmetric cryptography (DES, Triple DES, AES), asymmetric cryptography (RSA, DSA, ECC), and other functions. Java Card technology relies on the inherent security of the Java programming language to provide a secure execution environment. Data is stored within the application, and Java Card applications are executed in the Java Card virtual machine, which is an isolated environment separate from the underlying operating system and hardware. Java Card virtual machine usually manages several applications, different applications are separated from each other by an applet firewall which restricts and checks access of data elements of one applet to another.

2.2.4 MULTOS

MULTOS [31] is a multi-application smart card operating system, which enables multiple applications on a single smart card. Millions of MULTOS smart cards are being issued by banks and governments, they are used in applications such as contactless

payment, Internet authentication, loyalty programs, national secure ID cards, biometric e-Passports, and access control. Both contact and contactless MULTOS cards exist.

A MULTOS cards support application loading and deleting at any point in the card's life cycle.

Two technologies are used to ensure the secure environment. The first one is the on-card virtual machine that provides secure application run-time environment, memory management, and application loading and deleting. The second one is the Secure Trusted Environment Provisioning (STEP), which is a technology for securing the smart card, application code and application data.

Applications for MULTOS cards can be written in high-level languages (such as Java), which are then compiled to MEL bytecode. MEL bytecode can be executed by the virtual machine.

Each application has own application memory space, which consists of the application code and data. The application has full access rights to its own code and data, but can not directly access to memory of another application. The virtual machine checks the bytecode instructions to ensure they are valid and they do not access memory areas outside the application. If an application attempts to access an area out of its space, the instruction is rejected by the virtual machine and the application execution is terminated.

2.2.5 BasicCard

ZeitControl's BasicCard [32] is the smart card programmable in Basic. Basic was originally developed when computers had very limited resources. The same problem is today with smart cards, which makes Basic a suitable programming language for smart cards. BasicCards can be used in applications like electronic purse, ID card, medical card, Internet security, drivers license network access, software key, access control, gift and loyalty programs. It is focused on shortening the design and implementation time of a custom smart card application.

ZeitControl offers two versions of BasicCard with contactless interface – Contactless BasicCard ZC7.5 RFID and Dual Interface BasicCard ZC7.5 Combi. The latter one is a dual interface card combining both contact and contactless interfaces.

Both smart cards provide symmetric and asymmetric cryptography algorithms, random number generator and cryptographic hash algorithms SHA-1 and SHA-256. Supported symmetric cryptography algorithms are AES with key length 128, 192

or 256 bits, DES and Triple DES with key length 56, 112 or 168 bits. Supported public key cryptography algorithms include RSA encryption and signature algorithms with up to 4096-bit key length, with on-card key generation, and Elliptic Curve Cryptography (ECC) over finite fields of type GF(p), with up to 512-bit key length, also with on-card key generation.

2.3 Threats

There are many ways of attacking a smart card. The attacks on smart cards are typically divided into three main categories – physical attacks, logical attacks, and side-channel attacks. These attacks are summarized for instance in [33]. There are also threats common to all contactless devices, analysed for example in [34]. Both contactless smart cards and NFC devices face the threats specific to contactless communication, such as eavesdropping, interruption of operation from distance, covert communication, and the most dangerous – relay attack.

The smart card must have secure API and the protocols that are used to communicate with the smart card must also be secure. An inappropriate protocol implementation can yield new vulnerabilities even if the protocol itself is secure and the communication with the hardware is considered secure too. There can also occur a situation when an insecure protocol is proven to be secure by the formal reasoning. This can happen when the protocol does not consider all possibilities of the real world, or when the protocol is not implemented properly. There can be situations not defined by the protocol but possible in the real world. Since the focus in this thesis is on security verification on the protocol level, a separate section 2.4 is dedicated to security API and protocols.

2.3.1 Physical Attacks

Physical attacks on smart cards, also known as invasive attacks, or hardware attacks, are pervasive attacks and require advanced equipment. These attacks are performed by highly motivated attackers with high knowledge and assets, who manage to obtain physical access to the smart card. Physical attacks involve physical dismantling of the chip and usually many chips have to be destroyed before an attack is successful.

Physical attacks include scanning the chip structure with microscope, reverse engineering, reading the contents of card's EEPROM memory using powerful electron microscopes, probing the integrated circuit (IC) with a microprobe, re-activating burned fuses, and circuit modification. Although physical attacks on smart cards

typically require sophisticated and expensive equipment, there are also attacks that can be performed with only low cost equipment.

Smart card manufacturers constantly improve their smart card chip designs in order to make these attacks more difficult. Physical attack countermeasures that can be implemented in the IC are: [35]:

- Flexible and user-defined memory encryption of user memory, RAM, and ROM
- Use of a memory management unit to prohibit one application from accessing the code of another application
- Active shielding that renders the IC inactive when triggered
- Small IC geometry (0.22 μm as a maximum feature size) to deter microprobing
- Bus confusion and encryption of data travelling on the bus
- Continuous checking of the random characteristics of the IC
- Proprietary timing and IC layout

This type of attacks is out of scope of this thesis.

2.3.2 Logical Attacks

Logical attacks, also known as fault attacks, are non-pervasive and typically attempt to find and exploit any vulnerabilities or weaknesses in the design of the smart card application or of the whole smart card operating system. These attacks are relatively cheap and simple compared to other smart card attack types. Thanks to these facts much more potential attackers exist. However, these attacks are also easier to prevent, rigorous design and development process of the smart card can help to eliminate logical attack vulnerabilities.

Logical attacks include presenting the card with invalid commands, formats, or field lengths in order to induce an error in the operation of the IC, or buffer overflow. Erroneous operation may reveal sensitive information.

Besides the rigorous design and development process, the logical attack countermeasures are for instance sensors that check the correct IC operation, or redundant logical operations. If the IC is forced to operate outside the established sensor parameters, the IC goes into alarm mode or prevents operation completely. [35]

Examples of logical attacks on security API are provided in chapter 3.2.

2.3.3 Side-channel Attacks

Side-channel attacks are based on information gained from the physical implementation of a cryptosystem on the smart card. Side-channel attacks are non-pervasive, utilizing a side-channel that leak some information, which can be analysed and used to compromise the system and for example extract secret information such as keys stored on the card. Side-channel attacks are based for example on timing information, power consumption, fault analysis, or electromagnetic radiation.

Typical side-channel attack for contactless smart cards is the RF analysis attack.

Although many side-channel attacks require considerable technical knowledge of the internal operation of the smart card, side-channel attacks are probably the most popular attacks on smart cards, because they are relatively powerful and the required equipment is not so expensive as the equipment required for the physical attacks.

Side-channel attack countermeasures that can be implemented in the IC are: [35]:

- Random wait state insertion
- Bus confusion and memory encryption
- Continuous check of random characteristics
- Current scrambling/stabilizing
- Voltage regulation
- Dual bus rails, where the transmission of data is passed from one rail of the bus to the other to confuse the attacker

Example of side-channel attack on Mifare DESFire contactless smart card is provided in chapter 3.1.2.

2.3.4 Threats specific to Contactless Communication

This section provides an analysis of threats specific to contactless technology. These threats are inherent to the wireless communication.

Eavesdropping. The major difference between contact and contactless smart cards is the communication medium. Contactless cards use electromagnetic waves so the attacker can easily intercept and alter data being transmitted over the air, which is a big drawback when compared with contact smart cards. Eavesdropping the wireless communication is possible from a long distance. The eavesdropping and viewing data being exchanged is possible at a low cost. Not only that the eavesdropping is very

easy with appropriate equipment, it can also be executed from distance without being noticed by the user and without trace. There is also a possibility to perform a man-in-the-middle attack. The countermeasure against eavesdropping is the encryption of data being transmitted. In some cases only sensitive data are encrypted, commands and other data are left in the plain text, sometimes weak ciphers or pseudo-random number generators (PRNG) are used, all leading to other possible attacks.

Man-in-the-middle. An effective man-in-the-middle attack is practically impossible because of the short communication distance of contactless smart cards. The attacker would have to establish communication with both card and terminal without being spotted, and without the card and terminal hearing each other. The attacker cannot establish such an attack from distance, the only possibility is to be physically present between the card and the terminal. The user would spot such situation very likely, so the MITM is not considered a big threat. However, there is a possibility to perform a relay attack, described later, to establish a man-in-the-middle attack. One attacker establishes communication with a genuine smart card, the second attacker with a genuine reader, and they relay the communication over their own channel. This yields a possibility to modify the data being exchanged. The man-in-the-middle attack can be prevented by the mutual authentication.

Interruption of Operation. Communication between a card and a reader may be interrupted by transmitting random noise or some other signal at the same frequency. This can interrupt the proceeding transaction at any time and without being noticed. The application needs to know whether the transaction was performed correctly or there was an error. There should be a backup mechanism and backtracking which make sure that the transaction has ended in a regular state. The permanent jamming of the communication can also be classified as denial of service (DoS) attack.

Covert Communication. Contactless smart cards have one big disadvantage from the security point of view. Unlike contact smart cards they can communicate with the reader without user's notice. The contact card must be put into the reader, so the user is always aware of the communication (if the card is in his possession and is not stolen). However, in case of contactless cards, the fraudulent reader can remotely communicate with the card without notice even if the card is in the user's possession. The possible countermeasure is the strong mutual authentication.

Denial of Service (DoS). There can also be a DoS attack performed without the user's notice, for example on the prepaid card. The service the user has prepaid can be denied for example by debiting all monetary units from the card from a distance. The attacker has to understand the communication protocol between the

card and reader in order to send desired commands to the card. Much easier attack is emptying or destroying the smart card by inappropriate electromagnetic waves. The only countermeasure is the Faraday cage.

Communication Link and Dual Modes. This threat refers to smart cards with dual interface, both contact and contactless [36]. These cards usually share an underlying chip. The attacker chooses the less secure interface to attack the chip or he can also switch between these two interfaces during the communication. In order to avoid this attack cards should use only one interface at a time during the whole transaction.

Radio Frequency Analysis. This is a side-channel attack developed from power analysis and electromagnetic analysis [36]. It is base on the fact that the electromagnetic field surrounding the card depends on the actual power consumption of the card, because the card is powered by this field. This attack requires the card to be in possession of the attacker. The attacker can then learn some sensitive data from the card.

Relay Attack. The concept of contactless communication is based on the fact that devices participating in the communication are in the proximity of each other. The security of many applications relies on this fact. An access control system, for instance, grants access to a person who authenticates themselves with the smart card, because it is supposed that this person has the card in their possession at the moment. But what if the genuine smart card is far away rather than in the person's pocket and the communication between the terminal and the card is forwarded? The verbatim communication can be relayed without getting noticed by any of the two genuine parties. The first propose of this kind of attack dates back to 1976, when the chess grandmaster problem was published [37].

The relay attack involves two attackers forwarding verbatim communication between genuine PCD and PICC over their own communication link, using fake PICC and fake PCD. One attacker establishes communication with the the genuine PICC, the second attacker with the genuine PCD, and they transmit the communication over their own channel. The communication parties can be far away from each other and the genuine user has little chance to find out that his card is being attacked. The relay attack is very hard to be prevented, which makes it dangerous.

The attackers do not have to understand the communication to perform successful transactions, because they forward the verbatim communication. If the attackers understand the communication, they can alter data being transmitted and perform a man-in-the-middle attack. An example of device constellation for a relay attack is

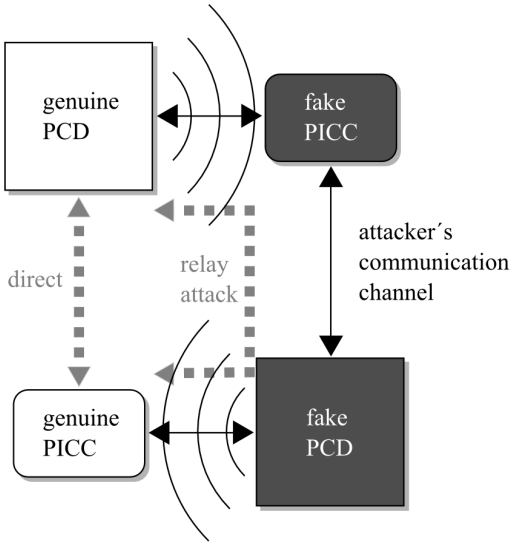

Figure 2.1: Relay attack device constellation

shown in figure 2.1. The maximal response time is limited, so the communication should not be delayed much by the attacker; however, the time limit is usually not restricting enough to prevent this attack. The ISO/IEC 14443 standard restricts the maximal response time, which is negotiated by the card and reader at the beginning of each communication, to 4.949 s.

Relay attacks cannot be prevented on the application level, the only possible defense is provided by the distance bounding protocols, which are based on restricting the round-trip time to some limit. An attacker trying to execute a relay attack causes a delay in the communication, therefore the round-trip time limit will prevent him from succeeding. The maximal time is computed from the speed of light and the maximal allowed distance.

The first distance bounding protocol was introduced in 1993 by Brands and Chaum [38]. It is based on a single bit round-trip delay measurement in a rapid challenge-response exchange. The reader sends out one bit and starts the timer, waiting for the response. The tag responds with one bit, the reader stops the timer after receiving the response. The reader performs n of these rounds and decides, whether the tag is or is not in the limited distance by comparing measured round-trip times with expected times.

The round-trip time is affected not only by the signal travelling from one point to another, but also by the delay induced by the tag processing the command. The

processing time must be very short and invariant in order to get relevant measurements. If the distance bounding protocol is performed at the ordinary communication frequency, which is typically 13.56 MHz, the resolution of one bit corresponds to more than 22 m. In order to increase the precision of distance bounding protocols, different communication methods are used for challenge-response exchanges.

Since Brands and Chaum presented the first distance bounding protocol in 1993, other protocols have been proposed. These protocols were based on technologies such as RF, Received Signal Strength, Ultrasound, Ultra Wide Band and Side-channels.

2.4 Security API and Protocols

Contactless smart cards have limited computational resources, which results in restrictions on cryptographic algorithms and cryptographic primitives that can be implemented on them. Despite the resource limitations, smart card manufacturers try to equip smart cards with the strongest cryptographic algorithms possible, many current cryptographic contactless smart cards employ state-of-the-art encryption and authentication algorithms. However, using a strong cryptographic algorithm does not necessarily mean that the smart card is secure. The smart card must have secure API and the protocols that are used to communicate with the smart card must also be secure. How to achieve these goals will be discussed in the following sections.

2.4.1 Security API

Applications that handle sensitive data, such as payment systems, have to be able to protect these data from a potential attack according to their security policies. The sensitive data often has to be stored in a device which faces a hostile environment. There has been developed a special hardware for storing sensitive data and for performing cryptographic operations, called secure hardware. Examples of such a hardware are smart cards and Hardware Security Modules (HSM). In terms of physical security, this hardware is usually tamper resistant, so the data stored in it cannot be easily extracted even if the attacker is highly motivated and has high assets. This hardware has to communicate, but cannot leak any sensitive information. For this purpose there is a security API, which serves as an interface between the secure hardware and the environment. The security API is designed to be able to communicate with the attacker without any sensitive data leak.

Although method of finding vulnerabilities in protocol implementations presented in this thesis cannot be considered an security API attack, it is closely related to

security API attacks since it finds vulnerabilities in the way the smart card is used by the protocol rather than in the protocol itself.

2.4.2 Security Protocols

Security protocols are communication protocols that aim to reach some goals despite the fact that the communication is transferred over a channel possibly controlled by an attacker, who can interfere with the protocol and perform malicious activity. Typical goals are confidentiality of sensitive data, protecting the integrity of messages, or giving one actor assurance about the identity of another actor with which it is communicating. These goals are typically achieved using cryptography.

Security protocols are usually used to protect something valuable, which means that high assurance about their correctness is required.

Security protocols are quite simple, but it may be a difficult task to make them really secure. The difficulties are not related only to the strength of the cryptographic algorithms used, it is also necessary to take into consideration all possible behaviors of hypothetical attackers, including violations of the protocol rules, and any possible forgery of messages. An attacker can alter any message and can forge and insert any number of new messages at each protocol step, which results in an unbounded number of malicious behaviors. This fact increases the complexity of the already complex communication protocol between concurrently interacting parties. Manual design of new security protocol is a very error-prone task, even with the existence of best practices and recommendations [39].

Nice example illustrating the difficulty of defining security protocols correctly is the Needham-Schroeder public-key protocol [40]. This protocol was believed to be secure for 17 years, until Lowe discovered a flaw in this protocol [41] by applying formal methods [42]. Another example can be the discovery of a logical flaw in the renegotiation feature of the widely used TLS protocol 13 years after the first version of the protocol was published.

Due to the complexity of security protocols, the development requires mathematically based methods for reasoning about their correctness. There are two main lines of research dealing with the rigorous methods for reasoning about security protocols. One is based on quite abstract, symbolic modeling, which was originally published by Dolev and Yao [43], and was developed mainly in the formal methods community. The other one was originally published by Goldwasser and Micali [44] and by Yao [45], and is based on more detailed computational models, involving complexity

and probability theories. Researchers tried to define a relation between these two approaches, either by proving computational soundness for the symbolic model, or by applying reasoning techniques that proved successful in the symbolic model to the computational model. The recent progress in this direction can be found in [46].

The symbolic approach is more abstract, enables better automation in developing proofs, but gives results that are more complex to relate to real world security goals. On the other hand, the computational approach is closer to reality, gives more realistic security assurance at the expense of increased difficulty in proof automation [47].

An example of formal verification of a cryptographic protocol of secure system based on contactless technology is [48].

2.5 Formal Methods

Formal methods are mathematically based languages for specifying a representative model of a system. They are used for checking when a property is satisfied by the model.

Formal methods can be used for proving security properties of protocols such as confidentiality, integrity, authentication, and anonymity. They can be used not only to find out whether the protocol meets these properties or not, but they can also be used to find the counterexample. These counterexamples can be considered possible attacks. Formal methods therefore provide us with the automated way of finding attacks and can also be used for proving that some attacks are not possible. For protocol modelling, the Dolev-Yao [43] model is usually used. Formal methods can be classified into theorem proving, formal logic, and model checking.

- *Theorem proving uses higher-order logic to reason about possible protocol executions by constituting a compelling proof that a particular property always holds. These logics are not subject to finite bounds, and provide mechanized proofs, including automated tools and proof checking, which can assist in parts of proofs and prevent errors in reasoning.* [49] The theorem proving methods are based on various proof search strategies such as the basic resolution strategy [50]. Some inductive theorem provers such as the NRL Protocol Analyzer [51] are very time-consuming because they involve interactive theorem proving by experts. Other security protocol analyzer is Isabelle [52] which also has the specialization for higher-order logic.

- *Formal logic focuses on the study of inference with a set of rules for making*

deductions that are made explicit. It formalizes such deductions with precise rules to decide if an argument is valid. This can be achieved by representing objects and relationships symbolically including the quantifiers and logical connectives. [49] Formal logic includes many logical systems, such as predicate logic or modal logic. The important logic from the security protocol analysis point of view is the logic of Burrows, Abadi and Needham (BAN) [53]. This logic is able to represent belief, freshness and some other properties, that are fundamental for analyzing security protocols. The BAN logic was used to analyze common protocols and successfully detected some minor bugs [54].

- *Model checking is a method to formally verify a finite-state concurrent system. The specification of the system is often written as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system. The verification is achieved by checking if the formal specification can be satisfied by the model.* [49] Model checking uses usually either CTL (Computation Tree Logic) or LTL (Linear Temporal Logic) for temporal logic formulas. Model checking provides an automated way of proving formulas or finding counterexamples, however faces the state explosion problem. This problem can be reduced by symbolic algorithms, partial order reduction, abstraction or by on-the-fly model checking. There are many tools used for security protocol analysis, using a general-purpose or a special-purpose model checker. One of the most protocol analyzers based on model checking is AVANTSSAR [55], which is used in this thesis.

The first two approaches focus on proving the correctness of the protocol according to the properties. Model checking, on the other hand, focuses on searching incorrect traces.

There are many cases where formal methods have detected attacks on protocols that were previously considered secure. Until formal methods were incorporated into the development process, security solutions were verified manually by people. Some traces were often not considered in the analysis. As a consequence, not all errors were detected.

Formal methods can be applied in an early phase of development, before the implementation is available. The designer can verify the protocol from an initial specification and the system requirement. In the case of model checking, if the model does not satisfy the properties, the tool gives a counterexample as output. This counterexample can be utilized in the system design.

A model is only an abstraction of the system, so formal verification cannot reveal implementation errors. Furthermore, if the specification is not correct or the requirements are not properly expressed, the analysis can not guarantee the security of the protocol.

2.5.1 History

Formal methods were initially used to the analysis of hardware systems, software systems and communication protocols in general. Since the 1980s, these methods have been also used for analysis of cryptographic protocols.

The first symbolic approach to cryptographic protocol analysis that was the starting point for model checking tools was that of Dolev and Yao [43]. They introduced the paradigm now known as the Dolev-Yao model.

The first tools that detected attacks on protocols were based on Dolev-Yao approach and model checking techniques. Most of them were specific purposed model checking tools. The first tool which exhaustively searched the state space was Interrogator [56], followed by the Longley-Rigby search tool [57] and the NRL Protocol Analyzer (NPA) [51].

The application of formal methods was considered unreliable for a long time, until the introduction of BAN logic [53]. This logic has a set of simple and intuitive rules and is the most popular example of belief logics. Belief logics have many advantages such as completed automation and faster execution. Their models are more abstract than model checking ones, so the research moved to the model checking techniques.

The fact that Lowe, using the FDR model checker, was able to successfully find a problem in the Needham-Schroeder public key protocol [42], that since then had been unnoticed for seventeen years, alerted people and many researchers started focusing on model checking techniques.

After two years, a new model checking tool concerned with the analysis of security protocols was developed. This tool was named Casper [58]. Other researchers started using their own model checkers for security protocol analysis, such as using of the Murphi model checker to analyze variations on the TLS protocol [59]. Later a special purpose model checkers were developed, such as Brutus [60].

Many tools have been used in the analysis of standards, sometimes detecting problems that would remained unnoticed otherwise. For example, the NPA was used to verify a number of protocols and protocol standards, including the Internet Key Exchange Protocol [61], the Group Domain of Interpretation (GDOI) Protocol [62],

and the Simmons Selective Broadcast Protocol [63]. The AVISPA tool was applied to a suite of protocol standards. ProVerif was used in the production of formally verified implementations of TLS [64] and the smart card protocol InfoCard [65]. Scyther was used for the analysis of the MQV family of protocols [66], for the analysis of the entity authentication protocols in the ISO/IEC 9798 standard [67], and the IKE key exchange protocols in the IPsec standard [68].

It is expected that model checking will ultimately become a standard tool for cryptographic protocol design, as it has become in hardware design. Although model checking has proved useful in the analysis and many protocol designers use it to prove correctness of a new protocol, it has not yet become a standard tool. [69]

2.5.2 Dolev-Yao Model

Dolev and Yao model [43] is based on state machines. A protocol is modeled as a machine consisting of an arbitrary number of honest agents executing the protocol. In this model, there are several concurrent runs of a protocol with the presence of an intruder, who can read, modify or delete the messages transmitted in the network. The intruder can also impersonate any honest agent. The model also formalizes an abstraction of cryptography where messages are represented by terms rather than bit strings, so cryptographic operators do not leak information, e. g., the only way for an adversary to decrypt an encrypted message is to have the corresponding decryption key.

All cryptographic protocol model checking applications are based on the Dolev-Yao model. However, the current approach is different that the original one. Active research is going on to make the Dolev-Yao model more precise and expressive.

An approach similar to the one of Dolev-Yao is based on using algebras for reason about the knowledge of the participants. This approach was first presented by Merrit [70]. The protocol is modelled as an algebraic system. The algebra is used to express the state of the participants, including the intruder, in terms of its knowledge about the protocol.

Chapter 3

Related Work

This chapter presents related work in the field of contactless smart card security and overviews the attacks and the techniques used to find new types of attacks and the verification processes of protocols. This thesis is focused on cryptographic memory cards with fixed file structure, so one section is dedicated to examples of published attacks on this type of contactless smart cards, namely Mifare Classic and Mifare DESFire smart cards. Another section is dedicated to attacks on security API, which in particular shows a method of automatic vulnerability finding in PKCS#11 cryptographic tokens, which was one of the inspirations for the method proposed in this thesis. The last section is dedicated to protocol attacks, which shows attacks on various protocols. The AVANTSSAR tool, which is used in this thesis, was used for finding vulnerabilities in the OAuth protocol.

The section titles contain the word ”attacks”, but the purpose of studying attacks is not to make any harm, but to be able to design better devices and better protocols that are not vulnerable to similar attacks. Especially the methods that can be used to find attacks are worth to be studied because they can be used and adapted to find new vulnerabilities in new devices and new protocols. Such methods can provide designers and developers with valuable information about the security of their products.

3.1 Mifare Classic and DESFire Attacks

3.1.1 Mifare Classic Attacks

The Mifare Classic is a suitable example to show some attacks on contactless smart cards. The security of Mifare Classic was build on the weak proprietary algorithm CRYPTO1. As soon as the algorithm is reverse engineered, the card can be easily

compromised. This corresponds with the well-known principle "Security by obscurity", which should be always avoided [71]. Although some studies [72] advise how to retain some security with the same hardware, it is better to use more secure hardware. The successor of the Mifare Classic card is the Mifare DESFire card, which is much more secure. It utilises strong ciphers 3DES and AES rather than some proprietary algorithm.

Keystream Recovery Attack

Gans, Hoepman and Garcia [26] have shown an attack that exploits the PRNG weakness to recover the keystream used in CRYPTO1 from the eavesdropped communication without knowing the encryption key. This paper also shows how to read all memory blocks from the first sector of the card without knowing the key. Additionally there is always a possibility to read the first 6 bytes of each block from the sector from which the communication has been eavesdropped. The plain text obtained in this attack can also be used in a brute force attack proposed by Nohl and Ploetz.

The weakness of the pseudo-random generator has been independently discovered by Nohl and Ploetz [22] and Gans, Hoepman and Garcia [26]. In the latter paper authors claim that a "random" nonce used in authentication repeats a few times per hour. Nohl and Ploetz discovered that the nonce is generated by an linear feedback shift register (LFSR) which shifts every $9.44\mu s$, which is exactly one bit period in the communication. Hence there is a possibility to get the same nonce after 0.618s if the communication with the card is established at the exact time.

The attacker has full control over the reader and performs communication with the card. This attack utilises the fact that the ciphertext C_1 is obtained by bitwise XOR-ing the plaintext P_1 with the keystream K. When we have two ciphertexts that are encrypted with the same part of keystream and we know one of the plaintexts, we can reveal the second plaintext. We can also reveal the relevant bits of the keystream.

$$\left.\begin{aligned} P_1 \oplus K &= C_1 \\ P_2 \oplus K &= C_2 \end{aligned}\right\} \quad C_1 \oplus C_2 \Rightarrow P_1 \oplus P_2 \oplus K \oplus K \Rightarrow P_1 \oplus P_2$$

There are some parts in the memory, that have the same value in all cards, or that have the value which can be easily discovered. Known data are for example the UID in the first block of the first sector (UID can be obtained in the anti-collision procedure), manufacturer data stored in the same same block or access keys stored in last blocks of all sectors. The attacker changes commands in order to read the known

parts of memory. The attack goes as follows:

1. The attacker eavesdrops and records some real communication between genuine reader and the card.
2. Now he can exploit the weakness of the PRNG. The card repeats the nonce as described. The attacker has full control over the fake reader so he is able to start the communication at the right time. When the same nonce is generated, the same keystream is used.
3. The attacker changes commands under the keystream by modifying the communication in order to receive blocks containing some known data.
4. The keystream can be revealed for all parts where the plaintext is known.
5. By shifting commands he can recover more keystream.

Authentication Replay
To communicate with the card the attacker has to authenticate first. He can replay the previously recorded authentication. For a successful replay of the authentication he must start the communication at the right time so the nonce is the same as was used in the recording. The authentication replay has following steps:

1. The attacker eavesdrops and records some real communication between genuine reader and the card.
2. The authentication requests are repeatedly sent to the card until the nonce generated by the card is equal to the one in the recorded authentication.
3. Now the attacker can send the recorded response to this nonce. This response is correct because the current and recorded nonces are equal. Except for the correct answer to the card's nonce this response contains its own challenge to the card.
4. The attacker gets the card's response to the reader's challenge.
5. Now he can resend same or modified commands to perform the above attack.

Wireless attacks

Garcia, Rossum, Verdult and Schreur [25] have shown that an attacker can retrieve all cryptographic keys of a card by communicating wirelessly with it. The legitimate

reader is not involved which is great advantage compared to the previously mentioned attack. They found and exploit the following vulnerability: After the successfull authentication for one sector the attacker can try to authenticate for another sector without knowing the key. In this case another authentication scheme is used which leaks 32 bits of information about the secret key of that sector.

Mifare Classic sends a parity bit for each byte. The parity bits are not computed from the actual data sent over the air, but they are computed from the plaintext. After the parity bit is computed, whole byte is encrypted. The parity bit is encrypted with the same bit of the keystream that is used for the first bit of the following byte. If the reader sends wrong parity bits, the communication is immediately aborted. When correct parity bits are sent, but wrong authentication data, the card doesn't cancel the communication, but sends the encrypted error code. This vulnerability was used for a brute-force attack [25].

3.1.2 Side-channel Analysis Attacks on Mifare DESFire

This section describes a non invasive side-channel attacks [73] [74] on the Mifare DESFire MF3ICD40, which is the older version using 3DES.

The side-channel analysis brings a new threat to the mathematically secure ciphers. It can help attackers to break modern ciphers, for which no efficient analytical or brute force attacks exist. Side-channel analysis of a white-box implementation of AES on a self-made RFID device with unprotected microcontroller was presented in [75][76], showing the vulnerability of RFID devices to side-channel attacks. The practical attack was shown by breaking the proprietary KeeLoq system [77]. The black-box analysis of a contactless smart card and the results were described in [78], proposing a leakage model for RFIDs that is the basis for analyses in [73] [74], which will be discussed in this section.

In [73], the first full key-recovery attack on the Mifare DESFire MF3ICD40 smart card was presented. This paper points out problems and obstacles that occur when performing side-channel analysis in practice, which are often neglected in academic papers. It shows first application of template attacks to break cryptographic RFIDs, allowing for potentially very fast determination of the secret key. Due to the lack of contacts to measure the power consumption directly, the electro-magnetic emanation of the RFID card was captured with near-field probes and then digitized with a Picoscope 5204 1GHz oscilloscope. The acquired measurements were evaluated on a personal computer. Despite the secure 3DES cipher and RFID obstacles, the authors

were able to extract all 112-bit keys and hence could gain full access to any Mifare DESFire MF3ICD40 card.

The measurement setup used in [73] is an extension of the setup described in [79], where the application of analog demodulation for side-channel analysis of RFIDs was presented.

DES works by dividing the key into subkeys, each of which is used in a different round of the cipher. Parts of these subkeys are combined with bits of input, which can be supplied by the attacker, and sent to a series of S-boxes. Because only a few bits go into each S-box, there is a relatively small set of possible input values, each of which results in a slightly different power consumption profile.

By conducting many experiments and observing the power consumed when calculating these S-boxes on different inputs for known and unknown keys, the attacker can correlate these traces to figure out the bits that come from an unknown key. The full key recovery attack needs approximately 250000 traces, which takes about 7 hours to collect.

The authors had not known the exact implementation of 3DES, they must have reverse-engineered it by examining power traces. During this reverse-engineering process, a countermeasure to side-channel analysis was found in Mifare DESFire MF3ICD40 – adding random dummy DES rounds. However, this protection was not sufficient to prevent the attacks.

Result of these attacks is a full recovery of the 3DES key of the Mifare DESFire MF3ICD40, employing standard equipment that can be built for approximately 3000$. The attacks can be executed within a few hours and hence pose a significant threat to the security of DESFire-based real-world systems. The newer version of DESFire, the Mifare DESFire EV1, uses AES and includes side-channel analysis countermeasures that circumvent the presented attacks.

3.2 Security API Attacks

Although security API is designed to be able to communicate with an attacker without any sensitive data leak, there exist some attacks aimed at security APIs, which can help an attacker to learn some sensitive information from a secure hardware.

One of the first academic papers dealing with API attacks was paper by Longley and Rigby [57] in 1992, presenting analysis of a key management interface of a cryptographic device using the logic programming language Prolog. Another paper dealing with secure hardware failures was the paper [80] by Ross Anderson in 1993.

This paper was focused on the ATMs and procedural and technical failures in the use of hardware security modules (HSM). This paper only showed failures but did not proposed any real security API attack. Later, the same author in [81] showed the first real attack exploiting one dangerous command in the security API. These works inspired others to study security APIs of secure hardware, such as HSMs or cryptographic tokens and smart cards.

The first API attacks were found manually, but it did not take long and automated methods using formal analysis came out. First formal approach to this problem was made with the first order predicate calculus theorem prover Otter [82].

3.2.1 HSM Security API Attacks

In this subsection some attacks on Hardware Security Modules (HSM) are described. HSMs are a kind of secure hardware as well as smart cards. HSMs are used in PCs or other devices for storing sensitive data. All examples presented in this subsection are concerned with ATMs (Automated Teller Machines). There are HSMs in each ATM that serve as a secure key storages and for performing cryptographic operations. Data in the ATM must be encrypted in all cables connecting HSMs in order to avoid eavesdropping or man-in-the-middle attacks.

Anderson published the first real attack [81] exploiting one dangerous command in the security API. At that time many banks computed customer's PIN from his PAN (Primary Account Number) by encrypting this number with a secret key. The result was then converted to the four digit number. When the customer wanted to change his password, then the bank computed the offset between the customer's old and new PIN and saved it into the database. One bank wished to change the PAN structure, but it would change all PINs the customers had. Therefore the bank established the new API transaction for converting the old offset to the new offset. The transaction was following: "given an initial account number of X and offset of Y, calculate an offset which will enable this PIN to be used on account number Z".

Originally the transaction was meant to be just a temporal solution and to be run once in a batch, however it was forgotten and left in the security API. The attack on this single transaction was found about a year later. If the attacker sent his own PAN as Z and his own offset as Y, the command would return the offset between any customer's issued PIN and the attacker's PIN, which is a great security issue. Note that this attack involved just one single dangerous transaction. In [83] Anderson suggested that the possible attack doesn't depend on one transaction, but that there

can be a chain of multiple transactions which could leak a sensitive information such as a key.

Another security API attack is the so called "Decimalisation Table Attack" [84]. This attack exploits the way how HSMs from IBM computed PINs from PANs. The PAN is first encrypted with the PIN derivation key and then shortened to four bytes. The PIN is now represented by the 64-bit binary block that should be converted to four decimal digits. It is done by the lookup table, so called decimalisation table. The decimalisation table was originally fixed, however it later become a parameter of the PIN generation command. The user could therefore specify arbitrary decimalisation table as the input for the command.

Decimalisation table:
`01234567890ABCDEF`
`01234567890123456`
Shortened encrypted account number:
`90AB`
Decimalised PIN:
`9001`

In a normal PIN verification process, if the PIN is incorrect, the HSM answers that it was incorrect and no other information can be learned. When the attacker enters a trial PIN of *0000* with the decimalisation table of all zeroes with just single *1* in let's say seventh position, then if the verification process succeeds, the attacker knows, that the correct PIN doesn't contain any digit *7*.

Only two examples are presented, but there are many other possible attacks, some of them are analyzed in [85] together with formal verification of security APIs.

3.2.2 Attacks on PKCS#11

This subsection is dedicated to automated vulnerability finding in devices compliant with the RSA Public Key Cryptography Standards number eleven (PKCS#11). The Tookan tool presented here was one of the inspirations for the method proposed in this thesis.

PKCS#11 specifies the *Cryptoki* API for performing cryptographic operations such as encryption and signature using cryptographic hardware such as cryptographic token. Sensitive cryptographic keys are stored inside the token and any cryptographic

operation is performed by the token without revealing the key. Compromising a key would allow an attacker to clone the token and to perform the same operations as the legitimate user.

Most current commercial cryptographic tokens and smart cards comply with this standard even if other interfaces are offered in addition. Attacks on this almost ubiquitous standard were found using formal analysis [86]. This is a nice example of finding vulnerabilities in an automated way.

Bortolozzo, Centenaro, Focardi and Steel show in their paper [86] how to extract sensitive cryptographic keys from a variety of commercially available tamper resistant cryptographic security tokens, exploiting vulnerabilities in their RSA PKCS#11 based APIs. They developed an automated tool that reverse engineers the particular functionality offered by a token, constructs a formal model of the API, calls a SATMC model checker to search for possible attacks, and executes any attack trace found directly on the token. This tool is called Tookan (TOOl for cryptoKi ANalysis).

Figure 3.1 shows a high level overview of how the Tookan tool works. The analysis consists basically of four steps:

1. Tookan uses reverse engineering to extract the capabilities of the token, results of this task are written in a meta-language for PKCS#11 models
2. Tookan uses information from the previous step to generate a model, which is given as input to the SATMC model checker.
3. Model checker output is sent to Tookan.
4. Tookan uses results from the model checker for testing on the token.

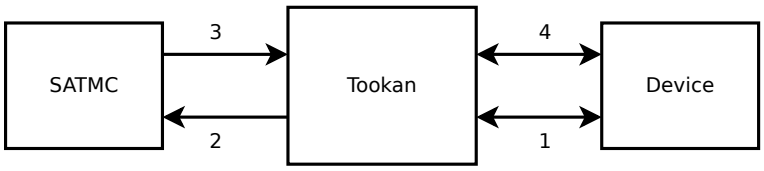

Figure 3.1: Tookan system diagram

Keys and certificates in the PKCS#11 token are objects and access to these objects is controlled. Objects are referenced using handles, which are pointers that does not reveal any information about the value of the object.

Objects have attributes, which may be bitstrings representing the value, or boolean properties of the object, such as whether the key may be used for encryption, or for encrypting other keys. New objects can be created by calling a key generation command, or by "unwrapping" an encrypted key. The token checks that the attributes of the object allow it to be used for every called function.

A session with a PKCS#11 token is initiated by supplying a PIN. The PIN may be intercepted for example by a keylogger, which allows an attacker to establish his own sessions with the device. The PKCS#11 API should protect its sensitive cryptographic keys even when communicating with an attacker.

To protect a key from being revealed, the attribute *sensitive* must be set to true. Once it is set to true, it cannot be reset to false. This should ensure that even if the attacker manages to get the PIN, he will not be able to reveal the keys marked as *sensitive*, because such keys are not readable and will always remain *sensitive*.

A sensitive key can be exported from the device if it is encrypted by another key, but only if its *extractable* attribute is set to true. An object with an *extractable* attribute set to false may not be read by the API, and once set to false, the *extractable* attribute cannot be set to true. Protection of the keys depends on the *sensitive* and *extractable* attributes.

Tookan was used to test 17 commercially available tokens. The results showed that 9 of these tokens that offered the functionality necessary to import and export sensitive keys in an encrypted form were vulnerable to attack, because they allowed the key value to be recovered after a few calls to the API. The other 8 tokens not vulnerable to these attacks had very restricted functionality (e.g. just asymmetric keypair generation and signing). Attacks reported by the Tookan tool are described in the following text.

The notation introduced in [86] for PKCS#11 based APIs, is following: $h(n_1, k_1)$ is a predicate stating that there is a handle n_1 for a key k_1 stored on the device. The symmetric encryption of k_1 under key k_2 is represented by $\{|k_1|\}_{k_2}$.

In the first attack the attacker uses key k_2 with attributes *wrap* and *decrypt* to attack a sensitive key k_1. This attack uses a key with the attributes set for decryption of ciphertexts, and for wrapping, which means encryption of other keys for secure transport. The attacker can simply wrap the sensitive key k_1 using k_2 and then decrypt it using k_2, which reveals the sensitive key k_1. This attack is shown in figure 3.2.

The second attack, shown in figure 3.3, is similar to the previous one, only the wrapping key is a public key $pub(z)$ and the decryption key is the corresponding

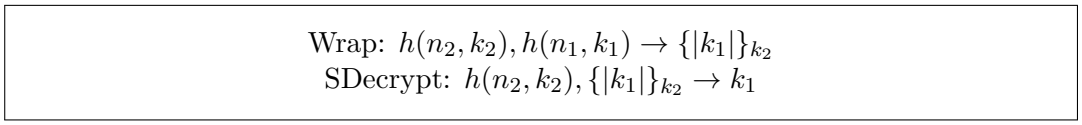

Figure 3.2: PKCS#11 attack 1

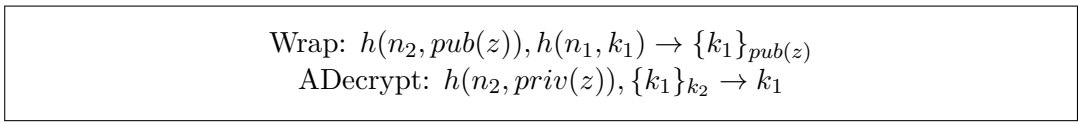

Figure 3.3: PKCS#11 attack 2

private key $priv(z)$. The sensitive key is wrapped and then the attacker can decrypt the obtained ciphertext using the private key.

The third attack is a flaw in the PKCS#11 implementation. Some of the analysed devices, when asked for the value of a sensitive key, returned the plaintext value of the key instead of error code, ignoring basic policy that explicitly requires that the value of sensitive keys should never leave the token.

The fourth attack is a flaw in the PKCS#11 implementation similar to the third attack, unextractable keys were found to be readable. PKCS#11 requires that keys which are declared as unextractable should not be readable, even if they are nonsensitive.

The fifth attack is changing sensitive keys into nonsensitive and unextractable keys into extractable, which was allowed by two types of analyzed tokens. Given that the previously mentioned third and fourth attacks are already possible and sensitive or unextractable keys are already accessible, the fifth attack does not pose an additional flaw of such devices.

3.3 Protocol Attacks

Formal verification seems to be a reliable way of proving security properties of some system. However, there can occur a situation when an insecure protocol is proven to be secure by the formal reasoning. This can happen when the protocol does not consider all possibilities of the real world, or when the protocol is not implemented properly. There can be situations not defined by the protocol but possible in the real world.

Example of such a problem was presented in [87]. There is a very simple application defined, a copy card with just five different messages. Even on such a small system there can be differences in the security claimed by the formal verifier and the reality.

The application copy card is used for paying small amounts of money. There is some prepaid amount of monetary units that can be spent. There is also a possibility to recharge the points on the card any time. Points can be loaded at dedicated vending machines by inserting real money. If the card is stolen, there is no possibility for the user to get the money back, but this is not a serious security problem, because there is just a few points on the card. We will focus on the security of the application provider, who doesn't want to be cheated. Here the cardholder is a possible cheater. He would like to get money on the card without paying the real money for it. Here we can distinguish two security requirements: 1) nobody except the genuine vending machine can store points on the copy card, 2) only points from a genuine card can be accepted by the terminal (merchant).

We will now focus on the protocol for loading points, on which the following attack is performed. The loading protocol uses a challenge-response protocol for authenticating the terminal. The card authentication is not required in the loading protocol, only in the paying protocol. The loading works as follows:

1. *Authenticate()* – at the beginning of the protocol the terminal sends the *Authenticate* command to indicate that it wants to authenticate itself to the card.

2. *ResAuthenticate(challenge)* – card challenge to the terminal.

3. *Load(value, hash(create AuthData(LOAD, passphrase, challenge, value)))* – the terminal issues the *Load* command that allows the card to increment its balance. It has two parts: *value* (number of points to be loaded) and hash value that contains the tag *LOAD*, a passphrase *passphrase*, the challenge from the previous message, and the *value*

The attack on the protocol is following:

1. The attacker glues wires on the smart card's contacts and connects them to his laptop so that he can eavesdrop and modify the communication between the card and the terminal. (this is possible in the Dolev-Yao attacker model)

2. The attacker inserts money to the vending machine and loads points on the card in the usual way.

3. While the card is still present in the terminal, the attacker sends an *Authenticate* message from the laptop to the card. The card answers with a new *ResAuthenticate* message, this message is passed to the terminal.

4. The terminal receives the message and behaves as specified in the protocol – creates a new *Load* message with the new challenge and passes it to the card.

5. The nonce is correct, hence the card accepts the value and increases the balance.

The attacker has paid once but the points on the card were loaded twice. The real implementation of the terminal should never accept a *ResAuthenticate* after sending a *Load* message, however the protocol specified and formally verified in [87] allows this attack.

3.3.1 EMV Attacks

EMV [1] stands for Europay, MasterCard and VISA and it is the most widespread protocol for smart card payments in the world. It is also known as "Chip and PIN". EMV standards support cards with ISO/IEC 7816 [18] contact and ISO/IEC 14443 [19] contactless interface. EMV is something like the protocol framework from which proprietary protocols can be build. An issuer (typically a bank) selects a subset of the EMV protocols to be used (digital signature methods, MAC algorithms) and chooses many customisable options regarding authentication and risk management. Each bank can have its own solution, therefore there can be attacks that work for some issuer's smart cards and doesn't work for other's. Banks can support online or offline PIN verification, authentication by signature or combination of these methods. There are two types of cards considering the authentication. The difference is in using the RSA digital signature. Data stored on the card have to be digitally signed in order to avoid the counterfeit. First type of EMV compliant card is so called SDA (static data authentication), which can not perform the RSA signature itself. The signature made by the issuer is stored in the card and can be exposed to the reader for verification. This type is vulnerable to cloning because the signature can be copied to a counterfeit card. These counterfeit cards usually doesn't need to be provided with the correct PIN, they always answer that the PIN was correct. This type of attack can be avoided by online authentication; however, not all merchants are connected to the network. The other possibility of avoiding this attack, even offline, is to use the second type of EMV card – DDA (dynamic data authentication). This card is capable of performing the RSA, so the challenge-response protocol can be used for

authentication. However, both types of cards can be successfully attacked with the attack presented in [88] which will be described in more detail.

The EMV protocol has three main phases – card authentication, cardholder verification, and transaction authorization.

1. **Card authentication.** In this phase the card is authenticated to the reader. The card provides the information which bank issued the card and proves the integrity of this data (data stored on the card are digitally signed). The terminal first requests a list of supported applications (file 1PAY.SYS.DDF01) and chooses one of them. Then the cardholder information is read and the digital signature is verified.

2. **Cardholder verification.** In this phase the terminal verifies the user's identity. There are three possible cardholder verification methods – PIN verification, signature verification and no verification. Terminal with the card negotiate the method they will use. Some terminals doesn't support some of the verification methods, PIN verification is preferred by merchants because then they have no liability for the fraud. The user has to know the correct PIN for the submitted card to proceed to the next phase. In the offline PIN verification process the PIN entered to the terminal is sent to the card and there verified. If successfully, the card returns OK (0x9000), otherwise an error (0x63Cx, where x is the number of remaining attempts). The card's response was not authenticated at the time of publishing the paper describing this attack. In the online PIN verification process the PIN is encrypted by the ATM and sent to the issuer over network for verification.

3. **Transaction authorization.** In this phase the terminal is ensured that the bank which issued the card authorizes the transaction.

The attack presented in [88] is based on the fact that the card response after the offline PIN verification process is not explicitly authenticated. It is a man-in-the-middle attack. The attacker's device is connected both to the card and the terminal to intercept and modify the communication. In the PIN verification process the device sends 0x9000 as the answer from the card, so the terminal believes that the PIN is correct. The attacker makes the card believe that the terminal doesn't support PIN verification and that the user is verified by the signature. Because no PIN is sent to the card, the attempt counter is not changed. The terminal only recognizes the situation when the PIN verification is performed and fails. In the scenario of this

attack the terminal believes that the PIN is OK and the following communication with the card continues without change.

The attack was successfully executed with a cheap off-the-shelf hardware. The fake card was connected to a general-purpose FPGA board which served as the interface between the card and PC. The PC was connected to the reader that can communicate with the stolen card. The communication was altered in the PC by simple Python program:

```
if VERIFY_PRE and command[0:4] == "0020":
   debug("Spoofing VERIFY response")
   return binascii.a2b_hex("9000")
```

The remaining communication was left unmodified.

3.3.2 OAuth Verification

This subsection shows examples of applying formal verification on a protocol in order to find vulnerabilities. It is focused on OAuth (Open Authorization) protocol, which is a standard being adopted by a growing number of sites such as Facebook, Google, Twitter, and several other social networking sites.

Multiple papers on the formalization and verification of the OAuth protocol have been published. In [89] the authors presented modeling of OAuth protocol using AVANTSSAR platform. This section shows results of this paper. The protocol was formalized with ASLan++ language and several security properties were proposed and specified using extended Linear Temporal Logic (LTL).

OAuth is an open authorization protocol defined by the Internet Engineering Task Force (IETF), that provides a general framework to let a resource owner authorize one third-party application to access the owner's resource on the resource server without revealing the owner's credentials to the third-party application. The resource server is usually same as the authorization server. OAuth 2.0 uses SSL/TLS for communication between the authorization server and the third-party applications. There are four authorization flows defined in the OAuth: Authorization code, Implicit grant, Resource owner password credentials, and Client credentials. Only the Authorization code flow was analyzed in [89].

The authorization code flow is used for server-side web applications. There are four roles in this flow: Resource Owner, User-Agent, Client and Authorization Server. The resource owner is the end user, which uses the user-agent, which is a browser, to connect to the client, which is a third-party web application, and the authentication

server, which is in most cases the server of a social network with large number of users. The resource owner has access to his resources on the authorization server and can grant clients access to these resources. Figure 3.4 shows a sequence diagram of the authorization code flow with some simplifications as modeled in [89]. The resource owner and user-agent are combined into a single role called Browser.

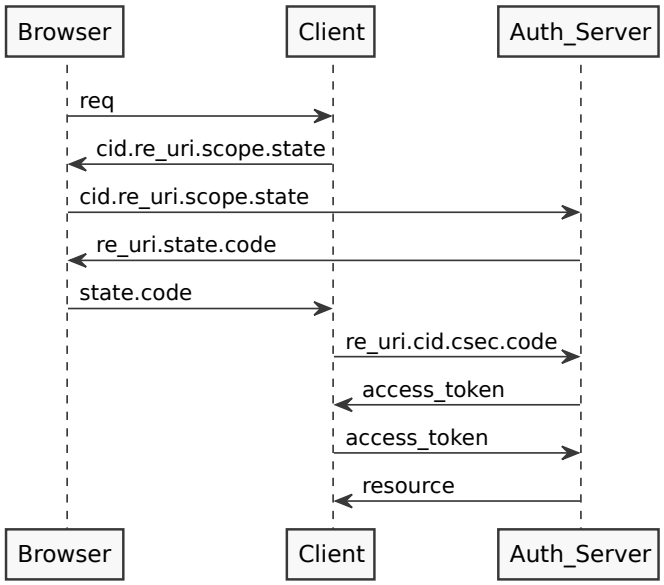

Figure 3.4: OAuth authorization code

1. Browser sends a login request *req* to the client (user clicks on the login button on the client's website).

2. Client initializes the authorization code flow by sending its client ID *cid* (unique identifier issued by authorization server in advance), an optional redirection URI *re_uri* (authorization server can respond to client via browser by the redirection URI), a scope of the access request *scope*, and a recommended secret value for preventing cross-site request forgery *state*.

3. Browser sends these client parameters to the authorization server.

4. Authorization server generates the authorization code *code*, which is bound to the client ID, and sends the authorization code with the redirection URI back to the browser.

5. Browser redirects the authorization code to the original client based on the redirection URI.

```
1 body{
2   Actor -Ch_B2C_1-> C : Req;
3   C -Ch_C2B-> Actor : ?A.?CID.?Re_Uri.?Scope.? State;
4   Actor -Ch_B2A-> A : CID.Re_Uri.Scope.State;
5   A -Ch_A2B-> Actor : ?Re_Uri.?State.?Code;
6   Actor -Ch_B2C_2-> C : State.Code;
7 }
```

Figure 3.5: Statement part of the Browser role definition

6. When the client receives the authorization code successfully, it then sends its credentials client ID *cid* and client secret *csec* to the authorization server directly, not via the browser.

7. Authorization server verifies the client's credentials and the authorization code and if successfully authenticated, the authorization server generates an access token *access_token* and sends it back to the client.

8. When the client receives the *access_token*, he can access user's resources stored in the authorization server with this token.

9. Authorization server authenticates the token *access_token*, if the access token is valid it sends the resource back to the client.

ASLan++ was used to formalize the OAuth model with three agents. The names of agents were abbreviated in the model, so instead of authorization server, browser, and client, the ASLan++ model definition contains letters A, B, and C. Figure 3.5 shows the message flow between browser and other participants as defined in the statement part of the browser role definition of the ASLan++ model.

In ASLan++ role definition, `Actor` is used to refer to current entity, in this case it is referring to the browser entity. There are five channels in the authorization code flow model. The first is a browser to client channel `Ch_B2C_1`, the second is a client to browser channel `Ch_C2B`, the third is a browser to authorization server channel `Ch_B2A`, the fourth is a authorization server to browser channel `Ch_A2B`, and the fifth is another browser to client channel `Ch_B2C_2`.

Line 2 corresponds with step 1 of the authorization code flow described earlier, line 3 with step 2, line 4 with step 3, line 5 with step 4, and line 6 with step 5.

Other agents – client and authorization server – have similarly defined roles describing their behavior in the protocol.

Four security aspects of OAuth were analyzed – confidentiality, authentication, authorization, and consistency. These four security goals were defined in ASLan++ together with the agent definitions. SATMC model checker was used and three attacks were found. These attacks violate the properties of confidentiality of state, confidentiality of authorization code, and consistency. Attacks found by the SATMC are described in the following text. Authors refer to these attacks as Attack of Secret State, Attack of Secret Code, and Attack of Consistency.

In the first attack, shown in figure 3.6 and referred to as Attack of Secret State, an intruder, in the diagram represented by i, intercepts the request from the browser and sends it to the client. Client generates a state and sends its information back to the intruder. The intruder gets the response and decomposes the state parameter.

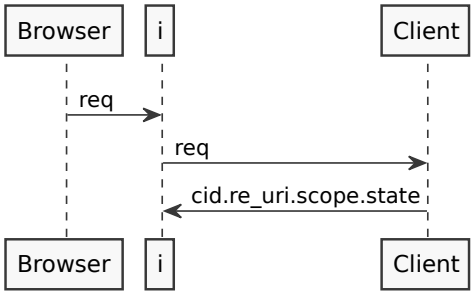

Figure 3.6: Attack of secret state

The second attack, shown in figure 3.7 and referred to as Attack of Secret Code, begins in the exactly same way as the first attack. After receiving the response from client, the intruder modifies the redirection URI and sends the modified response to browser. The browser sends the information about the client to the authorization server and receives the response containing the authorization code. Then the browser redirects the response to the modified URI and the intruder decomposes the response to get the authorization code.

The third attack, shown in figure 3.8 and referred to as Attack of Consistency, is similar to the second attack. The difference is that the intruder modifies the client ID instead of the redirection URI. The result is that users who intend to authorize honest client to access their resources, might authorize other dishonest clients.

We can see that insecure implementation and wrong deployment can cause vulnerabilities in the protocol. Verification and encryption mechanisms can be used to avoid these vulnerabilities.

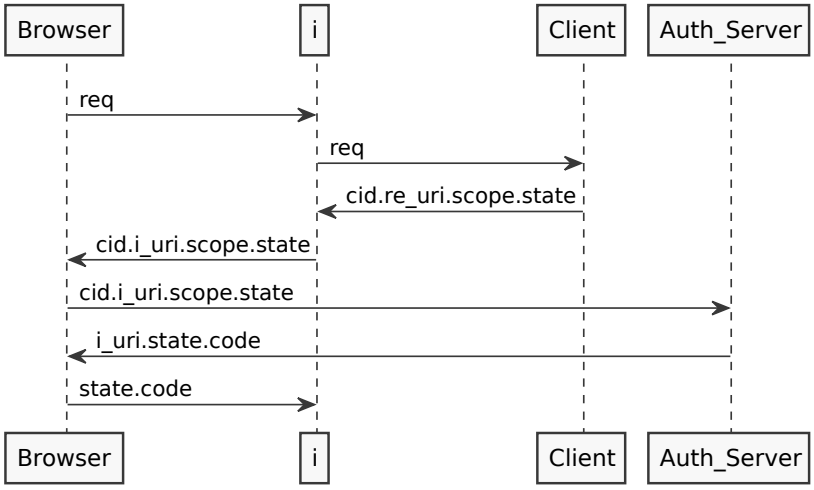

Figure 3.7: Attack of secret code

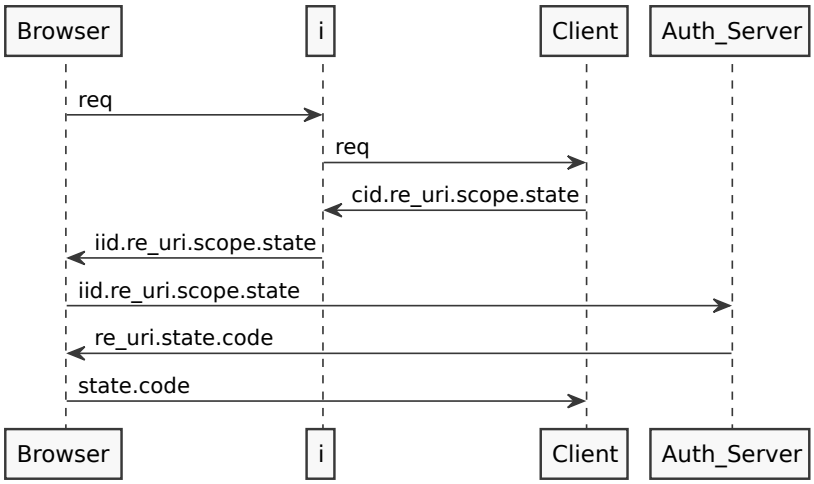

Figure 3.8: Attack of consistency

Chapter 4

Vulnerability Finding Method

This chapter introduces a method of semi-automated vulnerability finding in contactless smart card protocols.

The concept puts together a man-in-the-middle (MITM) attack with verification methods to find vulnerabilities in a semi-automated way. Figure 4.1 shows the scheme of the proposed system. The process consists of a cycle of several steps that can be performed several times to make the protocol secure.

- The first step is a MITM attack that can be used to analyze the protocol. The MITM hardware will communicate with both PCD and PICC, and eavesdrop on the communication to extract the protocol. It can be also used to fuzz test the protocol by altering commands and data in an unanticipated way.
- The next step is the formal model creation. Results from the analysis can be used together with the protocol and smart card specifications to create a formal model. The MITM at the beginning of the process can be theoretically used to create a formal model when analyzing a third party protocol even without the precise protocol specification, the protocol specification can be extracted by eavesdropping on real communication. The developer of a protocol can skip the first step and create the model only from the protocol and smart card specifications.
- The model will be verified by the model checker. In this phase the potential vulnerabilities can be found.
- The attack vectors found by the model checker will be used to execute the attack on the device, using the MITM. If the attack is successful, the vulnerability is reported, otherwise the model is refined.

- The hardware for performing MITM is useful for trying to execute an attack and to figure out how the formal model should be refined after each run of the model checker.

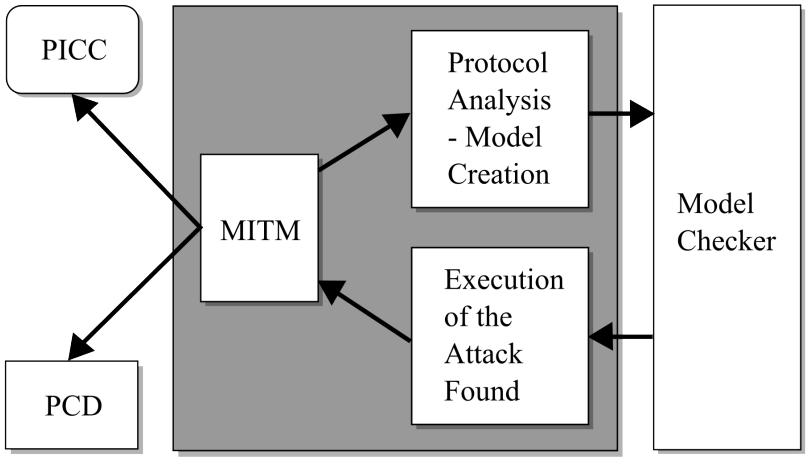

Figure 4.1: Scheme of semi-automated vulnerability search system

This cycle will be repeated multiple times until a vulnerability is found or the model checker concludes that there is no attack on this model. When an attack is found by a model checker and is not confirmed using MITM on the real devices, the model is refined and model checker is executed again. When a vulnerability is found and confirmed by MITM, the protocol should be improved to fix this vulnerability. The model should be updated and model checker should be executed again. Although the process is not yet fully automated, the model checking can find a vulnerability in the model automatically. The following sections discuss hardware for the MITM, protocol analysis, and formal verification.

4.1 Hardware

In this section the hardware used to perform a man-in-the-middle (MITM) attack is discussed. In real environment, MITM can be done using relay attack. Our device will act as the MITM between two legitimate parties of the protocol. As mentioned earlier, there are two contactless devices needed for relay attack – a fake PCD and a fake PICC. We have connected both devices to one PC, which is the main hardware part of the system.

We have established a real relay attack using Proxmark 3, which is an open hardware platform for RFID research purposes. This device was developed by Jonathan Westhues for performing sniffing, reading and cloning of RFID tags. Proxmark 3 incorporates the FPGA unit, used for low level signal processing, and the ARM processor, that implements the transport layer. It can be used as a sniffer, as a reader or as a card, using various protocols. Proxmark 3 supports both low frequency (125 kHz – 134 kHz) and high frequency (13.56 MHz) signal processing.

The Proxmark 3 is connected to the PC via USB; however, the software developed by the community does not support realtime communication over USB, so we had to add it. With the original software the PC sends a command to Proxmark, which returns the result after processing it. We needed a realtime communication with the device, because each data packet received by the device requires its immediate transmission to the computer in order to get the response from the genuine PICC. In order to establish communication with just one party – PCD or PICC – we have implemented the anti-collision procedure.

Figure 4.2 shows the constellation of devices used for the relay attack. Proxmark 3 acts as a fake PICC, communicates with the genuine PCD and forwards data blocks to the PC. It also transmits data blocks in the opposite direction. ACR122 acts as a fake PCD, doing the similar task with the genuine PICC. The PC controls both devices and relays data blocks between them. Data can be saved or altered in the PC.

In our implementation the communication is initialized separately with the genuine PCD and genuine PICC and the anti-collision is not forwarded. The anti-collision procedure is executed separately for both parties. The communication with the PICC must be established first in order to get the information needed to establish communication with the PCD, such as UID, ATQA and ATS [19]. No information is transferred during anti-collision procedure, because the responses must be fast. Then we can forward data blocks from PCD to PICC and vice versa. To keep the communication alive, it is important to maintain correct block numbering and to compute checksums separately for both parties, because every communication error on either of the sides, which would be recovered in normal communication, would break it in forwarded communication during relay attack. When block numbering and checksums are handled separately, the error on one side does not affect the opposite party and after error recovery the communication can continue.

In order to minimize the response time, we have implemented extending of frame waiting time (FWT) and waiting time extension (WTX) (both defined in [19]) directly

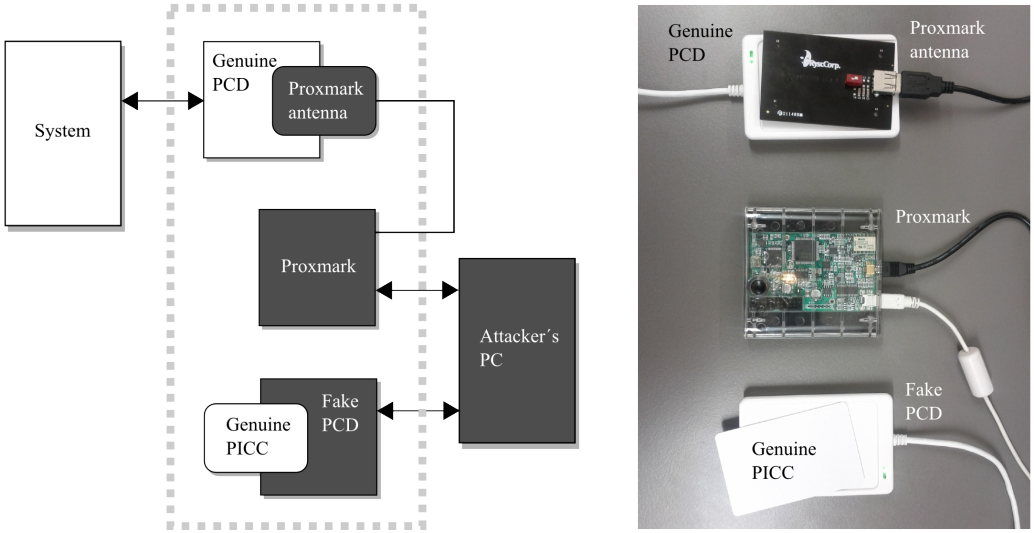

Figure 4.2: Constellation of devices

in Proxmark device, so the response time is minimal. We can increase the FWT up to its maximal possible value (4.949 s) by changing the ATS during the anti-collision. The other option is to send WTX after each command in order to get more time. These features are required for systems where the maximal response time provided by the genuine PCD is very short. We do not have to use these features in systems where the maximal response time is long enough. Providing the ability of extending the response time limit on the lowest level without any delay makes this platform useful even in systems with strict time limits.

4.2 Protocol Analysis

The goal of this part is to create a formal model of the implementation of the protocol. Smart card application issuers mostly don't publish their algorithms for any scientific feedback, hence there could be bugs that might remain hidden for a long time of using such a system. Furthermore, thanks to NFC, there are many more new applications being developed, and there is a great potential for the future. These applications also handle sensitive data and use contactless communication as well as smart cards. In order to be able to find any vulnerabilities in these closed source protocols, the wireless communication can be eavesdropped and the protocol can be extracted by analysing the data being exchanged. With the knowledge of the protocol, the formal model can be created. Limited knowledge of the protocol should therefore not entail

a problem, the data needed for creating the protocol formal model can be extracted from the eavesdropped communication.

The eavesdropping of the protocol can be used to extract the protocol from real communication and the MITM also allows us to alter arbitrary data, change command order, communicate with just one of the legitimate parties and try various commands even with wrong parameters. In theory, the model creation and refinement could be done automatically from data gained by eavesdropping and fuzz testing, which would make the whole process of vulnerability finding fully automatic. Learning techniques allow automatic inference of behaviour of a system as a finite state machine. For example in [90] the authors showed that a Mealy machine representing a model of EMV smart card can be successfully extracted using protocol fuzzing. However, we did not try to make automatic protocol fuzzing, so the process of vulnerability finding is only semi-automatic. Automatic Mealy machine creation using protocol fuzzing was left for future work.

It is very beneficial to have the protocol and smart card description when creating the formal model of the system and not to rely only on data from MITM eavesdropping. The protocol can be described for example as a sequence diagram and the smart card as a Mealy machine. The information gained using MITM together with the protocol and smart card specification gives us an overall image of the system being observed. The creation of formal model from the protocol and smart card description is explained in chapter 5.

4.3 Verification

The formal model of the protocol can be used to automatically find vulnerabilities using formal verification methods. These methods are used for proving security properties of protocols such as authentication, integrity, confidentiality and anonymity. Not only they tell us whether the protocol meets these properties but they can also find the counterexample. These counterexamples can be considered possible attacks. Formal methods therefore provide us with the automated way of finding attacks and can also be used for proving that some attacks are not possible. In this part a model checker will search for possible attacks, which will later be evaluated on the hardware.

4.3.1 Model checking tools

Many tools for verification of protocols are available. Although general purpose formal verification tools can be used to verify security protocols, it is better and more intu-

itive to use one of the tools designed specifically for verification of security protocols.

General purpose model checking tools have been adapted to model security protocols, such as Murphi [91], Spin [92], or UPPAAL [93].

Murphi [91] is a finite-state machine verification tool. It has its own input language, also called Murphi, which is based on a collection of guard → action commands. These commands are repeatedly executed in an infinite loop. In [94] a methodology for the analysis of cryptographic and security-related protocol using Murphi has been proposed.

Spin [92] is a generic verification tool supporting the correctness verification of asynchronous process systems in a rigorous and mostly automated way. Spin models are focused on proving the correctness of process interactions. Systems to be verified are specified in the language Promela (PROcess MEta LAnguage), and properties to be verified are specified using Linear Temporal Logic (LTL) formulas. In [95] Spin/Promela was adapted for verification of security protocol.

Uppaal [93] is a tool for modeling, validation and verification of real-time systems. System model in Uppaal is a parallel composition of timed automata extended with data types.

Tools dedicated to the verification of security protocols are for instance Casper/FDR3 toolbox [58], NRL protocol analyzer [51], Cryptyc [96], Scyther [97], LySA [98], and Choreographer [99]. There are also tools targeting on wide use and applicability to practical problems, such as ProVerif tool [100], AVISPA tool suite [101][102], and AVANTSSAR [55].

The Cryptyc [96] is a cryptographic protocol type checker that allows to check for violations of security policies. LySa [98] is a process calculus for security protocols. It applies static analysis technology to develop an automatic validation procedure for protocols. LySatool is an implementation of the analysis. Choreographer [99] is an integrated tool for security and performance analysis of UML models, which uses the LySatool as an analysis back-end.

Casper

Casper [58] is a program that will take a description of a security protocol in a simple, abstract language, and produce a CSP (Communication Sequential Process) description of the same protocol, suitable for checking using FDR3 (Failure Divergence Refinement).

NRL Protocol Analyzer

NRL protocol analyzer (NPA) [51] is a special-purpose verification tool for analysing security protocols, written in Prolog. NPA was one of the earliest tools for verifying the security of cryptographic protocols. It was not originally designed as a model checker, but later many of the model checker features were added, including the ability to check properties expressed in a temporal logic language, NPATRL [62].

Scyther

Scyther [97] verifies bounded and unbounded number of runs, using a symbolic analysis with a backward search based on partially ordered patterns. Scyther does not require the input of scenarios. Scyther implements model checking with respect to the unbounded model by performing a backward-style search. For this method, the model is extended with adversary events for encrypting, decrypting, hashing, and knowing messages. Infinite sets of states are represented by trace patterns, which are partially ordered sets of events that must occur in the traces, and whose messages may contain variables. The events in patterns must satisfy a number of criteria based on the semantics.

ProVerif

ProVerif tool [103] is an automatic cryptographic protocol verifier based on a representation of the protocol by Horn clauses. It can deal with an unbounded number of sessions of a protocol and an unbounded message space. It uses abstractions to obtain an efficient analysis method, such as: individual fresh values are abstracted into sets of fresh values, and each action of a thread can be executed multiple times.

AVISPA

AVISPA (Automated Validation of Internet Security Protocols and Applications) [101] is a tool funded by the European Union, which provides a push-button, industrial-strength technology for the analysis of large-scale Internet security-sensitive protocols and applications. AVISPA uses several different model-checking approaches. Protocol models are written in the High Level Protocol Specification Language (HLPSL) [104]. Protocols are specified in HLPSL in terms of their roles, using control flow patterns, data structures, alternative adversary models, as well as different cryptographic primitives and their algebraic properties. HLPSL specifications have a declarative semantics based on Lamport's Temporal Logic of Actions [105] and an operational

semantics defined in terms of a rewrite-based formalism called the intermediate format (IF). Once the model of the system is specified in HLPSL, AVISPA translates it into the IF, which is an input format for AVISPA back-end model checkers. AVISPA utilizes four back-end tools for validation of security protocols: On-the-fly Model-Checker (OFMC), Constraint-Logic-based Attack Searcher (CL-AtSe), SAT-based Model-Checker (SATMC), and Tree Automata based on Automatic Approximations for the Analysis of Security Protocols (TA4SP). The advantage of having multiple back-ends is that only one model can be specified and it can be analysed with four different tools. AVISPA is a popular tool and it was used for verification of security protocols that use smart cards multiple times.

AVANTSSAR

AVANTSSAR (Automated VAlidatioN of Trust and Security of Service-oriented ARchitectures) is a follow-up project of AVISPA, introducing new languages for describing models, the AVANTSSAR Specification Languages ASLan++ and ASLan. ASLan++ [106] is a high level formal language similar to the HLPSL, used for specifying security-sensitive service-oriented architectures, their associated security policies, and their trust and security properties. The semantics of ASLan++ is formally defined by translation to ASLan, the low-level specification language that is the input language for the back-ends of the AVANTSSAR Platform – OFMC, CL-AtSe, and SATMC.

OFMC [107] combines a number of techniques to enable the efficient analysis of security protocols. First, OFMC uses lazy data types as a simple way of building efficient on-the-fly model checkers for protocols with very large, or even infinite, state spaces. A lazy data type is one where data constructors build data without evaluating their arguments. Second, OFMC models the adversary in a lazy fashion, where adversary communication is represented symbolically and solved during search. Third, while OFMC performs verification for a bounded number of sessions, it works with symbolic session generation, which avoids enumerating all possible ways of instantiating possible sessions. Fourth, OFMC exploits a state-space reduction technique, inspired by partial-order reduction, called constraint differentiation [108]. Constraint differentiation works by eliminating certain kinds of redundancies that arise in the search space when using constraints to represent and manipulate the messages that may be sent by the adversary. Finally, OFMC also provides some limited support for handling different equationally specified operators on messages [109]. [69]

Cl-Atse [110] represents protocol states symbolically as a collections of non-ground facts, which record the states of different threads, the messages sent to the network, and the adversary knowledge. In particular, constraints are used to describe what the different agents know and a constraint calculus is used to solve for what they can know, from messages previously exchanged, i. e., the calculus is used to solve a variant of the non-ground intruder deduction problem. CL-Atse was designed to allow the easy integration of new deduction rules and operator properties. [69]

SATMC [111] is an open platform for model checking of security services. SATMC reduces the problem of checking whether a protocol is vulnerable to attacks of bounded length to the satisfiability of a propositional formula which is then solved by a state-of-the-art SAT solver. This is done by combining a reduction technique of protocol insecurity problems to planning problems and SAT-reduction techniques developed for planning and LTL that allows for leveraging state-of-the-art SAT solvers. SATMC provides a number of distinguishing features, including the ability to check the protocol against complex temporal properties (e.g. fair exchange); analyze protocols (e.g. browser-based protocols) that assume messages are carried over secure channels (e.g. SSL/TLS channels). [112]

4.3.2 Tool Selection

There are many papers describing and comparing various formal verification tools, such as NRL and FDR comparison [113], Casper/FDR, ProVerif, Scyther and Avispa comparison [114], or OFMC, Cl-Atse and ProVerif comparison [115]. Various tools have been studied and tested for the purpose of this thesis to find out which one is the best for security verification of protocols using contactless smart cards. During the process of selecting the right tool, various aspects of the tool had to be considered, such as performance, how difficult it would be to model desired features in the particular modeling language, and published results with the tools.

The AVANTSSAR tool was chosen for the security verification, mainly because the fact that the high level ASLan++ language can be easily used to model the desired features and because three different back-end model checkers can be used to verify the model. Also there are published papers suggesting that this tool and its back-end model checkers have good results in the field of security protocol verification. The performance seems to be good and for example performance comparison [115] of ProVerif with AVANTSSAR back-end model checkers Cl-Atse and OFMC shows better results of AVANTSSAR back-end model checkers; however, the difference is

not significant. AVANTSSAR developed from AVISPA and both tools seem to be proved and used by the community.

Chapter 5

Formal Model

This chapter provides a description of the proposed method that can be used to create a model of a contactless smart card and a terminal and to define states representing attacks. This model can be then used in model checking to find attack traces in the protocol. The model takes into account the implementation details of a particular smart card which could be possibly avoided in a high level protocol verification. These details are important because wrong use of smart card commands may introduce a vulnerability even if the high level definition of the protocol is secure. The ASLan++ language was chosen for protocol modeling, it can be used as an input for multiple back-end model checkers of the AVANTSSAR Platform.

A model of protocol in ASLan++ is defined by roles that can be played either by a legitimate party or by an adversary called intruder. We establish two main roles in the model description to represent the implementation – the first role represents the smart card with its functionality and settings, the second role represents the protocol. The protocol is executed by the terminal, the smart card only responds to commands from the terminal. The protocol can be therefore identified with the terminal in our model. The intruder model that is used is the well-known Dolev-Yao intruder model [43]. All communication is synchronous with the intruder, the intruder intercepts the messages from the legitimate user and each legitimate user receives messages only from the intruder. The intruder can be therefore identified with the network. Figure 5.1 shows the configuration of subjects in the model. The PCD executes the protocol and communicates with the PICC via the intruder, who is a man-in-the-middle. The goal of our vulnerability finding method is to find out if the intruder would be able to perform some attack in this configuration and find an attack trace.

The state explosion problem has to be addressed. If we create precise model of the smart card and the terminal functionality, the model will be too complex for the

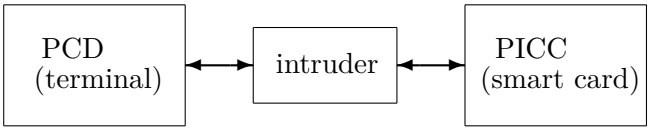

Figure 5.1: Intruder model

model checker, the number of states will be so high that the model checking execution time will be unacceptable. The goal of this thesis is to create modeling method that will create models which can be computed using model checking in acceptable time and which describe the functionality sufficiently. We create simplified models that are weaker than the precise model would be, so more attacks can be found. Attacks that are found by the model checker can be tested and in case of false positive the model can be adjusted to be more precise and not contain the particular false vulnerability. The resulting model will be a trade-off between precision and model checker execution time.

Since smart cards are usually used in applications where high level of security is required, confidentiality, integrity and authentication should be provided by the protocol to protect data that are being transferred between the smart card and the terminal. Confidentiality of data is achieved by encrypting the data using any of the state-of-the-art ciphers, which are strong enough to be relied on. In this thesis the strength of the cipher is supposed to be sufficient to resist attacks focused merely on breaking the cipher rather than finding vulnerabilities in the protocol. We therefore consider all ciphers unbreakable for purposes of this thesis so that we can focus on vulnerabilities in the protocol.

Integrity of messages exchanged between smart card and terminal can be ensured in multiple ways, such as computing the cyclic redundancy check (CRC) of plaintext and encrypting it together with data, or by using message authentication code (MAC), which is a cryptographic hash. MAC can be used to cryptographically secure the integrity of data even if these data are not encrypted.

Contactless smart cards usually require terminal authentication which ensures that the data will not be revealed to unauthorized entities. Each file in the smart card has usually access permissions that are used to authorize operations on these files. The access rights are determined according to the symmetric key that was used for authentication.

5.1 Modeling Tool

ASLan++ is the specification language used in AVANTSSAR. It is a high-level formal language for specifying security-sensitive service-oriented architectures. It is easy for system designers to use, because it is close to the way in which they think about systems. It can be used also by users who are not experts on formal specification language. The AVANTSSAR platform provides conversion from high-level ASLan++ to ASLan, which is a low-level specification language used by back-end model checkers to perform verification of security properties.

ASLan++ document consists of four parts: Entities, Declarations, Statements, and Goals. General schematic architecture of ASLan++ is shown in figure 5.2. An ASLan++ model is a hierarchical structure of entities. The top-level entity is called Environment, its sub-entity is called Session. Sub-entities of Session are used to describe characteristics of different agents or roles. The entity contains a collection of declarations, starting with keyword *symbols*, and a series of statements, starting with keyword *body*. Declarations are used to define types, variables, constants, and functions. They are the static part of the entity, while statements describe the dynamic part of the entity. Goals are used to formalize the desired security properties. The most general way to formalize security properties is to use extended Linear Temporal Logic (LTL) [116] formulas. Validation goals have a name and a LTL formula that is checked by the validation back-ends. Another way to define a goal is to define an assertion. Assertions are given as a statement in the body of an entity. They are expected to hold only at the given point of execution of the current entity instance.

The ASLan++ model can be checked by any of the AVANTSSAR back-ends. The back-end model checker will then give a counterexamples when an attack is found, which can be used to deduce the security flaw of the system. When no attack is found, it doesn't necessarily mean there is no vulnerability in the protocol. The reason may be that the model checker explores the search space to the maximal depth which was previously set in the back-end without finding any attack.

5.2 Smart Card Model

The PICC can be seen as a state machine. The PICC reads commands from PCD, changes its internal state according to these commands, and responds back. States of the machine are determined by the internal state of the PICC logic and by the value of internal variables of the PICC, such as content of files and used cryptographic

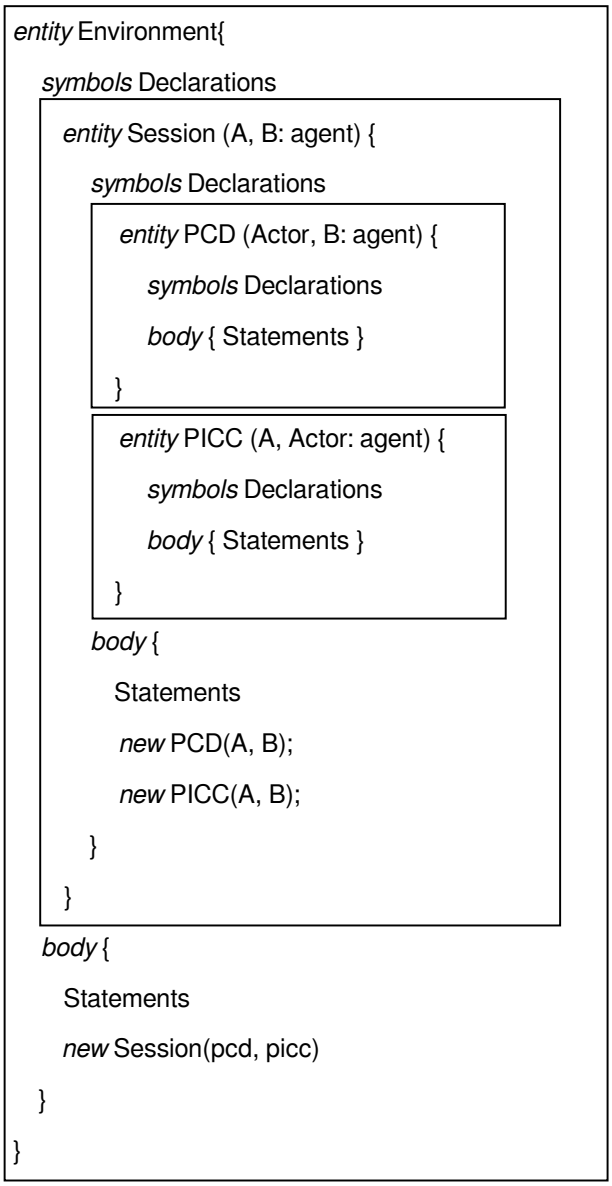

Figure 5.2: Schematic architecture of ASLan++

keys. Since the logic must have finite number of states and the files and keys can only have finite number of values, the number of states of the machine will be finite. The transition rules of the automaton are defined by the set of commands and parameters of these commands. Although the set of parameters will be high, it will be finite, so the number of transition rules will be finite as well. We can therefore model the PICC behavior using a finite-state automaton or, more specifically, a Mealy machine, whose output is determined by the current state and the current input. Another state machine concepts can be used instead, such as UML state machine, which is an enhanced realization of the finite-state automaton mathematical concept with characteristics of Mealy machine. UML state machine diagrams are convenient for describing contactless smart card behavior, because they support enhanced methods for simple picturing of complex behavior, such as extended states, hierarchically nested states, and orthogonal regions.

The automaton should describe behavior of a PICC in the level of detail suitable for model checking, which means the simpler the better. It should be designed to be simpler than the real implementation and allow false positives rather than false negatives. It should be as simple as possible, because the model checking could take unbearable amount of time due to the state explosion problem, if the number of states was not kept low. The model can allow false positives because it can be iteratively refined, but it should not allow false negatives, which would result in false belief that the system is secure. The automaton can be refined if false positives are found by the model checker.

Figure 5.3 shows a sample UML state machine diagram describing logic of the Mifare DESFire MF3ICD40, which is one of the cards later used to demonstrate the verification method. Mifare DESFire is a memory card, so the logic is quite simple. The card shown in the figure has three applications, the default application number 0 and two standard applications with numbers 1 and 2, and uses two keys for authentication, so the user can be authenticated using $key1$, $key2$, or not authenticated ($noKey$). Only basic commands needed for a payment protocol are modeled, the authentication command ($auth$), select application command ($select$), read file ($read$), and write file ($write$). Two actions of 1) putting the card to the proximity of the reader which starts the communication and 2) taking the card away from the reader to end the communication are represented by *activate* and *deactivate* transitions respectively.

The model should represent the behavior of a personalized issued card that is ready to be used in the protocol, which means that it does not have to support

all commands which are used for the smart card personalization or commands that are not enabled after the smart card is issued. This approach results in simpler models and shorter model checking computation times. The Mifare DESFire smart card supports more commands than the commands shown in figure 5.3, but these commands will not be used after the card is personalized in secure environment, so they are useless in the model. Also the application 0 in the model does not allow authentication, because it is used only during personalization for operations related to creating and setting up other applications.

When the contactless smart card is put to the proximity of the reader, it is activated and an anti-collision procedure is performed. The anti-collision procedure is used to allow multiple cards to communicate with the terminal without interference. After the anti-collision procedure, the terminal communicates only with one smart card at a time, the order of smart cards is negotiated during the anti-collision procedure. There is no reason for modelling the anti-collision procedure, so in the model the card gets immediately into the *active* state. When the card is taken away from the reader, the communication is terminated and the card is deactivated.

The diagram in figure 5.3 uses features of UML state machine diagrams to simply picture complex behavior. The diagram uses nested states. If a system is in the nested state (called substate), it is implicitly also in the surrounding state (called superstate). The state machine will attempt to handle any event in the context of the substate, but if the substate does not prescribe how to handle the event, the event is automatically handled at the higher level context of the superstate.

The figure describes an extended state machine which uses extended states to describe memory of the card. The extended state is a combination of the state and the extended state variables. This feature is very useful, because state machines without extended states need large number of states to implement variables. The machine from figure 5.3 can be pictured without extended states using orthogonal region implementing memory, as shown in figure 5.4. Each state can contain two or more orthogonal regions and being in such a state means being in all its orthogonal regions simultaneously. The number of states in the memory region is very large, so only a couple of states are depicted to show the notion. We could define the state machine without orthogonal regions, such machine would have states from the cartesian product of states in the current orthogonal regions.

The memory cards will result in very simple diagrams, while smart cards with more complex logic like Java Cards or BasicCards, which allow execution of arbitrary code, will result in more complex diagrams. Examples in this thesis are based on

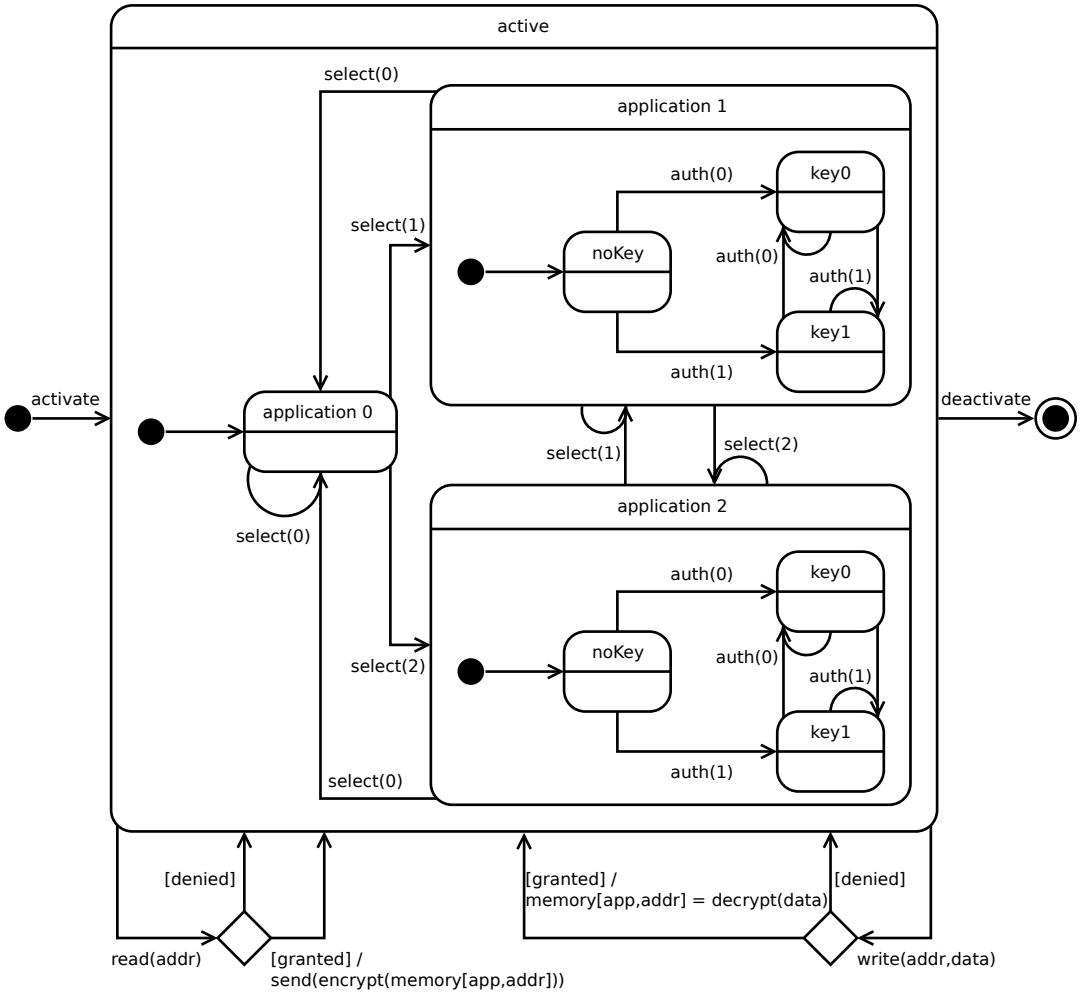

Figure 5.3: UML state machine describing basic Mifare DESFire behavior

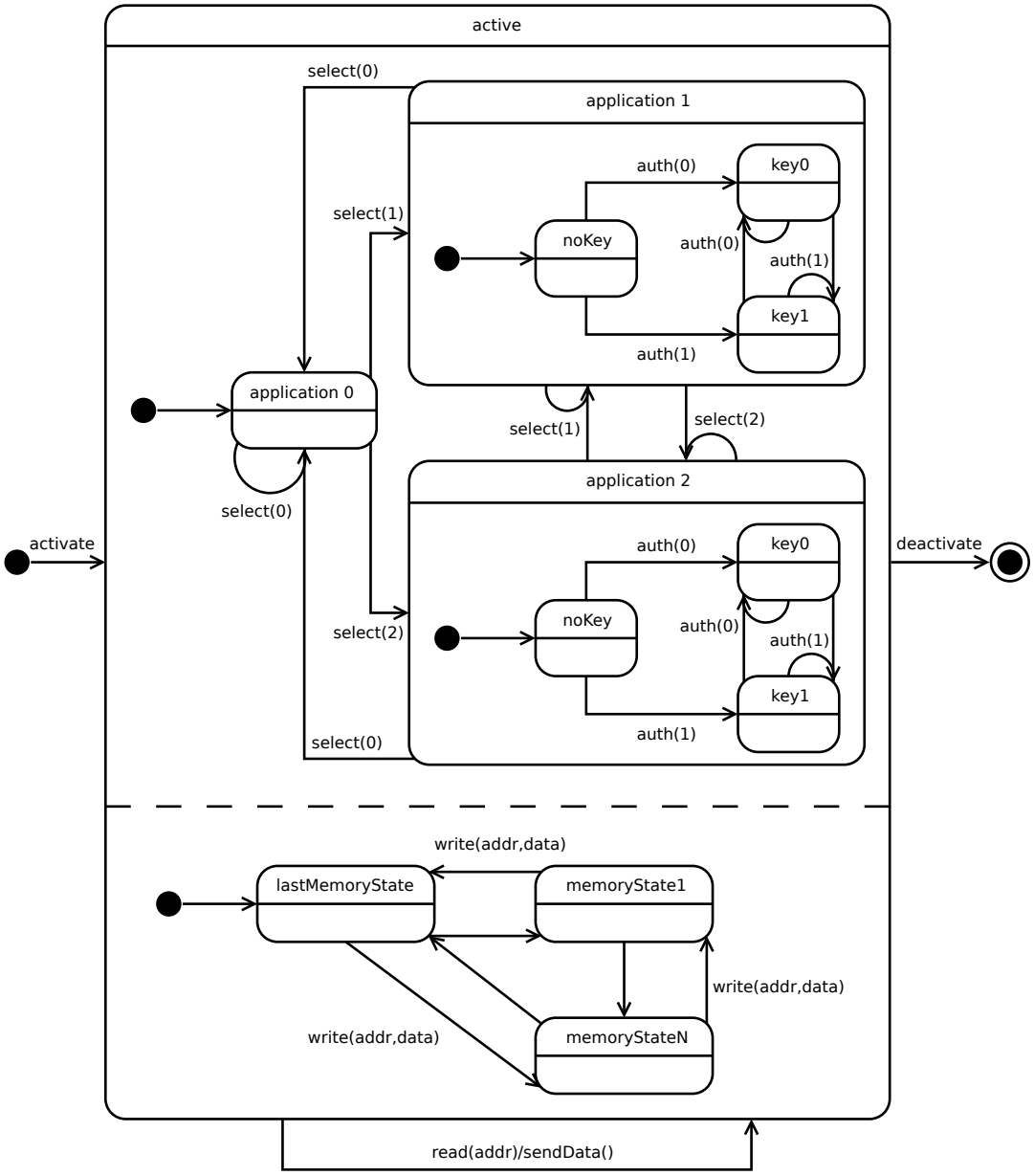

Figure 5.4: Mifare DESFire UML state machine with memory states

Mifare DESFire, but models of other card types can be also created.

Although UML state machines are very useful for depicting behavior of contactless smart cards, the behavior can also be described using simple finite-state automatons and Mealy machines. Such description is more formal and can provide more detailed insight.

We can create the Mealy machine representing the PICC by combining an automaton describing the PICC logic and an automaton representing the state of memory (the two machines that were combined using orthogonal regions in figure 5.4). The formal definition of the PICC Mealy machine will be provided later. We can analyse the logic and memory automatons separately.

The PICC logic automaton should describe behavior of PICC as a response to the commands sent by PCD. Let M_{logic} be a deterministic finite automaton defined as a quintuple $(Q_{logic}, \Sigma_{logic}, \sigma_{logic}, q_{logic0}, F_{logic})$, consisting of:

- a finite set of states Q_{logic}
- a finite set of input symbols Σ_{logic}
- a transition function $\sigma_{logic} : Q_{logic} \times \Sigma_{logic} \rightarrow Q_{logic}$
- a start state $q_{logic0} \in Q$
- a set of accept states $F_{logic} = Q_{logic}$ (PICC may end in all states)

Figure 5.5 shows an example of M_{logic} automaton describing logic of the Mifare DESFire based on 5.3.

In this example the card has three applications and uses two keys for authentication. The states are denoted by a pair of application number and authenticated key respectively. The initial state is the state where default application number 0 is selected and no authentication was performed – authenticated key 0. Only basic commands needed for a payment protocol are modeled, the select application command (*select*), the authentication command (*auth*), the read file command (*read*), and the write file command(*write*). Read and write commands do not change state of the automaton, for these operation the memory automaton will be needed. The diagram does not contain description of all transitions, which are same as in 5.3, and does not show final states. All states are potentially final, since the communication with the card can be ended or interrupted in arbitrary state.

The automaton describing the state of the PICC memory has states determined by the content of files, values of cryptographic keys, and values of all other variables

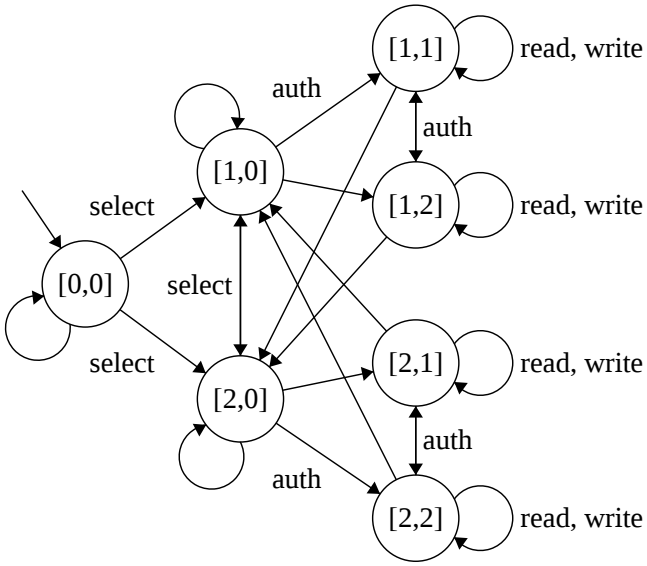

Figure 5.5: FSM describing smart card behavior for some basic commands

that are persistent in the PICC memory and that can be changed during the life of the card. It can be defined similarly as the M_{logic}. Let $A = a_1, a_2, ...a_n$ denote all memory blocks (files, keys, etc.), n is the number of memory blocks. Let D be a set of all possible data that can be stored in a block. Let $C_{write} = A \times D$ be a set of all write command parameters, which consist of memory address and data to be written and let $C_{read} = A$ be a set of read command parameters consisting of memory address and let c_{noop} be a command for no operation. Let M_{memory} be a deterministic finite automaton defined as a quintuple, $(Q_{memory}, \Sigma_{memory}, \sigma_{memory}, q_{memory0}, F_{memory})$, consisting of:

- a finite set of states $Q_{memory} = D_1 \times D_2 \times ... \times D_n$, where n is the number of memory blocks
- a finite set of input symbols $\Sigma_{memory} = C_{write} \bigcup C_{read} \bigcup \{ c_{noop} \}$
- a transition function $\sigma_{memory} : Q_{memory} \times \Sigma_{memory} \rightarrow Q_{memory}$ (commands for writing data C_{write} change state appropriately, C_{read} and c_{noop} do not change state)
- a start state $q_{memory0} \in Q$ (initial content of memory)
- a set of accept states $F_{memory} = Q_{memory}$ (PICC may end in all states)

The automaton describing the PICC is the combination of the automaton describing the PICC logic and the automaton representing the state of memory.

Let M be a Mealy machine defined by a 6-tuple $(Q, Q_0, \Sigma, \Lambda, T, G)$ consisting of the following:

- a finite set of states $Q = Q_{logic} \times Q_{memory}$
- a start state $Q_0 = (q_{logic0}, q_{memory0})$, which is an element of Q
- a finite set of input symbols $\Sigma \subseteq \Sigma_{logic} \times \Sigma_{memory}$; input alphabet will contain only meaningful commands:
  $(write, c_i)$, where $write \in \Sigma_{logic}, c_i \in C_{write}$
  $(read, c_i)$, where $read \in \Sigma_{logic}, c_i \in C_{read}$
  (c_i, c_{noop}), where $c_i \in \Sigma_{logic} \setminus \{write, read\}$, $c_{noop} \in \Sigma_{memory}$
- a finite set called the output alphabet $\Lambda = D \bigcup R$, where R is a set of PICC status responses and D will be used for read command responses
- a transition function $T : Q \times \Sigma \to Q$ mapping pairs of a state and an input symbol to the corresponding next state
- an output function $G : Q \times \Sigma \to \Lambda$ mapping pairs of a state and an input symbol to the corresponding output symbol

An intuitive interpretation of a Mealy machine is following. At any point in time, the machine is in some state $q \in Q$. It is possible to give inputs to the machine by supplying an input symbol $i \in \Sigma$. The machine then responds by producing an output symbol $G(q, i)$ and transforming itself to a new state $T(q, i)$.

The read and write commands will be processed only after correct authentication, which is determined by the state of the logic automaton. The read command will return the file content based on the state of the memory automaton, and write command will change the state of the memory automaton. All other transitions will return only status of the command execution.

5.2.1 States reduction

The model checking execution time strongly depends on the total number of states. In order to keep the model checking time short, the number of states of the state machine that simulates the smart card should be as low as possible, so some optimization should be performed. To reduce the number of states in the state machine we can

reduce the number of states used for logic (M_{logic}), or for memory (M_{memory}), or both.

To reduce the number of states that describe logic of the smart card, we can keep only states that has any side effect, for instance send data to the reader (read command) or make persistent changes in the memory (write command), and join them with the supporting states that represent the chain of commands. We can create optimized commands that are combination of multiple real commands. Each combined command has a side effect. We simulate commands for data transfer – *read* and *write*. This approach reduces execution time of the model checker. Figure 5.6 shows the state machine from figure 5.3 with reduced number of states. There are only two commands:

- read: this command is a combination of select application, authenticate, and read command
- write: this command is a combination of select application, authenticate, and write command

This reduction is possible and has no impact on attack finding results, because the supporting commands for selecting application and authentication can be performed multiple times and only the last performed command has impact on the following read or write command. The internal state is determined by the last select and authenticate commands, the previous commands are forgotten. The read and write commands will contain parameters for selected application number, which will be consequently part of the memory address, and other parameter for authentication and determining the authentication key. The figure contains the authentication token *auth*, which will be described later together with the authentication mechanism.

To reduce the number of states in the M_{memory}, we have to reduce the number of memory blocks that can be written to, and/or reduce the number of possible data that can be stored. If the card supports addressing of data blocks by application, file ID, offset and length, the number of possible write locations can be tremendous. Better approach is to have only memory locations that the application is supposed to write to or read from and one undesired location for each file that will be used to simulate writing or reading to bad location that will corrupt the result. Using this approach the total number of states will be reduced dramatically, which will also reduce the model checker execution time.

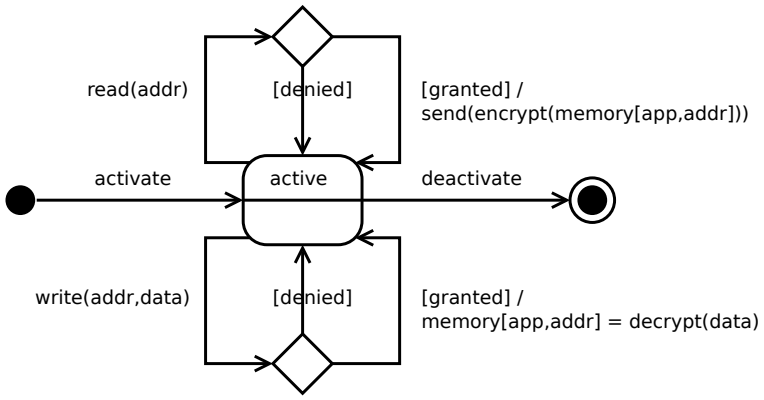

Figure 5.6: Reduced number of states

5.2.2 PICC Entity

When the PICC behavior is known and modeled for example using UML state diagram, the PICC role in ASLan++ can be created. The ASLan++ general schematic was shown in figure 5.2, the PICC behavior is defined in the *entityPICC*, which contains *symbols* declarations and *body*. The *body* of the PICC role can be created based on the UML state diagram. The basic PICC functionality that is created in the *body* is an infinite loop that reads commands from PCD, processes them, and sends responses back to the PCD, as shown in figure 5.7. The states of the PICC (as defined in the UML state diagram) are determined by values of state variables, that are defined in the *symbols* declarations part.

States can be defined in several ways. There can be one PICC state variable or there can be multiple state variables. In the latter case the PICC state is determined by values of all state variables together. The state variables may represent for instance the selected application and authenticated key.

PICC response is based on the current state and the received command. Both state and command variables are declared in the *symbols* part of the PICC entity. ASLan++ allows new type definition, so the state variable may be of type *state* and the command variable may be of type *command*. These types can be declared in the *symbols* part of the *Environment*. These types should be declared as subtypes of the basic type *text*. Variables could also be declared as *text* without creating new types.

For creating a model of Mifare DESFire with reduced set of commands as shown in 5.6 no states are necessary, because the model has only one state. The PICC responses are then based only on the received commands.

```
entity PICC (A, Actor: agent) {

  symbols
  State: state;
  Command: command;
  ...

  body {
  ...
    while(true) {

      % read command
      A -> Actor: ?Command;

      select {
        on(State = state1): {
          select {
            on(Command = command1): {
              ...
            }
            on(Command = command2): {
              ...
            }
            ...
          }
        }
        on(State = state2): {
          select {
            on(Command = command1): {
              ...
            }
            on(Command = command2): {
              ...
            }
            ...
          }
        }
        ...
      }
      % send response
      Actor -> A: ok;
    }
  }
}
```

Figure 5.7: PICC role in ASLan++

This section is dealing only with the logic automaton and shows only the basic structure of the PICC role. The PICC behavior is more complex when the memory automaton is taken into account. The memory automaton is not created in the same way by modeling its states, it is created in a more natural and straightforward way by introducing variables that represent the memory of the PICC and the state of the memory automaton is determined by the values of these variables. In other words, the state of the memory automaton is determined by the content of the PICC memory. The PICC will also have other variables for example for authentication purposes as described later, and we will consider it as part of the memory automaton.

The *body* part of the PICC entity can access the memory for read and write, so the resulting model will be the combination of the logic and memory automatons.

5.2.3 Basic Concepts

There are some basic concepts that can be put together to form a smart card model. These concepts are general and can be used to create a model of arbitrary smart card with pre-defined set of commands. We describe modeling of the following concepts:

- Applications
- Authentication
- Encryption
- Files and Permissions
- Personalization
- Integrity

The following sections describe the method of creating a PICC role in the ASLan++ for these concepts, how to implement basic commands (commands of the PICC automaton) and also how to implement the simplified commands (commands of the PICC automaton with reduced number of states).

Applications

Multi-application contactless smart cards support multiple applications even from different vendors on a single card. The application on cryptographic memory card is not an executable program, it is rather a set of resources dedicated to application outside the card. The application on the card can consist of files used to store data

and symmetric keys used for authentication and data encryption. The application outside the card can securely store data in the card and read them back later. This can be used for instance for payment applications or loyalty program applications, where some credit is stored on the card.

To simulate the application selection in the PICC role, we can use a state variable which is set by the PCD using a *select* command. The value of selected application is then used for file access. If we use the automaton with reduced number of states, the application selection is part of another command, such as the *read* or *write* command.

Authentication

The authentication process between smart card and terminal is usually mutual, both parties must prove possession of a common secret. In case of Mifare DESFire contactless smart card, the three-pass authentication is executed and the common secret is the DES/3DES key. When creating a model of a smart card, the authentication does not have to have precisely three message exchanges, it can be simplified in order to keep the number of states low. The simple way of simulating the mutual authentication process and modeling in ASLan++ is a fresh session key generation performed by one of the parties and sending it encrypted using the authentication key to the other party. The other party must check that the session key is fresh and was never used before during the protocol run. This approach uses a trick based on the fact that we can be certain of things that we cannot in the real environment. We can have a secret key shared only by legitimate entities and we can be sure that the intruder does not know the key. So if something is encrypted using this secret key, such as the fresh random session key, the receiving party can be sure that the message was encrypted by the legitimate counterpart, and also the sending party can be sure that only the legitimate counterpart can decrypt the message. The sending party generates the fresh session key to simulate new session key generation performed during the three-pass authentication, the receiving party must check that the key is really fresh and was never used before during the protocol run. The fresh session key generation and checking by the other party will prevent replay attack on the authentication. Figure 5.8 shows example of a three-pass authentication. $\{A.B\}_K$ means concatenation of A and B encrypted using encryption key K.

Thanks to the fact that in the model we can be certain of things that we cannot in the real environment and that the PICC can remember all previously used session keys and check that the new session key is really fresh, we can simulate the authentication

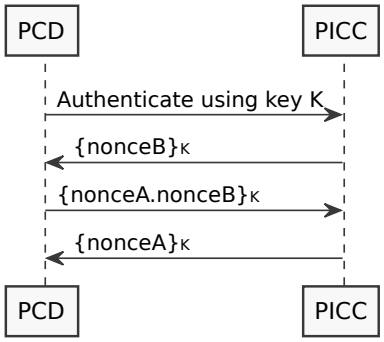

Figure 5.8: Three-pass authentication example

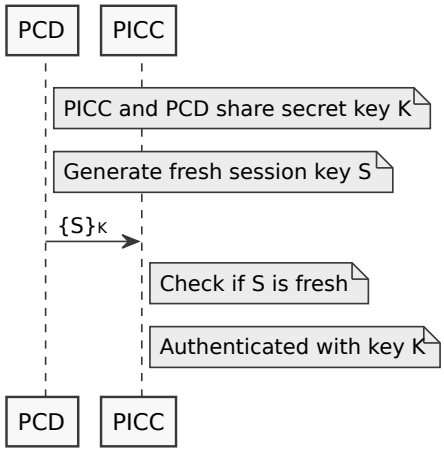

Figure 5.9: Simplified authentication used in model

using only one message exchange, as shown in figure 5.9.

After the one-pass authentication, PCD and PICC share the common session key, which could not be eavesdropped by the attacker, because it was encrypted with key not known by the attacker. The PICC knows which authentication key was used and can grant access to files accordingly. The authentication needs to be implemented in both PCD and PICC roles. The PCD always starts the communication and sends commands, so it will also generate a random session key.

In case of the automaton with reduced number of commands, the authentication can be part of another message. Figure 5.10 shows possible ASLan++ source of one-pass authentication, where the authentication token is part of the $readFile$ command. The PCD generates fresh $SessionKey$ and sends it in the $auth$ token with authentication key $key1$, which is not known by the intruder. PICC checks that the session key was never used before (authentication resulting in fresh session key) or

```
entity PCD (Actor, B: agent) {
  ...
  body {
    % fresh session key generation
    SessionKey := fresh();
    % read name
    Actor -> B: readFile(addressName, auth(key1, SessionKey));
    B -> Actor: enc(SessionKey, ?Data);}
  }
}
entity PICC (A, Actor: agent) {
 ...
  body {
    while(true) {

      % read command
      A -> Actor: ?Command;

      select {
        on(Command = readFile(?DataAddress, auth(?AuthenticatedKey,
              ?SessionKeyTemp))): {
          % authentication
          select {
            on(!UsedSessionKeys->contains(SessionKeyTemp) |
                    SessionKey = SessionKeyTemp): {
              % store current session key
              UsedSessionKeys->add(SessionKeyTemp);
              SessionKey := SessionKeyTemp;

              % authenticated
              ...
            }
          }
        }
      }
    }
  }
}
```

Figure 5.10: One-pass authentication in ASLan++

that it is the current session key, in which case the protocol continues with the old session key (no new authentication). The current session key is stored in variable $SessionKey$ and the set of all used session keys is $UsedSessionKeys$. In case of successful authentication, the current session key is stored in $UsedSessionKeys$ set for later use.

Encryption

The high level language ASLan++ already supports modeling of communication encryption, but it does not consider various modes of encryption algorithms. In ASLan++ any data can be encrypted using symmetric or asymmetric cipher. These ciphers are considered unbreakable for purposes of protocol modeling, therefore the intruder cannot learn the plaintext of the encrypted data unless he knows the corresponding key. The complexity of breaking the encryption algorithm is out of scope of this thesis. But there are different modes of encryption that must be taken into account when creating a model even if the cipher algorithm itself is considered unbreakable. Symmetric ciphers are used in the following modes:

- ECB – Electronic Codebook
- CBC – Cipher Block Chaining
- CFB – Cipher Feedback
- OFB – Output Feedback
- CTR – Counter

The ECB mode encrypts each block of data in the same way independently on the other blocks. The initialization vector is same for each block. The other modes are more secure, because each block encryption depends on the previous blocks, which makes the cryptanalysis more difficult. The initialization vector of the cipher is changed after each block encryption, so each block is encrypted using different initialization vector. Mifare DESFire MF3ICD40 specification states that DESFire uses CBC mode. Although each block of data is encrypted in CBC mode, same initialization vector is used for each block, which means that for short data blocks the data is encrypted using ECB and we will consider it as ECB mode for purposes of this thesis. This mode is prone to replay attacks, because each data block is encrypted using the same initialization vector and the same key. In ASLan++ each block is encrypted

```
% ECB mode
encryptedECB := enc(SessionKey, Data);

% CBC mode
encryptedCBC := enc(SessionKey, nextIV(lastIV), Data);
```

Figure 5.11: ECB and CBC encryption modes in ASLan++

using same key and there are no initialization vectors, so we can consider it the ECB mode.

From the protocol modeling perspective, the CBC, CFB, OFB, and CTR modes do not differ. They use the initialization vector which is different for each block. The strength of these modes is out of scope of this thesis. We can model these modes by adding fresh number (not used before and not known by the intruder) to the data being encrypted, simulating the changing initialization vector. This approach will provide resistance to replay attacks.

Encryption in ECB mode can be written in ASLan++ as $enc(SessionKey, Data)$, a non-invertible function representing $Data$ encrypted using key $SessionKey$. Non-invertible means that although it may be overheard by the intruder, the intruder is not able to invert the function to get the $SessionKey$ or $Data$.

In case of CBC, we can use initialization vectors that are chained using custom function $nextIV()$ so that fresh initialization vector is used each time. The first initialization vector is custom vector $zeroIV$, the next one is $nextIV(zeroIV)$, the next one is $nextIV(nextIV(zeroIV))$, etc. The encryption in the CBC mode can then look like this: $enc(SessionKey, nextIV(lastIV), Data)$, where $lastIV$ is the last initialization vector. Other encryption modes can be modeled along the same lines.

Figure 5.11 shows encryption in ECB and CBC modes.

Files and Permissions

Smart cards provide file system with permissions that can control access to each file based on the key that was used for authentication. We can model files and permissions in ASLan++ either as variables or as facts. If the structure of files is static and will not change during the life of the smart card, it is possible to model files using variables in PICC role. Each file would be a variable and file permissions would be variables as well. Better approach is to use ASLan++ facts. Facts are global and more flexible,

so when using facts it is possible to check content of PICC files even from the PCD role, and it is possible to add new facts and retract existing facts, which can be used to simulate flexible file system where files can be created and deleted. Figure 5.12 shows how the file system can be declared in ASLan++ as fact *fileSystem* with four parameters for data address, authentication keys to get read and write permission, and data itself.

```
fileSystem(text, symmetric_key, symmetric_key, message): fact;
```

Figure 5.12: PICC file system in ASLan++

The first parameter of the fact represents the address of the file and is of type *text*, which is the most simple type in ASLan++. The second parameter represents authentication key that must be used to obtain read permission to this file and is of type *symmetric_key*, which is an ASLan++ type for symmetric keys. Analogously, the third parameter is the authentication key for write permission. The fourth parameter represents data stored in the file and is of type *message*, which is a compound type that can store any combination of data of any other type.

Although address has a simple type, it represents a number of values that constitute the address on a real card, such as selected application number, file ID, offset, and length of data. We decided to have a separate fact for each data block that can be addressed instead of one fact per file, which results in more than one fact per file. Blocks of different lengths and offsets may overlap, so not all blocks will contain meaningful data. Such blocks will contain the message *corrupted* to easily recognize unwanted data.

Long files will contain many fact definitions, but for modeling purposes we can reduce the number of possible file addresses by defining only the desired addresses and one invalid address instead of all possible invalid addresses. Reading from this invalid address will return *corrupted* and writing to this location will save *corrupted*.

Personalization

Behavior of each smart card type can be modeled using basic principles of applications, authentication, encryption, files, and permissions. All cards of one type has the same behavior. For using in a protocol, such as payment protocol or loyalty program, the smart card must be personalized. Personalization is a process when the smart

card is initially populated with data of an intended smart card user, such as the name or the account number. Consequently, each smart card will contain different data in files. This process should be taken into consideration when modeling the smart card protocol. The personalization process does not have to be modeled, since it usually takes place in a trusted environment. The smart card can be used in the modeled protocol only after the personalization, so we can create the model of a card which is already personalized. To create the model of a personalized smart card, all files must be created and populated as they would be during the personalization process.

Integrity

Integrity of data exchanged between the PCD and the PICC is important, but it is not always possible for the PCD or the PICC to check the integrity. The attack definitions described later will cover these attacks so that any attack on integrity will be reported by the model checker.

There are situations in which the PCD or the PICC can check the integrity of data to avoid an attack, such as if some mechanism providing integrity assurance is used or if the integrity of data is protected by itself. The integrity protecting mechanisms can be for example message authentication code (MAC) or encryption in CBC mode. The data with its own protection mechanism are for example certificates, which are digitally signed. For data with this property we can implement integrity check in the ASLan++ source so that the PCD or PICC can find out that the data has been altered and perform a response to such attack. Otherwise the PCD or the PICC cannot distinguish between genuine data and forged data, so the integrity assurance depends on the inability of attacker to send forged data. The model checker may find an attack on integrity, in such case some integrity mechanism should be implemented.

5.3 Application Logic Model

There are two interacting roles in the ASLan++ model, the PICC, representing the card, and the PCD, representing the terminal. The PICC is only executing commands sent to it from the PCD, so we model the application logic of the protocol in the PCD role. The PCD role contains the application logic of the terminal and of the back-end systems. It issues commands to the PICC and decides what to do next when the response from PICC is received. The PCD represents the protocol run.

During the development, the developer can use the sequence diagram of the protocol or the flow diagram of the application as the basis for the PCD model. The PCD

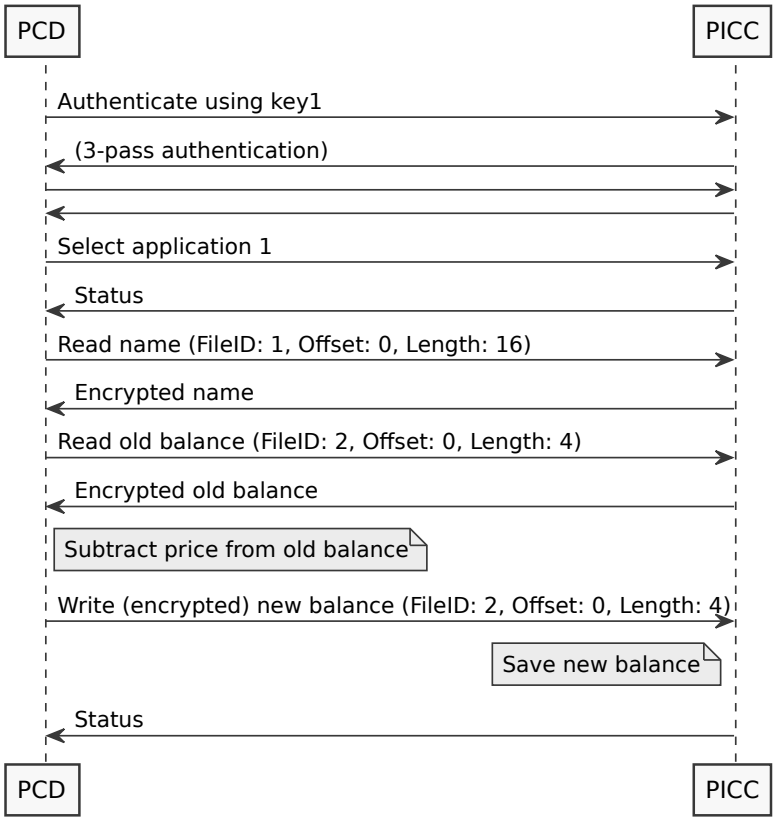

Figure 5.13: Sample payment protocol

role should contain the logic (or simplified logic) of the application. The intruder can also play the PCD role, but he does not have to follow the logic in the role definition, he can perform arbitrary actions. The role definition is good only for the legitimate entity behavior.

Figure 5.13 shows the diagram of a sample payment protocol that will be used to demonstrate the protocol logic modeling. The diagram shows only the communication between two legitimate parties where no error occurs. A flow diagram can be used to better describe the logic of the PCD. The PCD role in ASLan++ should reflect the PCD logic shown in the diagram.

The previously described states reduction of the PICC role will reduce the number of commands by making them more complex. So for example the three-pass authentication followed by the *select* command for selecting application and then by the *read* command will result in only one command combining them together. This fact must be taken into account when translating the model checker results into the applicable attack paths.

Figure 5.14 shows how the PICC role implementation of the protocol may look like when the number of Mifare DESFire commands is reduced only to *read* and *write* in order to reduce model checking execution time. First two parameters of both commands are same. The first parameter is in both cases the address of data to be read or written. Mifare DESFire uses application number, file ID, offset of data in file, and length to address particular data block, so the address will represent the combination of these values. For modeling purposes, each of these combinations will be named according to the variable it will store. So for example the cardholder's name will be stored in application number 1, in file with file ID 1, with offset 0 and length 20; this particular data block address will be named $addressName$ to indicate that this address is used to store the name. Other addresses will be named in the same manner. Addresses not intended to store data will also have some name.

The second parameter $auth(key1, S)$ is an authentication token. It is a session key S encrypted using private key $key1$ ($key1$ is shared between legitimate entities and not known by the intruder). The PICC checks whether S is the current session key (no new authentication) or S is a fresh session key (authentication using $key1$). Every old session key (invoked by replay attack) is rejected by the PICC.

The third parameter in the *write* command is the data to be written encrypted using the session key from the second parameter. The response of the *read* command is the data encrypted using the session key from the second parameter, the response of the *write* command is only a status message. Symmetric encryption of $oldBalance$ using key S is denoted $\{oldBalance\}_S$.

5.4 Attack Definition

In the previous sections the model creation was described. The model is written in ASLan++ language, which can be automatically translated to the ASLan language, which is an input format for the back-end model checkers. The attack definition must be provided for the model checker to find any attack traces. The attack is defined as a condition that should never happen in normal protocol run and that means that the intruder learned something that he should not have learned (confidentiality), or that he changed something that he should not have changed (authentication, integrity). These conditions are defined in the ASLan++ model and then translated to states that mean an attack. If the model checker finds a path to one of the attack states, a possible attack is reported. The attack trace should be evaluated and in case of false positive, refinements should be made to the model. The model checker should

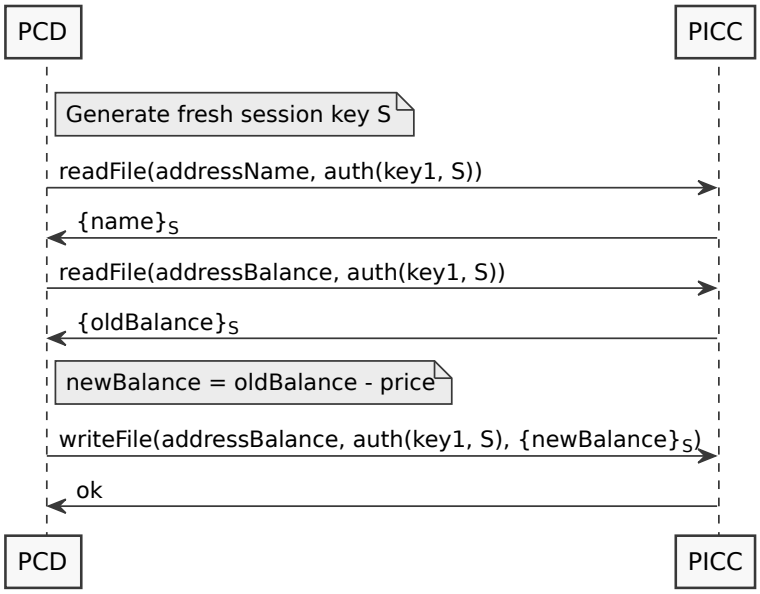

Figure 5.14: Payment protocol with reduced set of commands

be run again and this process should be repeated until real attack is found or the model checker concludes that there is no attack.

Although there are means for defining security goals of confidentiality and authentication in ASLan++, these do not fit well for the purposes of our attack definitions. We will use assertions that will always hold unless an attack is under way. We can easily set goals that the protocol should achieve, covering all desired security goals, by defining assertions in the PCD role that can contain information from PICC which would not be available in real environment, such as content of files (because files are modeled as global facts). Example in figure 5.15 shows an assertion that can be used at some point in the PCD or PICC role to check content of some file on the card.

```
assert ok: fileSystem(addressBalance,key1,key1,newBalance)
```

Figure 5.15: Attack definition in ASLan++

We can interpret this assertion as follows: if the file at address *addressBalance* contains the value *newBalance*, it is ok, otherwise the model checker will stop and an attack will be reported.

Chapter 6

Experimental Results

To demonstrate how the model creation process works and how the security attributes of a protocol can be verified, three examples are provided in this chapter.

The first example introduces a simple payment system that uses a Mifare DESFire like contactless smart card to store some value. The model of communication protocol between the terminal and the card will be created and used as an input for the model checker. The first example uses Mifare DESFire MF3ICD40, the second example uses theoretically improved version of the same card, and the third example uses Mifare DESFire EV1.

Let us suppose that we need to develop a new payment protocol which uses contactless smart cards. The card will be issued to the cardholder personalized with his name and the initial balance. The cardholder will be able to pay for goods with this card. After he pays using the card, the price will be subtracted from the current balance. The balance can be increased by the authorized entity. From these basic requirements we can decide how the payment system should be implemented and create a sequence or flow diagram of the application. The developer should first create the sequence or flow diagram of the protocol and create and optimize the automaton representing the smart card, then he can create the PCD and PICC models, define conditions that represent attacks and verify using model checking. Finally, he can implement the protocol in the target programming language.

Let us create an intuitive protocol that will fulfil the stated requirements. The cardholder's name and balance will be stored on the contactless smart card in files. We decided to use Mifare DESFire as one of the most widespread contactless smart cards. When the cardholder puts the contactless smart card to the proximity of the contactless smart card reader at the point of sale (POS) terminal, the anti-collision procedure is performed and the payment protocol can be executed. The mutual

authentication should be performed at the beginning of the transaction and the data that are then transmitted should be encrypted in both directions. The terminal first reads the cardholder's name and then the balance. If the balance is higher than the price for the goods, the price is subtracted from the balance by the POS terminal and the resulting balance is written to the smart card.

The model of the protocol consists of the PICC role, the PCD role, and the attack definitions, as described earlier. We have combined the concepts of multi-application card, authentication using pre-shared key, encryption in ECB mode, and file system with permissions to create a model of Mifare DESFire contactless smart card, supporting all basic commands required by the protocol. In case of Mifare DESFire MF3ICD40 we have to distinguish between data encrypted using encryption mode of DES and decryption mode of DES, because this type of smart card uses only encryption, while the terminal must use only decryption. Plain data *oldBalance* encrypted using session key S is denoted $enc(S, oldBalance)$, while the same plain data *oldBalance* decrypted using session key S is denoted $dec(S, oldBalance)$. The PCD role was modeled to reflect the sequence diagram of the protocol, which was shown in figure 5.14.

The attack definition consists of integrity and confidentiality checks implemented using assert. There is one assert at the end of the PCD role stating that the balance on the card is equal to the *newBalance* value. In other words, when the protocol is executed successfully and the PCD checks the written balance and comes to a point where it believes that the balance on the card is set to *newBalance*, the actual value on the card is really *newBalance*. This assert can be realized thanks to modeling of files as facts, which are visible globally. Other asserts can be used to check intermediate states of the protocol.

The ASLan++ model was translated to the ASLan format and used as an input for the Cl-Atse model checker. Several model checker runs and protocol adjustments revealed some possible attacks, which are discussed in the following sections. The output of the Cl-Atse model checker is the ATK file containing the sequence of messages leading to a successful attack. These attack traces are also provided together with the corresponding sequence diagrams, which are more illustrative. The attack descriptions are accompanied with informal description for better understanding of how it works. When an attack was found, it was tested on a real card using man-in-the-middle (MITM) hardware described in chapter 4. When the attack was successful, a countermeasure was added to the protocol in order to fix the vulnerability, and another round of model checking was performed. When the attack was not successful

in the real environment, the model was refined and another round of model checking was performed as well. The steps that were performed are described at each sample verification in the following sections.

Source code of the sample verification models in ASLan++ language are provided for reference in the Appendix of this thesis.

6.1 Sample Verification 1 – Mifare DESFire MF3ICD40

The first example uses Mifare DESFire MF3ICD40 contactless smart card with no modifications. Using a straightforward approach to designing a protocol without thinking much about security, the resulting protocol contains some vulnerabilities, which were found using the model checker. The model checker reports an attack that it has found. When the attack was found, we have proposed and implemented a countermeasure and repeated the model checking. All subsequent attacks and countermeasures for the first example are mentioned in the following text. The CL-Atse model checker with parameters *–short –opt –nb 5* was used in this example.

6.1.1 Attack 1

The first attack that was found was caused by the fact that the address of data blocks on the card is not cryptographically protected. The first model checker run revealed the attack trace shown in figure 6.1. Entities `picc` and `pcd` are legitimate, entities `<picc>` and `<pcd>` belong to the intruder.

```
 pcd  -><picc> : readFile(addressName,auth(key1,n119(SessionKey)))
<pcd> -> picc  : readFile(addressBalance,auth(key1,n119(SessionKey)))
 picc -><pcd>  : enc(n119(SessionKey),oldBalance)
<picc>-> pcd   : enc(n119(SessionKey),oldBalance)
```

Figure 6.1: Attack 1 – model checker output

The first line of the attack trace means that the legitimate PCD sends to intruder's PICC the command $readFile$ with address parameter $addressName$ and authentication token $auth(key1, n119(SessionKey))$. It means that in real environment the PCD would first authenticate using $key1$ and select application which contains the $addressName$, and then send $readFile$ command with file ID, length, and offset corresponding to $addressName$. The $n119(SessionKey)$ in the attack trace represents

session key that was freshly generated. The second line means that the intruder's PCD forwards the *readFile* command to the legitimate PICC with the same authentication token, but with changed address. The address *addressBalance* is different from the address in the first command. When using real commands, the difference would be in application number, file ID, length, or offset. Since only two files are defined in this protocol, the attacker would probably be able to change only the file ID to perform the attack successfully. The result of this change is that PICC returns different data than demanded (on the third line), which are then forwarded to the PCD (on the fourth line). The PCD expects name, which is for instance in file with ID 1, but gets balance, which is in file with ID 2. The PCD does not implement any validity check, so it would carry on without any suspicion. At this point, the model contains assert condition to check that PCD gets correct data. In this case this assertion was not satisfied, therefore the model checker stopped and reported the attack. We can also decide that the name is not important and put the assert only after reading balance, which is more important for the protocol. The result would be same, the attacker would change the address. In the real environment the length of the name and balance would be different, so these attacks might not always work as suggested by the model checker. However, the attacker can change more than one parameter in the real command, so he can for instance change the file ID and also the length of data. If the attacker wants to forge balance, which is for example 4 Bytes long, he can set the file ID and offset to some part of name (which is for example 16 Bytes long), and set the length to 4 Bytes, so the length of data read will be 4 Bytes. If these 4 Bytes represent some balance value, the PCD has no means to find out that this data block is not genuine balance and that it is forged by the intruder.

This attack trace is the exact output of the model checker that was saved in the ATK file. To better understand the result, it is good to create a sequence diagram, which is more illustrative. Figure 6.2 shows the output of the model checker translated into the sequence diagram. Intruder's actions causing an attack are shown in bold in all following figures.

The output of the model checker is quite concise due to the reduction of the PICC model and can be translated to the real environment commands and responses. The attack trace which uses data address as one value is then translated to more than one attacks that use separate values for application number, file ID, length, and offset. Figure 6.3 shows an attack based on forged file ID. There are also notes showing where an intruder can change application number, offset, and length.

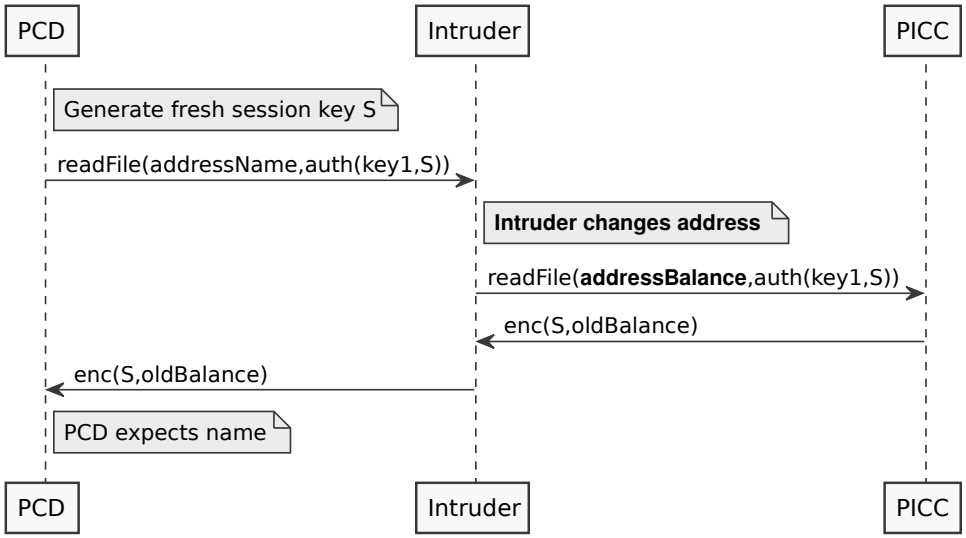

Figure 6.2: Attack 1 based on changing address

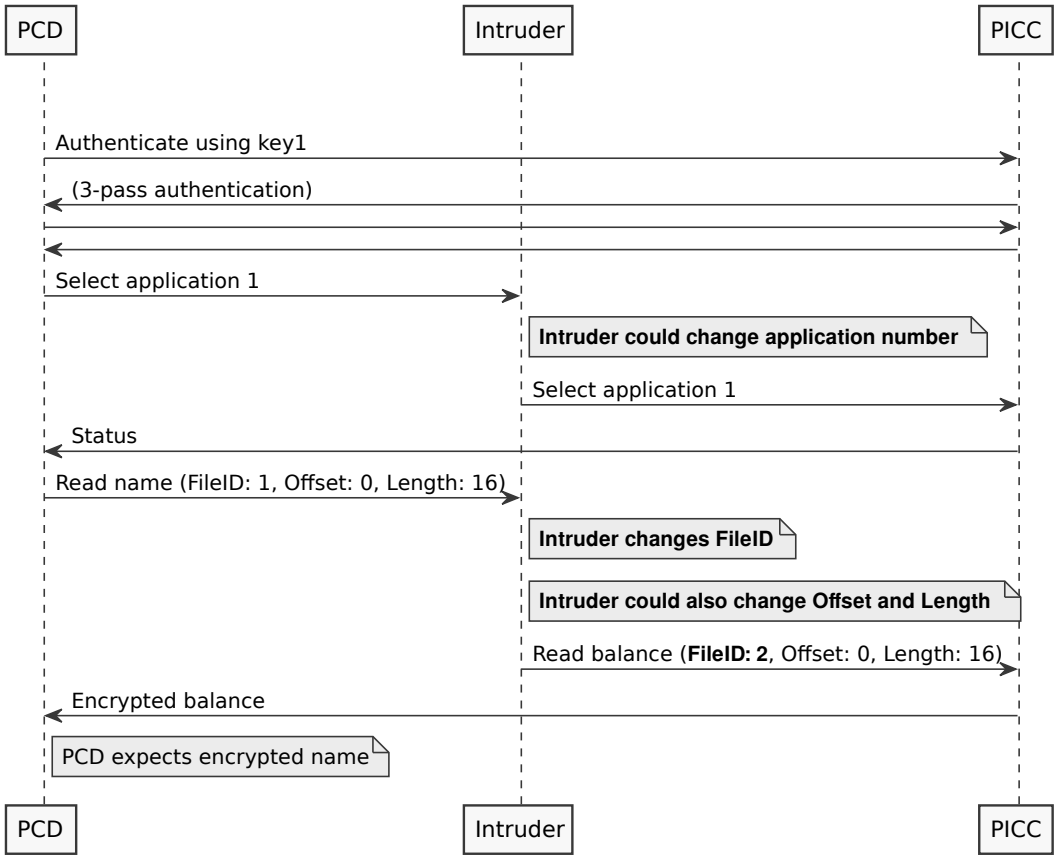

Figure 6.3: Attack 1 – real commands

These attacks are caused by the fact that the address of data blocks on the card are not cryptographically protected. The attacker changes the data address, which consists of the application number, file ID, length, and offset. This results in the PICC returning wrong data block or writing into wrong address. These attacks were described by the author of this thesis in [117].

6.1.2 Testing Attack 1 on a Real Device

To find out whether the found attack trace represents real vulnerability or it is only a false positive, we had to try it in a real environment. We used the MITM hardware to execute the protocol several times and alter parameters of the commands to force PICC to use different application number, file ID, offset, and length. Figure 6.4 shows the native Mifare DESFire MF3ICD40 Application Protocol Data Units (APDU) exchanged between legitimate PCD and PICC. Remarks in the figure describe what command or response is being sent and with what parameters. It also includes the length of each parameter of the command or response. Each APDU is preceded with the direction, either from PCD to PICC or vice versa. Where the intruder acting as a MITM makes any changes to the commands or responses, the direction ends or starts in the entity Intruder and Intruder's action is described in a remark.

As we can see, the attack was successful and the protocol should be improved to fix this vulnerability. The protocol model does not consider the length of data being sent, so we can argue that in some situations the proposed attack would not be possible, because the PCD expects data of some particular length and the intruder is only able to send data of a different length. It could also be the case in this example. On the real card both files have same length – 32 Bytes – and the values are stored at the beginning of the file. The balance in this example has 4 Bytes. To keep the examples short, we only read 8 Bytes of the name. Thanks to the fact that the balance file is not 4 Bytes long but 32 Bytes long, we can read the same number of Bytes from balance as we would read from name. There is therefore no limitation on length. Let us suppose that the length of balance file is only 4 Bytes. Is such a case this attack would not be possible. However, it would still be possible to make the attack the other way round – to change the address so that the PCD would read 4 Bytes from name instead of demanded 4 Bytes from balance. We can extend the model by introducing lengths of data in *enc* and *dec*. The length will be the part of the encrypted data, so the attacker will not be able to alter it and it will be possible for both PCD and PICC to check the length so that they will accept only data with

```
# Select application 1 (Command: 1B, Application number: 3B)
PCD -> PICC: 5a 01 00 00
# OK (Status: 1B)
PICC -> PCD: 00

# Authenticate with key 1 (Command: 1B, Key number: 1B)
PCD -> PICC: 0a 01
# Three-phase authentication
PICC -> PCD: af 5a bd 19 c7 50 5c fb e1
PCD -> PICC: af 49 1e 89 0d e9 ac e9 32 db b9 a6 42 d6 cc d8 4d
PICC -> PCD: 00 b7 d1 da 7c e0 dd 98 6b
# Session key (DES): 00 01 02 03 fe b6 d9 ec

# Read name (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B)
PCD -> Intruder: bd 01 00 00 00 08 00 00
# Intruder changes the address
Intruder -> PICC: bd 02 00 00 00 08 00 00
# Encrypted data (Status: 1B, Data: 16B)
PICC -> PCD: 00 f3 d2 1b d4 09 2b 53 6a 5f 37 51 69 da 09 18 b8
# Decrypted data (Data 8B, CRC: 2B, Padding: 6B)
# 00 00 10 00 00 00 00 00 8a 17 00 00 00 00 00 00
```

Figure 6.4: Attack 1 – APDUs

the expected length.

Figure 6.5 shows the output of the model checker when the data lengths are used. As we can see, the attacker could not change the address of the name because the only way to get long enough block of data is to read from the file containing the name. The file containing the balance is too short (4 Bytes). The attack is based on changing the address of balance in the *readFile* command to *addressNameCorrupted*, which is an imaginary address of some part of file containing name, starting at arbitrary offset in this file and having the length of balance. The lengths of data blocks are defined in the fact definition part of the ASLan++ source file. If the balance is coded for example as unsigned integer, the real world usability of this attack is obvious – the attacker will replace the cardholder's balance with name, which will probably represent much bigger number than the real balance.

As we can see, introducing data block lengths to help PCD to distinguish between legitimate and forged data is not very useful and we cannot consider it as a practical countermeasure. The next subsection describes a countermeasure that can be used to avoid these attacks.

```
 pcd  -><picc> : readFile(addressName,auth(key1,n114(SessionKey)))
<pcd> -> picc  : readFile(addressName,auth(key1,n114(SessionKey)))
 picc -><pcd>  : enc(n114(SessionKey),name,lengthName)
<picc>-> pcd   : enc(n114(SessionKey),name,lengthName)
<pcd> -> picc  : readFile(addressNameCorrupted,
                   auth(key1,n114(SessionKey)))
 pcd  -><picc> : readFile(addressBalance,auth(key1,n114(SessionKey)))
 picc -><pcd>  : enc(n114(SessionKey),corrupted,lengthBalance)
<picc>-> pcd   : enc(n114(SessionKey),corrupted,lengthBalance)
```

Figure 6.5: Attack 1 – with data length

6.1.3 Countermeasure to Attack 1

A countermeasure to the previous attacks can be some integrity checking on the application layer of the payment system. For purposes of integrity checking, CRC or cryptographic signature can be used. CRC would be enough, because all data transmitted between PCD and PICC are encrypted, so the intruder cannot change data nor CRC, which is part of the encrypted data, without corrupting whole data block. If there is for example only one file containing the CRC protected data, the PCD can easily distinguish this valid data block from another data block from another application, file, offset, or with different length. So let us add such integrity checking to the protocol model in the PCD role, which will help PCD to distinguish valid balance from another corrupted data. In the PCD role we will add a validity check after each line in the code where some data are received from PICC. For example a validity check for *name* will look like this:

```
select {
  on(Data = name): {
```

The parentheses must be closed at the end of the PCD role. Thanks to this validity checking in the model the PCD role will continue in the protocol only when it receives data that are not corrupted, which is exactly what would be achieved using some integrity checking method in the real environment. Such validity checking can also be simpler and use the term *corrupted* to ensure that the data received is correct, but it would not distinguish between data types:

```
select {
  on(Data != corrupted): {
```

This validity check will not distinguish between valid name and valid balance, it will only ensure that the data block is built correctly, for example with valid CRC or cryptographic signature.

6.1.4 Attack 2

After implementing the proposed countermeasure, the model checker revealed the attack shown in figure 6.6.

```
 pcd  -><picc> : readFile(addressName,auth(key1,n122(SessionKey)))
<pcd> -> picc  : readFile(addressName,auth(key1,n122(SessionKey)))
 picc -><pcd>  : enc(n122(SessionKey),name)
<picc>-> pcd   : enc(n122(SessionKey),name)
<pcd> -> picc  : readFile(addressBalance,auth(key1,n122(SessionKey)))
 pcd  -><picc> : readFile(addressBalance,auth(key1,n122(SessionKey)))
 picc -><pcd>  : enc(n122(SessionKey),oldBalance)
<picc>-> pcd   : enc(n122(SessionKey),oldBalance)
 pcd  -><picc> : writeFile(addressBalance,auth(key1,n122(SessionKey)),
                  dec(n122(SessionKey),newBalance))
<picc>-> pcd   : ok
```

Figure 6.6: Attack 2 – model checker output

This attack is based on discarding the *writeFile* command by the intruder. The beginning of the communication is the standard protocol execution where the intruder only forwards messages between legitimate PCD and legitimate PICC. At line number 9, the legitimate PCD sends the *writeFile* command in order to store the *newBalance* in the smart card; however, the intruder never forwards this message to PICC. This results in situation where PCD assumes that *newBalance* has been written, but PICC have not received the command, so the balance on PICC remains unchanged. At this point, PCD thinks that the current balance on card is *newBalance*, which is not true.

The model contains assert condition to check that PCD gets correct data. In this case this assertion was not satisfied, so the model checker reported an attack.

Figure 6.7 shows a sequence diagram of an attack on the improved protocol that was found and that is based on discarding *writeFile* command by the intruder.

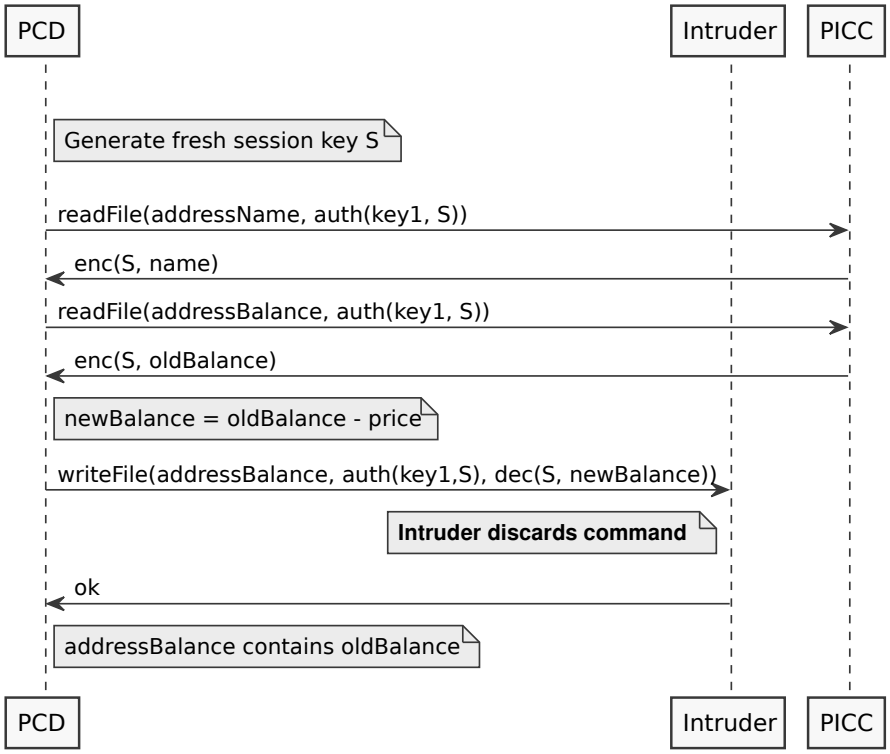

Figure 6.7: Attack 2 based on discarding write command

6.1.5 Testing Attack 2 on a Real Device

To find out whether the found attack trace represents a real vulnerability or it is only a false positive, we had to try it in real environment. We used the MITM hardware to execute the protocol and to discard the *writeFile* command and at the end of the protocol. Figure 6.8 shows the APDU commands exchanged between legitimate PCD and PICC. As in the previous example, remarks in the figure describe what command or response is being sent, its parameters and lengths of these parameters. Each APDU is preceded with the direction, either from PCD to PICC, or vice versa. Where the intruder acting as a MITM makes any changes to the commands or responses, the direction ends or starts in the entity Intruder and Intruder's action is described in a remark.

As we can see, the attack was successful and the protocol should be improved to fix this vulnerability. The next subsection describes a countermeasure that can be used to avoid this attack.

```
# Select application 1 (Command: 1B, Application number: 3B)
PCD -> PICC: 5a 01 00 00
# OK (Status: 1B)
PICC -> PCD: 00

# Authenticate with key 1 (Command: 1 Byte, Key number: 1 Byte)
PCD -> PICC: 0a 01
# Three-phase authentication
PICC -> PCD: af a7 1a 16 94 54 67 21 3c
PCD -> PICC: af 49 1e 89 0d e9 ac e9 32 fe 7b 0d b3 15 0d 29 6b
PICC -> PCD: 00 b7 d1 da 7c e0 dd 98 6b
# Session key (DES): 00 01 02 03 39 db cd 10

# Read name (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B)
PCD -> PICC: bd 01 00 00 00 08 00 00
# Encrypted data (Status: 1B, Data: 16B)
PICC -> PCD: 00 67 2a ec 93 bb 3d da b5 82 5b de d8 a4 38 e3 ff
# Decrypted data (Data 8B, CRC: 2B, Padding: 6B)
# 4a 6f 68 6e 00 00 00 00 8b cd 00 00 00 00 00 00

# Read balance (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B)
PCD -> PICC: bd 02 00 00 00 04 00 00
# Encrypted data (Status: 1B, Data: 8B)
PICC -> PCD: 00 9d 43 54 11 cf 6f cc 9a
# Decrypted data (Data 4B, CRC: 2B, Padding: 2B)
# 00 00 10 00 91 c3 00 00

# Write new balance (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B,
#                    Encrypted data: 8B)
PCD -> Intruder: 3d 02 00 00 00 04 00 00 96 be 10 1c 43 b3 a6 5a
# Intruder discards the write command
# OK (Status: 1B)
Intruder -> PCD: 00
```

Figure 6.8: Attack 2 – APDUs

6.1.6 Countermeasure to Attack 2

A countermeasure to this attack can be reading the balance once again at the end of the protocol to check the written value. If the read balance is correct, the protocol ends and the cardholder can take the goods. Figure 6.9 shows a sequence diagram of the proposed countermeasure.

Thanks to the fact that the Mifare DESFire MF3ICD40 only encrypts using DES or 3DES and the PCD only decrypts, this countermeasure is sufficient. If the data

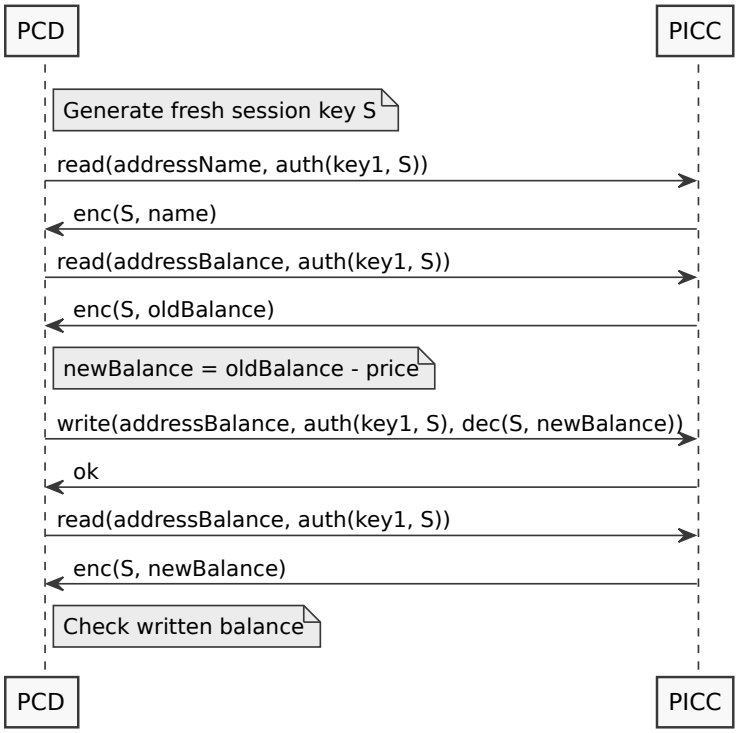

Figure 6.9: Countermeasure to attack 2

transmitted by both PCD and PICC were encrypted and the receiving device always decrypted the data, this proposed countermeasure would not be enough. The intruder would be able to learn the encrypted *newBalance* from the *writeFile* command and then replay this data as the response to the last *readFile* command, so the PCD would be mislead by the attacker. The balance check would successfully pass, but the actual balance on the card would not be changed. The intruder would discard the *writeFile* command so the balance on the card would be *oldBalance* and the replay attack would ensure that the PCD does not find it out. If we change the model in the way that all transmitted data are encrypted, the model checker finds the described attack, which is shown in figure 6.10.

The countermeasure to this attack would be a re-authentication after data writing, which means that data that are read after re-authentication are encrypted using new session key, so the intruder cannot replay the previously eavesdropped encrypted balance. Figure 6.11 shows a sequence diagram of the proposed countermeasure. The re-authentication is simulated by generating fresh session key, which is used in the following communication. This re-authentication is emphasized in boldface in the figure.

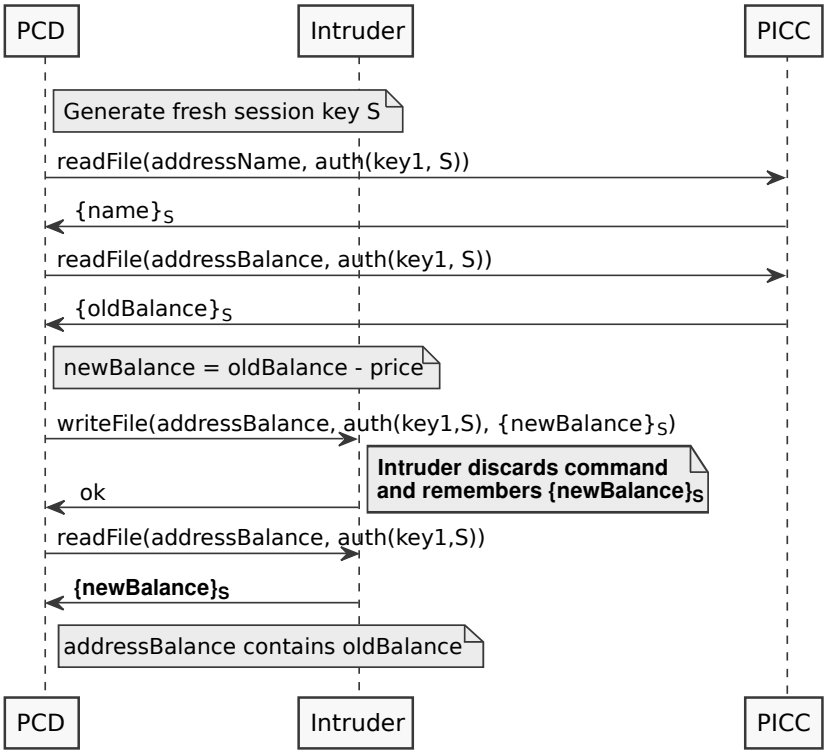

Figure 6.10: Hypothetical attack

Note that in case of Mifare DESFire MF3ICD40 we do not have to consider this attack and we can use the first proposed countermeasure.

6.1.7 Attack 3

After implementing the proposed countermeasure, the model checker revealed the attack in the improved protocol, which is shown in figure 6.12.

This attack is based on changing the address in the *writeFile* command to another valid file or another offset in the same file. The *newBalance* will be saved to another address and then read from this address for checking. The check in PCD will successfully pass; however, the balance file on the card will contain *oldBalance* instead of *newBalance*. The assert in the PCD role used to check that PICC has *newBalance* at the correct address will not be satisfied, therefore the model checker will stop and report the attack.

Figure 6.13 shows a sequence diagram of this attack, intruder's actions are emphasized in boldface.

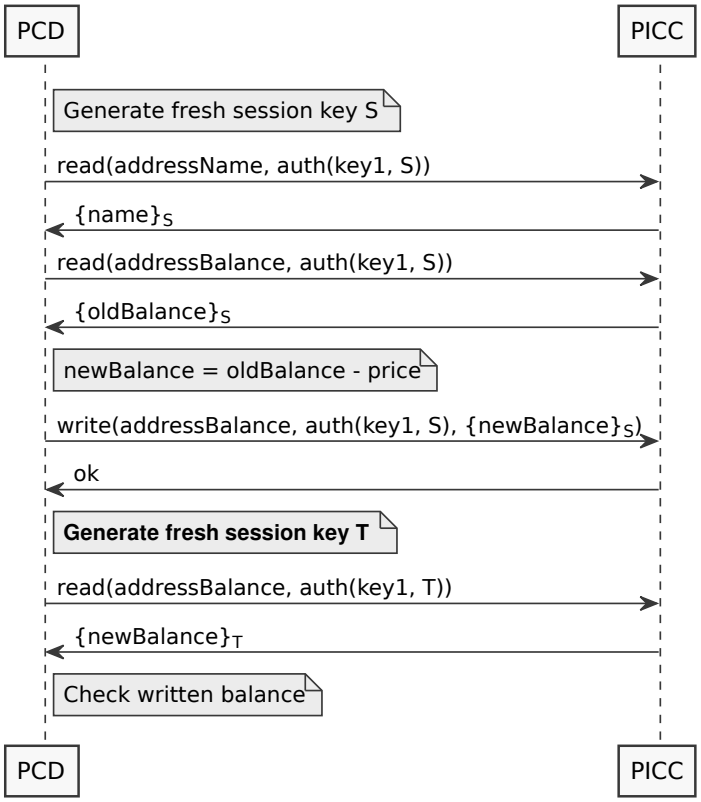

Figure 6.11: Hypothetical attack countermeasure

6.1.8 Testing Attack 3 on a Real Device

To find out whether the found attack trace represents a real vulnerability or it is only a false positive, we had to try it in a real environment. We used the MITM hardware to execute the protocol and to change the *writeFile* command parameters to save *newBalance* to a different address. Then we changed the *readFile* command parameters so that the PICC would read *newBalance* instead of the actual value at the balance address. Figure 6.14 shows the APDU commands exchanged between legitimate PCD and PICC. As in previous examples, remarks describe commands with their parameters and responses.

As we can see, the attack was successful and the protocol should be improved to fix this vulnerability. The next subsection describes a countermeasure that can be used to avoid this attack.

```
 pcd  -><picc> : readFile(addressName,auth(key1,n120(SessionKey)))
<pcd> -> picc  : readFile(addressBalance,auth(key1,n120(SessionKey)))
 picc -><pcd>  : enc(n120(SessionKey),actualBalance)
<pcd> -> picc  : readFile(addressNameCorrupted,
                  auth(key1,n120(SessionKey)))
 picc -><pcd>  : enc(n120(SessionKey),corrupted)
<pcd> -> picc  : readFile(addressName,auth(key1,n120(SessionKey)))
 picc -><pcd>  : enc(n120(SessionKey),name)
<picc>-> pcd   : enc(n120(SessionKey),name)
 pcd  -><picc> : readFile(addressBalance,auth(key1,n120(SessionKey)))
<picc>-> pcd   : enc(n120(SessionKey),actualBalance)
 pcd  -><picc> : writeFile(addressBalance,auth(key1,n120(SessionKey)),
                  dec(n120(SessionKey),newBalance))
<pcd> -> picc  : writeFile(addressBalanceCorrupted,
                  auth(key1,n120(SessionKey)),
                  dec(n120(SessionKey),newBalance))
 picc -><pcd>  : ok
<pcd> -> picc  : readFile(addressBalanceCorrupted,
                  auth(key1,n120(SessionKey)))
 picc -><pcd>  : enc(n120(SessionKey),newBalance)
<picc>-> pcd   : ok
 pcd  -><picc> : readFile(addressBalance,auth(key1,n120(SessionKey)))
<picc>-> pcd   : enc(n120(SessionKey),newBalance)
```

Figure 6.12: Attack 3 – model checker output

6.1.9 Countermeasure to Attack 3

A countermeasure to this attack can be allowing writing to only one file and restricting writing to all other files. This is very strong restriction and will affect the usability of the protocol. However, after executing the model checker on the improved protocol, no more attack was found.

6.1.10 Conclusion

The cause of all these attacks is the weakness of Mifare DESFire MF3ICD40 contactless smart card – not encrypting or signing commands, application number, file ID, length, and offset.

Note that the size of files and data being transferred between PCD and PICC is ignored in the basic model, so the model checker will sometimes find an attack which is not feasible in real environment due to the size limitations. These false positives can be avoided by also including size of files and data in the model as shown in the

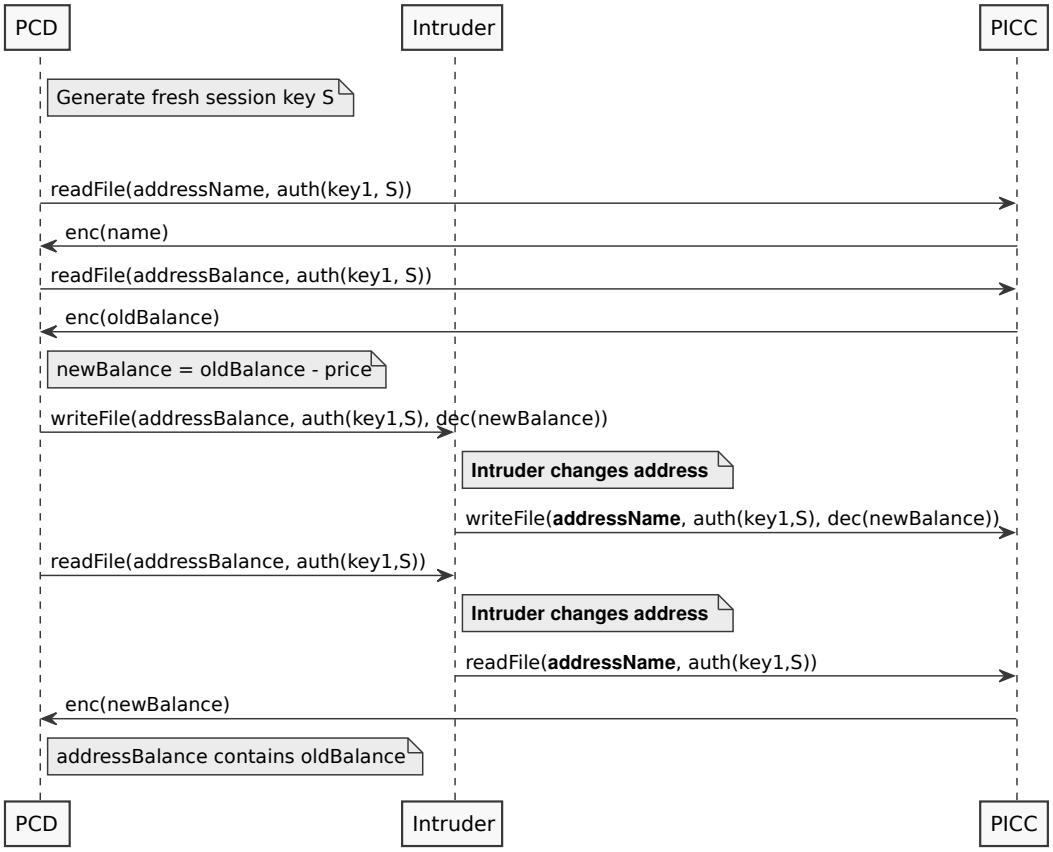

Figure 6.13: Attack 3 based on writing to another file

first attack in this example.

This example contains several countermeasures and model adjustments, so several source codes were used as input for model checking. The source code provided in the Appendix is a sample implementation which shows the proposed countermeasures in comments. The intention was to provide one source code for each example; however, not everything described in this example is included due to the complexity of the combined source code.

6.2 Sample Verification 2 – Improved smart card

The second example of attack finding using formal verification methods is based on a protocol, which is similar to the protocol in the previous example, but which is using an improved contactless smart card. The contactless smart card used is a hypothetical card similar to Mifare DESFire with the difference that everything in the communication is encrypted (after successful authentication), not only data. This

improvement will prevent attacks from the previous example. Another small change is that anyone can change the name on the smart card; this field is only informative, so there is no need to protect it. This will be modeled using $key2$, which will be known to the intruder and which will be required for granting write permission. The balance field will be protected in the same way as in the previous example, using $key1$, which is not known by the intruder. Figure 6.15 shows a sequence diagram of the protocol used in this example. The command modeling is different from the previous example, $encryptedCommand$ is a tuple containing the authentication token $auth(key1, S)$ and the command encrypted using the symmetric session key, such as $\{readFile(addressName)\}_S$. Everyone, including the intruder, can learn these two components from the $encryptedCommand$. The authentication does not depend on the content of the encrypted part of the $encryptedCommand$. This model represents encryption of whole commands using ECB mode, so commands can be replayed by the intruder. However, the intruder is not able to find out the content of the encrypted part, he cannot find out which command it is.

The Cl-Atse model checker with parameters *–short –opt –nb 3* was used in this example. The model checker found an attack, which is shown in 6.16.

We call this a "command injection" attack. An attacker authenticates using the publicly known $key2$ and writes the forged command to the field at address $addressName$. Then, during the protocol run initiated by the legitimate PCD, when the PCD reads the name field at address $addressName$, an attacker (man-in-the-middle) eavesdrops the forged command encrypted using current session key (not known be the attacker). He can then send the encrypted forged command to the PICC and the PICC cannot find out that it was not sent by the legitimate PCD, because it is encrypted using the current session key that was established during the authentication using $key1$, which is known only to the legitimate PCD. This is a kind of replay attack, because the intruder overhears the command in the form of encrypted data from PICC and replays the same encrypted data to the PICC as new command.

Despite the fact that commands and their parameters are encrypted, the intruder can execute arbitrary command, he only has to prepare it in advance. Figure 6.17 shows a sequence diagram of the command injection attack found by the model checker with the imaginary command $maliciousCommand$, which is used to emphasize the fact that arbitrary malicious command can be injected, not only the command suggested by the model checker, which is $writeFile(addressBalance, falseBalance)$. Figure 6.18 shows the same attack using real world commands. Intruder's actions are

stressed in bold in both diagrams.

This attack could not be tested in real environment, because the smart card is only a hypothetical improvement of Mifare DESfire. However, we can suppose that this attack would work.

The countermeasure to this attack can be any replay attack protection which will ensure the freshness of the messages. Any encryption mode different from the default ECB mode would prevent this attack. We did not implement this countermeasure as part of this example, because the next example focuses on a smart card which uses CBC encryption mode and we can see there that the CBC mode prevents any replay attacks.

6.3 Sample Verification 3 – Mifare DESFire EV1

In this example the protocol uses Mifare DESFire EV1, which is an improved version of Mifare DESFire MF3ICD40. There are many improvements implemented in this card, the most important from the protocol modeling perspective is the encryption in CBC mode. The older Mifare DESFire MF3ICD40 also used CBC mode, but only inside one command, the initialization vector (IV) was set to zero after each command and response. Mifare DESFire EV1, on the other hand, uses real CBC mode and the IV is set to zero only at the beginning of the communication and then never again. This approach helps to prevent any kind of replay attack. The encrypted part of the message always includes CRC32 of the whole command, so nothing can be altered by the intruder, which is also a huge improvement compared to the older version.

The CBC encryption mode of the Mifare DESFire EV1 is modeled by adding an IV to the encrypted part of the tuple *encryptedCommand*, which was used in the previous example. The tuple *encryptedCommand* will contain the authentication token $auth(key1, S)$ and the command together with IV encrypted using symmetric session key. The zero IV is denoted $zeroIV$, the following IVs are denoted $nextIV(zeroIV)$, $nextIV(nextIV(zeroIV))$, etc. For the sake of simplicity of the PICC role implementation in ASLan++, the first IV which is used is $nextIV(zeroIV)$, not $zeroIV$, but this is only a small detail. The encrypted data will be different each time, which will prevent replay attacks.

Figure 6.19 shows a sequence diagram of the protocol being verified in this example. It is simpler than the protocol from the previous examples, the balance is read, updated, and written back to the smart card.

The Cl-Atse model checker with parameters *–short –opt –nb 3* was used in this

example. The model checker found an attack, which is shown in figure 6.20.

The attack found in the first model checker round is based on discarding the write command. It relies on the fact that the PCD does not wait for the authenticated answer from the PICC. The intruder gets the write command from PCD but does not forward it to the PICC. The balance on the PICC does not get updated, while PCD expects that new balance is stored on the card.

When this attack is executed in the real environment, it is found that it does not work, because PCD backtracks the transaction when it does not get any response from PICC in some time. The model is therefore refined by adding the following line to the PCD role, which will make PCD wait until it gets a response from PICC:

```
B -> Actor: ?;
```

The attack shown in figure 6.21 was found by the model checker in the refined model.

This attack is based on the fact that the intruder sends some dummy response. It was found that this attack is again not feasible in the real environment, because PCD checks that the response was *ok*, not arbitrary data. The model is refined accordingly:

```
B -> Actor: enc(?, ?, ok);
```

The attack shown in figure 6.22 was found by the model checker in the refined model.

The correct status *ok* is sent as a response to the PCD in this attack. However, it is not encrypted using correct session key. The real Mifare DESFire EV1 smart card sends the status in plaintext with CRC32 of the whole message encrypted using session key. It is therefore possible to change status so that it looks correctly, but after decrypting and verifying CRC32 one can find out that it is not legitimate. This is the case in our example, testing the attack on real hardware shows that the PCD verifies the CRC32 of the message, therefore another refinement must be performed, which reflects the fact that the intruder must know the session key in order to build correct response:

```
B -> Actor: enc(SessionKey, next(nextIV(nextIV(nextIV(zeroIV)))),
```

After this refinement the model checker did not find any other attack.

```
# Select application 1 (Command: 1B, Application number: 3B)
PCD -> PICC: 5a 01 00 00
# OK (Status: 1B)
PICC -> PCD: 00

# Authenticate with key 1 (Command: 1 Byte, Key number: 1 Byte)
PCD -> PICC: 0a 01
# Three-phase authentication
PICC -> PCD: af ed d4 96 60 88 1a 4d 97
PCD -> PICC: af 49 1e 89 0d e9 ac e9 32 36 60 a6 0e bf a2 d3 be
PICC -> PCD: 00 b7 d1 da 7c e0 dd 98 6b
# Session key (DES): 00 01 02 03 05 7f d8 33

# Read name (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B)
PCD -> PICC: bd 01 00 00 00 08 00 00
# Encrypted data (Status: 1B, Data: 16B)
PICC -> PCD: 00 9f e5 60 b4 e4 59 01 76 67 54 05 ab 93 92 24 39
# Decrypted data (Data 8B, CRC: 2B, Padding: 6B)
# 4a 6f 68 6e 00 00 00 00 8b cd 00 00 00 00 00 00

# Read balance (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B)
PCD -> PICC: bd 02 00 00 00 04 00 00
# Encrypted data (Status: 1B, Data: 8B)
PICC -> PCD: 00 ca c2 72 78 64 90 5a fd
# Decrypted data (Data 4B, CRC: 2B, Padding: 2B)
# 00 00 10 00 91 c3 00 00

# Write new balance (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B,
#                    Encrypted data: 8B)
PCD -> Intruder: 3d 02 00 00 00 04 00 00 dc 87 f5 e0 77 78 25 70
# Intruder changes the address
Intruder -> PICC: 3d 01 10 00 00 04 00 00 dc 87 f5 e0 77 78 25 70
# Decrypted data (Data 4B, CRC: 2B, Padding: 2B)
# 00 00 08 00 c0 98 00 00
# OK (Status: 1B)
PICC -> PCD: 00

# Read balance to check
# (Command: 1B, File ID: 1B, Offset: 3B, Length: 3B)
PCD -> Intruder: bd 02 00 00 00 04 00 00
# Intruder changes the address
Intruder -> PICC: bd 01 10 00 00 04 00 00
PICC -> PCD: 00 07 e5 a0 95 61 1c ba 14
# Decrypted data (Data 4B, CRC: 2B, Padding: 2B)
# 00 00 08 00 c0 98 00 00
```

Figure 6.14: Attack 3 – APDUs

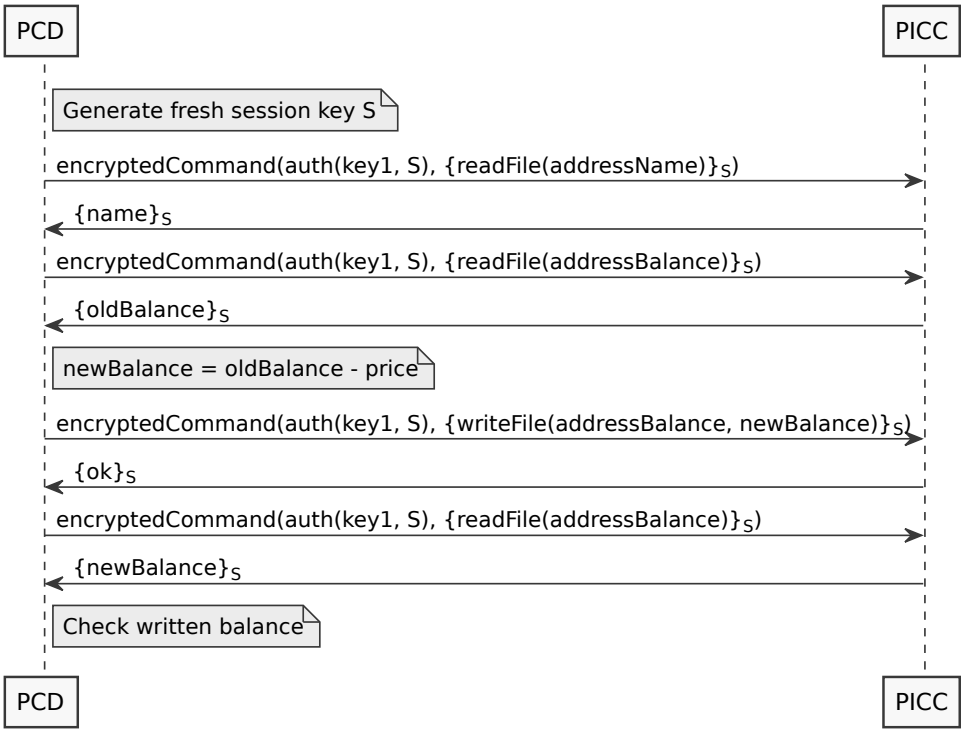

Figure 6.15: Protocol

```
 pcd   -> <picc> : encryptedCommand(auth(key1,n119(SessionKey)),
                     enc(n119(SessionKey),readFile(addressName)))
<pcd>  ->  picc  : encryptedCommand(auth(key2,SessionKey(161)),
                     enc(SessionKey(161),writeFile(addressName,
                     writeFile(addressBalance,falseBalance))))
<picc> ->  pcd   : enc(n119(SessionKey),readFile(addressName))
 picc  -> <pcd>  : enc(SessionKey(161),0)
<pcd>  ->  picc  : encryptedCommand(auth(key1,n119(SessionKey)),
                     enc(n119(SessionKey),readFile(addressName)))
 picc  -> <pcd>  : enc(n119(SessionKey),writeFile(addressBalance,
                     falseBalance))
<pcd>  ->  picc  : encryptedCommand(auth(key1,n119(SessionKey)),
                     enc(n119(SessionKey),writeFile(addressBalance,
                     falseBalance)))
```

Figure 6.16: Model checker output

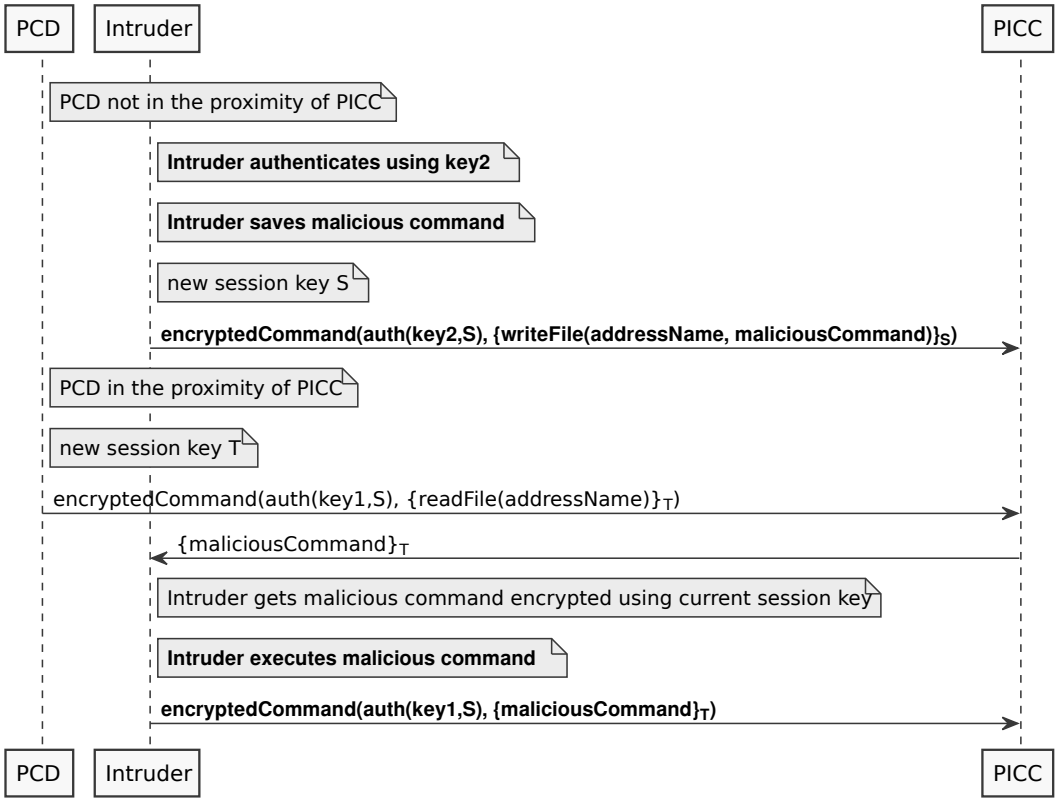

Figure 6.17: Command injection attack

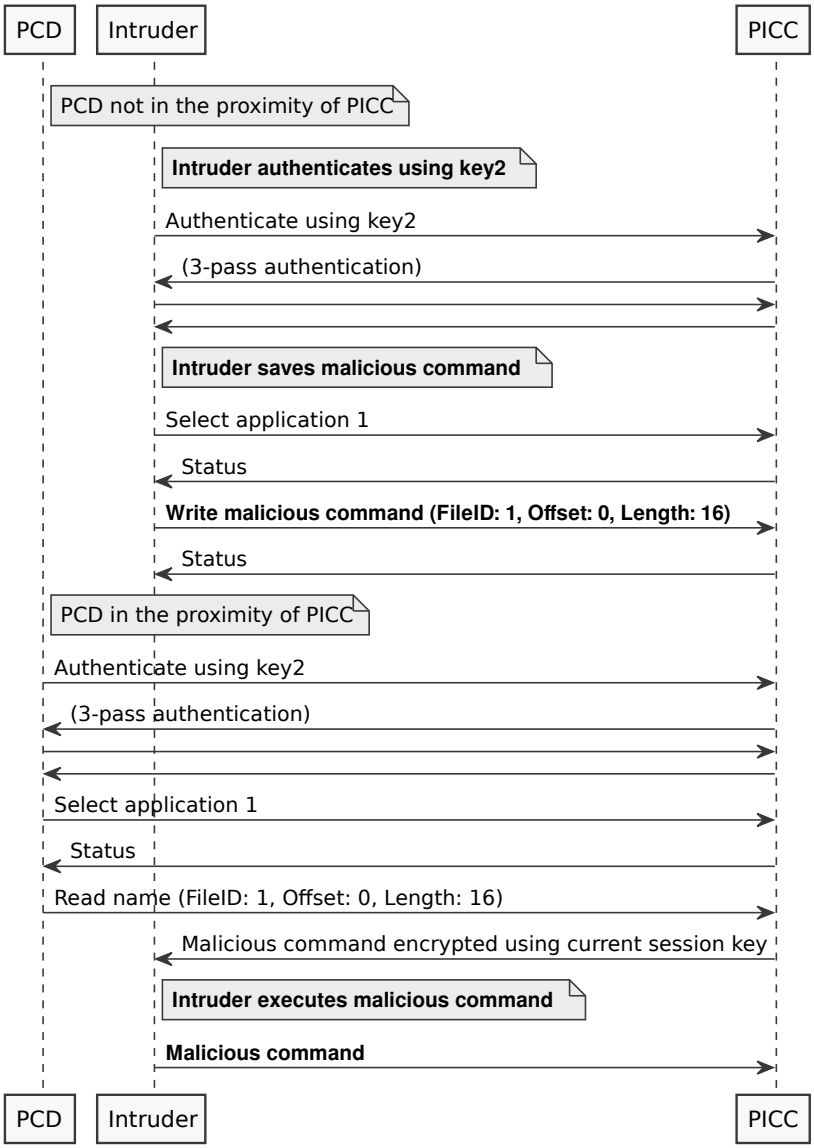

Figure 6.18: Command injection attack – real commands

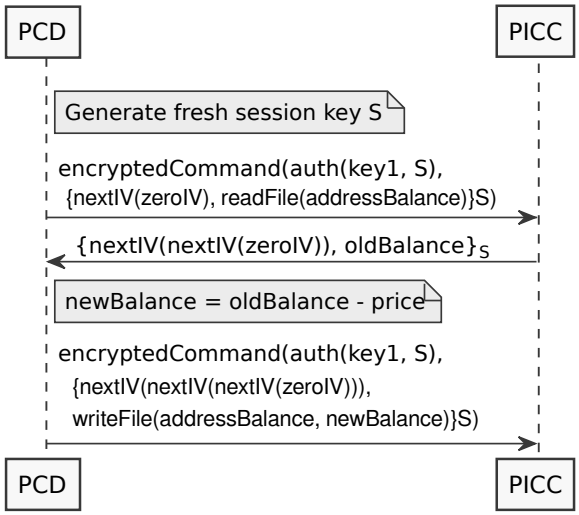

Figure 6.19: Protocol

```
 pcd  -><picc> : encryptedCommand(auth(key1,n111(SessionKey)),
                  enc(n111(SessionKey),nextIV(zeroIV),
                  readFile(addressBalance)))
<pcd> -> picc  : encryptedCommand(auth(key1,n111(SessionKey)),
                  enc(n111(SessionKey),nextIV(zeroIV),
                  readFile(addressBalance)))
 picc -><pcd>  : enc(n111(SessionKey),nextIV(nextIV(zeroIV)),
                  oldBalance)
<picc>-> pcd   : enc(n111(SessionKey),nextIV(nextIV(zeroIV)),
                  oldBalance)
 pcd  -><picc> : encryptedCommand(auth(key1,n111(SessionKey)),
                  enc(n111(SessionKey),nextIV(nextIV(nextIV(zeroIV))),
                  writeFile(addressBalance,newBalance)))
```

Figure 6.20: Model checker output 1

```
 pcd  -><picc> : encryptedCommand(auth(key1,n112(SessionKey)),
                  enc(n112(SessionKey),nextIV(zeroIV),
                  readFile(addressBalance)))
<pcd> -> picc  : encryptedCommand(auth(key1,n112(SessionKey)),
                  enc(n112(SessionKey),nextIV(zeroIV),
                  readFile(addressBalance)))
 picc -><pcd>  : enc(n112(SessionKey),nextIV(nextIV(zeroIV)),
                  oldBalance)
<picc>-> pcd   : enc(n112(SessionKey),nextIV(nextIV(zeroIV)),
                  oldBalance)
 pcd  -><picc> : encryptedCommand(auth(key1,n112(SessionKey)),
                  enc(n112(SessionKey),nextIV(nextIV(nextIV(zeroIV))),
                  writeFile(addressBalance,newBalance)))
<picc>-> pcd   : Dummy(118)
```

Figure 6.21: Model checker output 2

```
 pcd  -><picc> : encryptedCommand(auth(key1,n115(SessionKey)),
                  enc(n115(SessionKey),nextIV(zeroIV),
                  readFile(addressBalance)))
<pcd> -> picc  : encryptedCommand(auth(key1,n115(SessionKey)),
                  enc(n115(SessionKey),nextIV(zeroIV),
                  readFile(addressBalance)))
 picc -><pcd>  : enc(n115(SessionKey),nextIV(nextIV(zeroIV)),
                  actualBalance)
<picc>-> pcd   : enc(n115(SessionKey),nextIV(nextIV(zeroIV)),
                  actualBalance)
 pcd  -><picc> : encryptedCommand(auth(key1,n115(SessionKey)),
                  enc(n115(SessionKey),nextIV(nextIV(nextIV(zeroIV))),
                  writeFile(addressBalance,newBalance)))
<picc>-> pcd   : enc(Dummy(121),Dummy_1(121),ok)
```

Figure 6.22: Model checker output 3

Chapter 7

Protocol Modeling Limitations

7.1 Attacks not Covered

Although formal verification methods are useful for finding vulnerabilities on the protocol level, the usability of this technique on other attacks on contactless smart cards is limited. Other attacks, such as physical attacks, side-channel attacks, and attacks specific for contactless communication are out of scope of this method, since this method is not suitable for them and there is no way how to model properties that would be necessary to find such attacks.

In this chapter another method that can increase the security of contactless smart cards is proposed. This method is focused on possible attack that is not covered in the protocol modeling method and cannot be found using formal verification, because it is an attack on low level communication, where timing is of importance.

This chapter is dedicated to preventing relay attacks, which is a type of attack that cannot be prevented on the application level. Relay attacks are possible due to the contactless communication link and were described in section 2.3.4. Two countermeasures are proposed in this chapter. These methods can be used to prevent real attacks that induce delays significantly longer than the delay caused by the time travelling longer distance. They can be used against most likely attacks, which are not expensive and can be easily performed by attackers with moderate skills, which makes them very dangerous.

7.2 Relay Attack

7.2.1 How to Perform a Relay Attack

This section discusses relay attacks from the perspective of attackers in order to get better idea what possibilities the attackers have in terms of used hardware, what are the time restrictions that they must not exceed, and what tricks they can use to shorten the delay caused by the relay attack.

Hardware

The attackers can use either off-the-shelf equipment, or build their own. They can buy standard contactless smart card readers or any NFC enabled device, such as smartphone. As speed is the most important requirement on relaying the communication, the right hardware choice is crucial.

The first possibility is to use off-the-shelf smart card readers. The example of software which can be used is *libnfc* [118] – a library providing users with the possibility of using two standard contactless readers for a relay attack. The advantage of this approach is obvious, the attacker can use cheap off-the-shelf hardware and only write the software or use some existing. However, this approach can be very tricky, since the attacker has no control over the physical layer. Most readers cannot be used for relay attack due to timing issues induced by PC/SC interface. The off-the-shelf readers are connected to the PC via RS232 or USB, this communication link induce significant delays.

Another possibility is to use NFC enabled smartphones. This is currently popular topic due to the fact that smartphones are light and affordable for most people and can easily establish communication link between each other via bluetooth, Wi-Fi, GSM, etc. Relay attack using NFC devices can be potentially great threat because, unlike self-built hardware and PC connected readers, these devices are hard to be spotted. NFC enabled smartphones were used for relay attack for example in [119] and [120]. The former paper presents the attack which can be used only to forward communication between two other NFC smartphones in peer-to-peer mode. The latter paper goes further and presents relay attack applicable also to communication between active and passive devices, so the communication between reader and card can be forwarded. This was the first time something like this was successful; however, the attacker does not fully control the physical layer, so it cannot be used in some scenarios. They were not able to set arbitrary UID to the fake passive target, which

would be required in systems where the UID is validated. There are two types of UIDs – unique and random (used to ensure untraceability). Random UIDs always start with 0x08. NFC devices can use both types, but they cannot change the unique UID at the moment. Because the communication between two NFC devices is over network where the communication is buffered, there are also significant delays induced.

Adversaries can also build their own equipment. In [121], a practical relay attack with self-built hardware was demonstrated. The communication was relayed up to a distance of 50 m. There were many other successful implementations, since in past this was the only way to perform a relay attack. The advantage of this approach is that the attacker has full control over the physical layer of the communication. The greatest disadvantage is the complexity of this approach, building such hardware requires appropriate knowledge and equipment. The fact that each attacker has to build own hardware rather than using already existing one prevents the spreading of this type of attack. But the full hardware control is unexceptionable advantage. This is currently the only possibility to make the relay attack fast enough to compete with distance bounding protocols.

There is also a possibility to use a special hardware designed for research purposes, such as [122] or [123], which can be purchased assembled and ready to use. These devices are open hardware designs and the attacker has full control over the physical layer. These devices lack full software support, since they are used for experiments rather than for commercial purposes.

Round-trip Time Restrictions

ISO/IEC 14443 defines the *Frame Waiting Time* (FWT), which is the maximal time the reader waits for the card's response. When this time is exceeded, the error is detected and the reader tries to recover. After a few attempts the card is rejected. The forwarded communication must be therefore fast enough to not exceed this limit. The FWT is negotiated at the beginning of the communication. It is computed from the Frame Waiting Integer (FWI), which is part of the Answer to Select (ATS) set by the card.

$FWT = (256 \times 16/fc) \times 2^{FWI}$

In the formula above, fc is the carrier frequency 13.56 MHz. Typical FWT value for DESFire card is 77.33 ms (FWI = 8). The FWT can be set up to 4.949 s according to the standard (FWI = 14). However, the FWT does not affect the anti-collision procedure.

The time for the card's response can be extended by the *Frame Waiting Time Extension* (WTX), which is an S-block defined in ISO/IEC 14443-4. The card uses WTX instead of the answer when it needs more time than the defined FWT to process the received block. The reader then extends the waiting time by the time defined in the WTX. The upper limit of this extension is same as for FWT. WTX can be used repeatedly. If the standard time is not sufficient in the particular situation and we cannot or do not want to negotiate longer FWT for some reason, we can extend the time with WTX.

The Frame Waiting Time can be very relaxed so the attackers can meet the time requirements even if the communication is relayed over network.

Another time restriction can be caused by some distance bounding protocol. Distance bounding protocols were described in section 2.3.4. The time limit in distance bounding protocol would be much shorter and the attackers would not be able to relay the communication, unless they used some trick to reduce the response time, such as one of the methods described in the following subsection.

Reducing Response Time

The attacker can reduce the response time in the relay attack by overclocking the forged reader in order to get the response from the smart card faster than the legitimate reader would get it. It is possible due to the fact that the smart card's processor is clocked by the signal generated by the reader. This would give the attacker a chance to reduce the round-trip time and not exceed the time limit defined in the distance bounding protocol or Frame Waiting Time.

Another possibility how to decrease the response time during relay attack is the Late-commit attack [124] based on the fact that receivers integrate the signal amplitude over whole bit period. During the initial $\frac{m-1}{m}$ of the time interval the attacker sends no energy and for the final $\frac{1}{m}$ of the interval sends m-times stronger signal. The result of the integration will be same as in normal one bit transmission. However, the attacker can delay deciding which value to send by $\frac{m-1}{m}$ of the bit period. Using this method, the attacker can slightly mitigate the effect of delays caused by relay attack.

7.2.2 Delays in Relay Attacks Over Buffered Connection

This subsection discusses delays in relay attacks that use a buffered connection, such as a network, for communication between attackers. These delays are much longer and could be detected even when no distance bounding protocol is used. Real attacks

are not perfect and induce additional delay to the delay caused by the signal travelling longer distance, which is the delay the distance bounding protocols are used to detect. This delay is caused by hardware components processing the signal and sending it to a different location. If the communication is relayed over a distance exceeding the range of one transmitter, it is likely that some buffering will be used. If the data are sent over network using TCP/IP, the induced delay will be significant.

The attacker can reduce the response time in the relay attack by overclocking the forged reader in order to get the response from the smart card faster than the legitimate reader would get it. This would give the attacker a chance to reduce the round-trip time and not exceed the time limit defined in the distance bounding protocol.

Delays that occur when performing a relay attack are caused by three main factors:

- Speed of light is not infinite, therefore an additional delay is induced, corresponding to the time needed for the signal to travel between attackers' devices.
- The hardware used as a fake reader and a fake card needs non-zero time for execution of the required operations.
- The communication between attackers' devices over long distance will likely require buffering, which inherently delays the communication.

The state-of-the-art distance bounding protocols aim at the delay caused by the speed of light. The signal needs more time to travel between legitimate devices when the relay attack is performed than it would in normal situation. Distance bounding protocols therefore try to prevent all theoretical attacks, because nothing can travel faster than light. The speed of light is approximately 3×10^8 m/s, so the signal passes the distance of a kilometer in the order of microseconds. As we will see, the limitation by speed of light, which is the main target of state-of-the-art distance bounding protocols, is negligible compared to the other delays.

Relay attacks are dangerous when they are executed over longer distances, so that the legitimate user can not find out something is happening. These distances will likely exceed the range of one transmitter, so the attackers' devices have to be connected via multiple intermediate re-transmitters or over network. Data in networks are transported in packets, which requires buffering. Latency in computer networks based on TCP/IP is measured in order of milliseconds, which is much higher than delays due to the signal travelling the distance at speed of light. It would be possible to construct re-transmitters that would not buffer any data, and modulate

the incoming signal directly to the outgoing signal. The real implementation of such re-transmitter would also induce delay on electronic components, but it would be significantly lower than delays in buffered communication.

Experimental Relay Attack

We have established a real relay attack using Proxmark 3, which is an open hardware platform for RFID research purposes. Description of the hardware platform was provided in section 4.1. This experimental relay attack was performed for purposes of the protocol analysis described in previous chapters, so there was no effort to make it as fast as possible. The measurement was also not precise. However, it showed that the relay attack delay can be in order of milliseconds. The average measured delay induced by a relay attack was about 27 ms. Proxmark responded fast enough, the communication was delayed mostly by the USB link between Proxmark and PC. Other relay attack implementations can be faster, but communication over USB or network slows it down significantly.

7.3 Relay Attack Mitigation

We propose a method to prevent real-world attacks that induce delays significantly longer than the delay caused by the time travelling longer distance. This method is described in the first subsection. In the second subsection we show a method that is a countermeasure to the overclocking attacks. The method is based on overclocking the legitimate reader to the limit the communicating card can still reliably operate, which reduces to minimum the time the attacker can gain by overclocking the forged reader. We have implemented the overclocking method in the reader and show the results. The signal was analysed on the oscilloscope. The communication time was reduced while the card was still able to reliably operate.

7.3.1 Passive Detection

The reader can monitor the communication and detect anomalies. It does not make any changes to the transmitted signal or data being sent, so we call it passive detection. Alternatively, the reader monitoring can be provided by an external device such as Proxmark 3, which can be used to eavesdrop on the communication and which provides precise timing data.

This method can be used against relay attacks where significant delays are induced for instance by buffered communication link between attackers' devices. The passive detection is based on precise measuring the responses of all commands. Initially, the fingerprint of each type of smart card is made, all response times are measured and saved for later use. During the communication, all response times are continuously measured and compared to the times saved in the smart card's fingerprint. In case of any anomaly, the possible attack is reported.

Additionally, the reader should have much shorter delay restrictions. The Frame Waiting Time should be restricted to minimal values for which the smart card can operate reliably, and the Frame Waiting Time Extension should be disabled by default and allowed only in reasonable situations.

The relay attack over short distance performed with custom made hardware would not be detected by passive detection. However, attacks over computer network or attacks using off-the-shelf USB readers could be detected, because they induce much bigger delays, as discussed in the previous section. These attacks are not expensive and can be easily performed by attackers with moderate skills, which makes them very dangerous. This countermeasure is quite easy to implement compared to distance bounding protocols. It can be worth implementing such countermeasure even if it

does not protect against all theoretical attacks, because it protects against the most likely attacks.

7.3.2 Overclocking

As mentioned earlier, attackers can reduce the round-trip time by overclocking the communication with the legitimate smart card, while communicating on the normal frequency of 13.56 MHz with the legitimate reader. The result is that they get the response from the card faster that the legitimate reader would get it, so they can send the response back sooner than the reader expects and reduce the delay caused by the relay attack. The distance bounding protocol could therefore be circumvented.

The proposed method is based on overclocking the legitimate reader to frequency as high as possible, where the smart card is still reliably operating, which reduces chances for the attackers to perform successful relay attack. The timing is shown in figure 7.1. The first row depicts the time the ordinary communication takes. This is the time the attacker must not exceed in order to keep the relay attack undetected by the round-trip time measurements. The second line shows the relay attack time, which consists of the delay caused by the relay attack, which is the time of flight of the signal and delays on intermediate devices, and time needed by the attacker to execute the command, which is equal to the time needed in the standard communication. In this case the total time exceeds the time of the standard communication. The third line is the case of overclocking attack, which reduces the time of the command execution by the attacker. In this situation the total time is same as the time of the standard communication, which will likely make the relay attack successful. The last line shows the proposed method of overclocking the legitimate reader, which will result in reducing the time of the standard communication, establishing new time limit. So even if the attacker is overclocking the communication with the legitimate card as well, he will exceed the new time limit.

Implementation with Proxmark

We have implemented the reader that communicates with the smart card on the frequency 16 MHz using Proxmark 3. Figure 7.2 compares the response times between standard communication at 13.56 MHz and communication of our overclocked reader running at 16 MHz. Mifare DESFire smart card was used and the depicted command is the polling command, which is periodically sent by the reader. By increasing the frequency, approximately $53\mu s$ was spared on this basic command. The response is

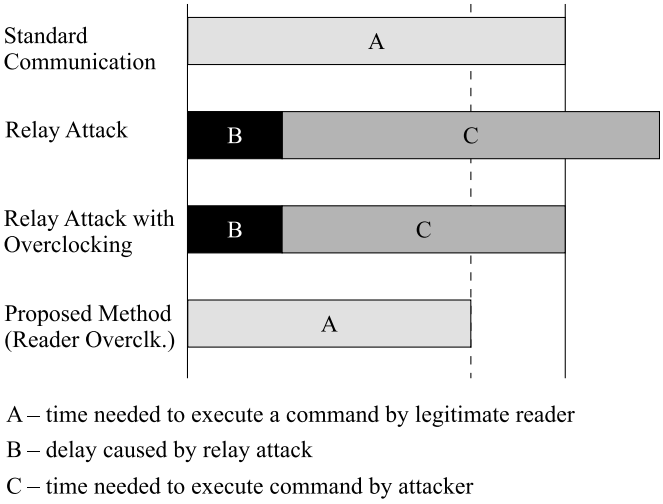

Figure 7.1: Time consumption

not clearly visible in the signal, because it is modulated on a subcarrier 848 kHz, so all parts of the communication are marked in the graph.

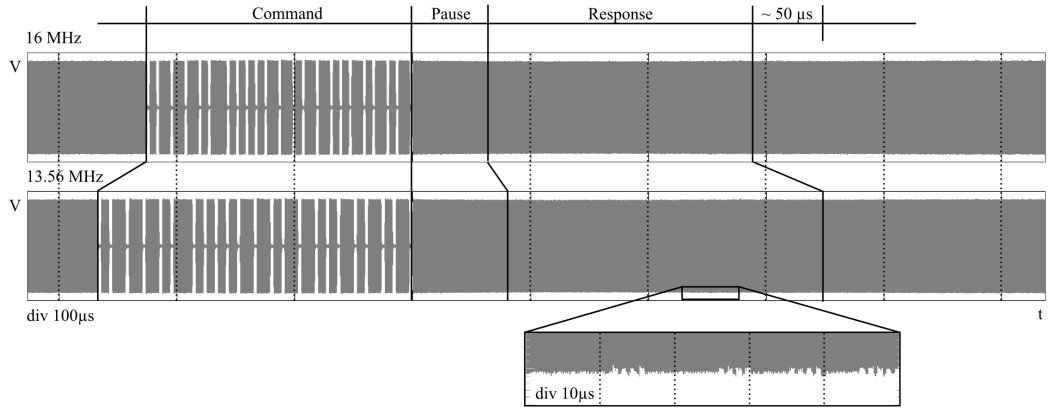

Figure 7.2: Response times comparison

Chapter 8

Conclusions

This thesis analyses contactless smart card protocol threats and presents a method of semi-automated vulnerability finding in contactless smart card protocols using model checking. The high level goal of this thesis was to investigate security of contactless smart card protocols and to find methods of improving security of these protocols. The contribution of this thesis is twofold: 1) the method of semi-automated vulnerability finding using formal methods, which can be used for finding high level attacks on the protocol level, and 2) the countermeasures to relay attacks performed over a network, which were created after relay attacks investigation.

The focus in this thesis is on the high level attacks on the protocol level. Possibility of these attacks was analysed and a method of semi-automated vulnerability finding using formal methods was proposed. The formal model can be created from the protocol definition or extracted from the eavesdropped communication. Unwanted states that pose an attacks are specified. After analysing the protocol and creating the model including the attack states, model checking can be used to automatically find vulnerabilities.

AVANTSSAR platform is used for the formal verification, the models are written in the ASLan++ language. Examples demonstrate the usability of the proposed method.

This thesis deals mainly with simple smart cards with fixed file structure and pre-defined set of commands. These smart cards provide authentication based on symmetric keys, multiple applications and file system with access permissions. Access control is based on keys that are used for authentication, data may be encrypted using some symmetric cipher. One of the most popular and widespread contactless smart cards that uses this scheme is Mifare DESFire, which was used in examples in this thesis. Other smart cards have more sophisticated operating system and can execute

applications on their chip, such as Java Cards, MULTOS cards or BasicCards. Their application logic can be modeled as well, but this thesis is focused mainly on smart cards with fixed file structure and pre-defined set of commands.

The method presented in this thesis was used to find a previously unpublished weakness of the Mifare DESFire MF3ICD40 contactless smart card. Some features of the Mifare DESFire MF3ICD40 were found to be very dangerous and it may be very difficult to implement protocol using this card in a secure way. Although these features are not considered vulnerabilities of the smart card itself, they help to introduce vulnerabilities into the implementation.

We have shown how the inappropriate protocol implementation can yield new vulnerabilities even if the protocol itself is secure and the communication with the hardware is considered secure too. We have demonstrated a sample attack on fictional payment protocol implementation on Mifare DESFire smart card. There is a potential for adversaries to perform similar attacks on real systems. We have introduced a concept of automated vulnerability search using formal verification methods to find complex attack traces which are not likely to be found manually. There is a possibility to use the source code to get an overall image of the protocol and to create the model which is as close to reality as possible, or a man-in-the-middle attack can be used to get information about the protocol from the implementation.

Not all kinds of attacks are covered by the proposed method, so one type of the remaining attack types – the relay attack – was investigated separately. A minor part of this thesis was dedicated to relay attack investigation and countermeasure proposal.

We have proposed a method based on passive detection to prevent real attacks that induce delays significantly longer than the delay caused by the time travelling longer distance. It can be used against most likely attacks, which are not expensive and can be easily performed by attackers with moderate skills, which makes them very dangerous. This countermeasure is quite easy to implement compared to distance bounding protocols. It can be worth implementing such countermeasure even if it does not protect against all theoretical attacks.

We have shown a possible countermeasure to the overclocking attacks. The method is based on overclocking the legitimate reader to the maximal limit where the communicating card can still reliably operate. This method reduces to minimum the chances of the attacker to gain time by overclocking the communication with the legitimate card and hence to circumvent the time limit. We have implemented the reader that communicates with a smart card on the frequency 16 MHz and tested it

with a real card.

Further research may be focused on finding more automatic methods of creating formal model from the analysed protocol. Learning techniques allow automatic inference of behaviour of a system as a finite state machine and can be used to extract such formal models from software on smart cards or to extract the protocol. Such automated reverse-engineering takes little effort and is fast. The finite state machine models obtained can be used in the method presented in this thesis. This approach would improve this method by making it more automatic.

The results presented in this thesis were published in journal [125] with impact factor and international conferences [126], [117], and [127].

Bibliography

[1] EMV. Emv contactless specifications for payment systems. contactless communication protocol. version 2.6. [Online] Cited 2016-04-12. Available at: http://www.emvco.com.

[2] Mastercard contactless. [Online] Cited 2016-04-12. Available at: http://www.mastercard.com/contactless/index.html.

[3] Visa. contactless payments. [Online] Cited 2016-04-12. Available at: https://www.visaeurope.com/receiving-payments/contactless.

[4] American express contactless payments. [Online] Cited 2016-04-12. Available at: https://network.americanexpress.com/en/globalnetwork/contactless/.

[5] Czech Railways. Inkarta. [Online] Cited 2016-04-12. Available at: http://www.inkarta.cz/eng-co-je-inkarta.aspx.

[6] State government victoria. myki. [Online] Cited 2016-04-12. Available at: http://ptv.vic.gov.au/tickets/myki/.

[7] Transport for london. oyster card. [Online] Cited 2016-04-12. Available at: https://oyster.tfl.gov.uk.

[8] Octopus cards limited, hong kong. [Online] Cited 2016-04-12. Available at: http://www.octopus.com.hk/.

[9] Nfc world. loyalty. [Online] Cited 2016-04-12. Available at: http://www.nfcworld.com/technology/loyalty/.

[10] Iso/iec 15693. identification cards – contactless integrated circuit cards – vicinity cards. [Online] Cited 2016-04-12. Available at: http://www.iso.org/.

[11] HID Global. Access control solutions. [Online] Cited 2016-04-12. Available at: http://www.hidglobal.com/products/Cards-and-Credentials/iCLASS.

[12] International Civil Aviation Organization (ICAO). Document 9303 Machine Readable Travel Documents (MRTD). Part 1: Machine Readable Passports, 2005.

[13] ISO/IEC 7501. Identification Cards - Machine Readable Travel Documents, October 2005. [Online] Cited 2016-04-12. Available at: http://www.iso.org/.

[14] Mykad - malaysian identity card. [Online] Cited 2016-04-12. Available at: http://www.jpn.gov.my/.

[15] NFC trials, pilots, tests and live services around the world, 2010. [Online] Cited 2016-04-12. Available at: http://www.nfcworld.com/.

[16] Dominique Paret. *RFID and Contactless Smart Card Applications*. John Wiley & Sons, 2005.

[17] C. Mulliner. Vulnerability analysis and attacks on nfc-enabled mobile phones. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 695 –700, march 2009.

[18] ISO/IEC 7816. Identification cards – integrated circuit(s) cards with contacts, 2004. [Online] Cited 2016-04-12. Available at: http://www.iso.org/.

[19] ISO/IEC 14443. Identification cards - contactless integrated circuit cards - proximity cards, 2010. [Online] Cited 2016-04-12. Available at: http://www.iso.org/.

[20] Smart card basics. types of smart card. [Online] Cited 2016-04-12. Available at: http://www.smartcardbasics.com/smart-card-types.html.

[21] NXP Semiconductors. Mifare classic 4k - mainstream contactless smart card ic for fast and easy solution development (product data sheet). [Online] Cited 2016-04-12. Available at: http://www.nxp.com/documents/data_sheet/MF1S70YYX.pdf.

[22] Karsten Nohl, David Evans, Starbug Starbug, and Henryk Plötz. Reverse-engineering a cryptographic rfid tag. In *Proceedings of the 17th conference on Security symposium*, pages 185–193, Berkeley, CA, USA, 2008. USENIX Association.

[23] Flavio D. Garcia, Gerhard Koning Gans, Ruben Muijrers, Peter Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling mifare classic. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 97–114, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Nicolas T. Courtois, Karsten Nohl, and Sean O'Neil. Algebraic attacks on the crypto-1 stream cipher in mifare classic and oyster cards. Cryptology ePrint Archive, Report 2008/166, 2008. `http://eprint.iacr.org/`.

[25] Flavio D. Garcia, Peter van Rossum, Roel Verdult, and Ronny Wichers Schreur. Wirelessly pickpocketing a mifare classic card. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 3–15, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Gerhard Koning Gans, Jaap-Henk Hoepman, and Flavio D. Garcia. A practical attack on the mifare classic. In *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS '08, pages 267–282, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] NXP Semiconductors. Mifare desfire mf3icd40 contactless multi-application ic (short form specification) data sheet. [Online] Cited 2016-04-12. Available at: `http://www.nxp.com/documents/short_data_sheet/075532.pdf`.

[28] NXP Semiconductors. Mifare desfire ev1 contactless multi-application ic, product short data sheet. [Online] Cited 2016-04-12. Available at: `http://www.nxp.com/documents/short_data_sheet/MF3ICDX21_41_81_SDS.pdf`.

[29] NXP Semiconductors. Mifare desfire ev2 contactless multi-application ic (preliminary short data sheet). [Online] Cited 2016-04-12. Available at: `http://www.nxp.com/documents/short_data_sheet/MF3DX2_MF3DHX2_SDS.pdf`.

[30] Oracle. Java card technology. [Online] Cited 2016-04-12. Available at: `http://www.oracle.com/technetwork/java/embedded/javacard`.

[31] MULTOS. Multos technology). [Online] Cited 2016-04-12. Available at: `https://www.multos.com/technology/`.

[32] ZeitControl. Basiccard. [Online] Cited 2016-04-12. Available at: `http://www.basiccard.com/`.

[33] Konstantinos Markantonakis, Keith Mayes, Damien Sauveron, and Ioannis G Askoxylakis. Overview of security threats for smart cards in the public transport industry. In *IEEE International Conference on e-Business Engineering*, pages 506–513. IEEE, 2008.

[34] Martin Henzl. Security of contactless smart cards. In *Proceedings of the 17th Conference STUDENT EEICT 2011*, pages 585–589, Brno, CZ, 2011. VUT v Brne.

[35] Smart card alliance. what makes a smart card secure?, smart card alliance payments council white paper, October 2008.

[36] Helena Handshuh. Contactless technology security issues. Information Security Bulletin, 2004.

[37] John Horton Conway. *On Numbers and Games*. London Mathematical Society Monographs. Academic Press, London, 1976.

[38] Stefan Brands and David Chaum. Distance-bounding protocols. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 344–359. Springer Berlin Heidelberg, 1994.

[39] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Softw. Eng.*, 22(1):6–15, January 1996.

[40] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[41] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information processing letters*, 56(3):131–133, 1995.

[42] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer, 1996.

[43] Danny Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[44] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[45] Andrew C Yao. Theory and application of trapdoor functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 80–91. IEEE, 1982.

[46] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *Journal of Automated Reasoning*, 46(3-4):225–259, 2011.

[47] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26(1):99–123, 2014.

[48] Hyun-Seok Kim, Jun-Hyun Oh, Ju-Bae Kim, Yeon-Oh Jeong, and Jin-Young Choi. Formal verification of cryptographic protocol for secure rfid system. In *Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management - Volume 02*, pages 470–477, Washington, DC, USA, 2008. IEEE Computer Society.

[49] Qingfeng Chen, Chengqi Zhang, and Shichao Zhang. *Secure transaction protocol analysis: models and applications*. Springer-Verlag, Berlin, Heidelberg, 2008.

[50] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, January 1965.

[51] Catherine Meadows. The nrl protocol analyzer: An overview. *The Journal of Logic Programming*, 26(2):113–131, 1996.

[52] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[53] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8:18–36, February 1990.

[54] Martín Abadi and Mark R. Tuttle. A semantics for a logic of authentication (extended abstract). In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, PODC '91, pages 201–216, New York, NY, USA, 1991. ACM.

[55] AVANTSSAR. Automated validation of trust and security of service-oriented architectures. [Online] Cited 2016-04-12. Available at: `http://www.avantssar.eu/`.

[56] Jonathan K Millen, Sidney C Clark, and Sheryl B Freedman. The interrogator: Protocol secuity analysis. *Software Engineering, IEEE Transactions on*, (2):274–288, 1987.

[57] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers & Security*, 11(1):75 – 89, 1992.

[58] Gavin Lowe. Casper: a compiler for the analysis of security protocols. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 18–30, Jun 1997.

[59] John C Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using murφ. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151. IEEE, 1997.

[60] Edmund M Clarke, Somesh Jha, and Will Marrero. Verifying security protocols with brutus. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):443–487, 2000.

[61] Catherine Meadows. Analysis of the internet key exchange protocol using the nrl protocol analyzer. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 216–231. IEEE, 1999.

[62] Catherine Meadows, Paul Syverson, and Iliano Cervesato. Formal specification and analysis of the group domain of interpretation using npatr and the nrl protocol analyzer. 2004.

[63] Catherine Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.

[64] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for tls. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 459–468. ACM, 2008.

[65] Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Nikhil Swamy. Verified implementations of the information card federated

identity-management protocol. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 123–135. ACM, 2008.

[66] David Basin and Cas Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *Computer Security–ESORICS 2010*, pages 340–356. Springer, 2010.

[67] David Basin, Cas Cremers, and Simon Meier. *Provably repairing the ISO/IEC 9798 standard for entity authentication.* Springer, 2012.

[68] Cas Cremers. *Key exchange in IPsec revisited: Formal analysis of IKEv1 and IKEv2.* Springer, 2011.

[69] David Basin, Cas Cremers, and Catherine Meadows. Model checking security protocols. *Handbook of Model Checking*, 2011.

[70] Richard A DeMillo, Nancy A Lynch, and Michael J Merritt. Cryptographic protocols. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 383–400. ACM, 1982.

[71] Auguste Kerckhoffs. La cryptographie militaire. Journal des sciences militaires, 1883.

[72] W. Teepe and K. Nohl. Making the best of mifare classic, 2008. [Online] Cited 2016-04-12. Available at: `www.cs.ru.nl/~wouter/papers/2008-thebest-updated.pdf`.

[73] David Oswald and Christof Paar. Breaking mifare desfire mf3icd40: power analysis and templates in the real world. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 207–222. Springer, 2011.

[74] Timo Kasper, David Oswald, and Christof Paar. Side-channel analysis of cryptographic rfids with analog demodulation. In *RFID. Security and Privacy*, pages 61–77. Springer, 2012.

[75] Michael Hutter, Stefan Mangard, and Martin Feldhofer. *Power and EM Attacks on Passive 13.56 MHz RFID Devices.* Springer, 2007.

[76] Thomas Plos, Michael Hutter, and Martin Feldhofer. Evaluation of side-channel preprocessing techniques on cryptographic-enabled hf and uhf rfid-tag prototypes. In *Workshop on RFID Security*, pages 114–127, 2008.

[77] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the keeloq code hopping scheme. In *Advances in Cryptology–CRYPTO 2008*, pages 203–220. Springer, 2008.

[78] Timo Kasper, David Oswald, and Christof Paar. Em side-channel attacks on commercial contactless smartcards using low-cost equipment. In *Information Security Applications*, pages 79–93. Springer, 2009.

[79] Timo Kasper, David Oswald, and Christof Paar. Side-channel analysis of cryptographic rfids with analog demodulation. In *RFID. Security and Privacy*, pages 61–77. Springer, 2012.

[80] Ross Anderson. Why cryptosystems fail. In *Proceedings of the 1st ACM conference on Computer and communications security*, CCS '93, pages 215–227, New York, NY, USA, 1993. ACM.

[81] Ross J. Anderson and Markus G. Kuhn. Low cost attacks on tamper resistant devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, London, UK, 1998. Springer-Verlag.

[82] Paul Youn, Ben Adida, Mike Bond, Jolyon Clulow, Jonathan Herzog, Amerson Lin, Ronald L. Rivest, and Ross Anderson. Robbing the bank with a theorem prover. Technical Report UCAM-CL-TR-644, University of Cambridge, Computer Laboratory, aug 2005.

[83] Ross J. Anderson. The correctness of crypto transaction sets. In *Revised Papers from the 8th International Workshop on Security Protocols*, pages 125–127, London, UK, 2001. Springer-Verlag.

[84] Mike Bond, , Mike Bond, and Piotr Zielinski. Decimalisation table attacks for pin cracking. Technical report, 2003.

[85] Mike Bond. *Understanding Security APIs*. PhD thesis, University of Cambridge, Jan 2004.

[86] Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing pkcs# 11 security tokens. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 260–269. ACM, 2010.

[87] N. Moebius, K. Stenzel, and W. Reif. Pitfalls in formal reasoning about security protocols. In *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, pages 248–253, Feb 2010.

[88] Steven J. Murdoch, Saar Drimer, Ross Anderson, and Mike Bond. Chip and pin is broken. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 433–446, Washington, DC, USA, 2010. IEEE Computer Society.

[89] Haixing Yan, Huixing Fang, Christian Kuka, and Huibiao Zhu. Verification for oauth using aslan++. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 76–84. IEEE, 2015.

[90] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal models of bank cards for free. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 461–468. IEEE, 2013.

[91] David L Dill, Andreas J Drexler, Alan J Hu, and C Han Yang. Protocol verification as a hardware design aid. In *ICCD*, volume 92, pages 522–525. Citeseer, 1992.

[92] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[93] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[94] John C Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using murφ. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151. IEEE, 1997.

[95] Josang Audun. Security protocol verification using spin. In *Proceedings of the First SPIN Workshop, INRS-Telecommunications, Montreal, Canada*, 1995.

[96] Christian Haack and Alan Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In *CONCUR 2005–Concurrency Theory*, pages 202–216. Springer, 2005.

[97] Casimier Joseph Franciscus Cremers. *Scyther: Semantics and verification of security protocols*. Eindhoven University of Technology, 2006.

[98] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, H Riis Nielson, and H Riis Nielson. Automatic validation of protocol narration. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 126–140. IEEE, 2003.

[99] Stephen Gilmore, Valentin Haenel, Leïla Kloul, and Monika Maidl. Choreographing security and performance analysis for web services. In *Formal Techniques for Computer Systems and Business Processes*, pages 200–214. Springer, 2005.

[100] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. In *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, pages 331–340. IEEE, 2005.

[101] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, P Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, et al. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification*, pages 281–285. Springer, 2005.

[102] Luca Viganò. Automated security protocol analysis with the avispa tool. *Electronic Notes in Theoretical Computer Science*, 155:61–86, 2006.

[103] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *csfw*, page 0082. IEEE, 2001.

[104] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A high level protocol specification language for industrial security-sensitive protocols. In *AUSTRIAN COMPUTER SOCIETY*, pages 193–205, 2004.

[105] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[106] David von Oheimb and Sebastian Mödersheim. Aslan++ – a formal security specification language for distributed systems. In BernhardK. Aichernig, FrankS. de Boer, and MarcelloM. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2012.

[107] David Basin, Sebastian Mödersheim, and Luca Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, 2004.

[108] Sebastian Mödersheim, Luca Vigano, and David Basin. Constraint differentiation: Search-space reduction for the constraint-based analysis of security protocols. *Journal of Computer Security*, 18(4):575–618, 2010.

[109] David Basin, Sebastian Mödersheim, and Luca Vigano. Algebraic intruder deductions. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 549–564. Springer, 2005.

[110] Mathieu Turuani. The cl-atse protocol analyser. In *Term Rewriting and Applications - Proc. of RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286, Seattle, WA, USA, 2006.

[111] Alessandro Armando, Roberto Carbone, and Luca Compagna. Satmc: a sat-based model checker for security-critical systems. In *TACAS'14: Proceedings of the 20th international Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45. Springer, 2014.

[112] SATMC. A sat-based model-checker for security protocols and security-sensitive applications. [Online] Cited 2016-04-12. Available at: `http://www.ai-lab.it/satmc/`.

[113] Catherine A. Meadows. *Computer Security — ESORICS 96: 4th European Symposium on Research in Computer Security Rome, Italy, September 25–27, 1996 Proceedings*, chapter Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches, pages 351–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[114] Cas J.F. Cremers, Pascal Lafourcade, and Philippe Nadeau. Comparing state spaces in automatic protocol analysis. In *Formal to Practical Security*, volume 5458/2009 of *Lecture Notes in Computer Science*, pages 70–94. Springer Berlin/Heidelberg, 2009.

[115] Pascal Lafourcade, Vanessa Terrade, and Sylvain Vigier. Comparison of cryptographic verification tools dealing with algebraic properties. In *Formal Aspects in Security and Trust*, pages 173–185. Springer, 2009.

[116] A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 385–396, July 2007.

[117] M. Henzl and P. Hanacek. Modeling of contactless smart card protocols and automated vulnerability finding. In *Biometrics and Security Technologies (ISBAST), 2013 International Symposium on*, pages 141–148, July 2013.

[118] libnfc. Public platform independent near field communication (nfc) library, 2010. [Online] Cited 2016-04-12. Available at: `http://nfc-tools.org/`.

[119] Lishoy Francis, Gerhard Hancke, Keith Mayes, and Konstantinos Markantonakis. Practical nfc peer-to-peer relay attack using mobile phones. In SiddikaBerna Ors Yalcin, editor, *Radio Frequency Identification: Security and Privacy Issues*, volume 6370 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin Heidelberg, 2010.

[120] Lishoy Francis, Gerhard Hancke, Keith Mayes, and Konstantinos Markantonakis. Practical relay attack on contactless transactions by using nfc mobile phones. Cryptology ePrint Archive, Report 2011/618, 2011.

[121] Gerhard P Hancke. A practical relay attack on iso 14443 proximity cards. *Technical report, University of Cambridge Computer Laboratory*, 59:382–385, 2005.

[122] Proxmark. A radio frequency identification tool, 2010. [Online] Cited 2016-04-12. Available at: `www.proxmark.org`.

[123] OpenPCD. Openpcd passive rfid project. [Online] Cited 2016-04-12. Available at: `www.openpcd.org`.

[124] Gerhard P. Hancke and Markus G. Kuhn. Attacks on time-of-flight distance bounding channels. In *Proceedings of the First ACM Conference on Wireless Network Security*, WiSec '08, pages 194–202, New York, NY, USA, 2008. ACM.

[125] M. Henzl and P. Hanacek. A security formal verification method for protocols using cryptographic contactless smart cards. *Radioengineering*, 25(1):132–139, April 2016.

[126] M. Henzl, P. Hanacek, P. Jurnecka, and M. Kacic. A concept of automated vulnerability search in contactless communication applications. In *Security Technology (ICCST), 2012 IEEE International Carnahan Conference on*, pages 180–186, Oct 2012.

[127] M. Henzl, P. Hanacek, and M. Kacic. Preventing real-world relay attacks on contactless devices. In *Security Technology (ICCST), 2014 International Carnahan Conference on*, pages 1–6, Oct 2014.

Appendices

List of Appendices

Appendix A

ASLan++ Source of Example 1

```
specification example
channel_model CCM

entity Environment {

  types
  command < text;
  dataAddress < text;
  encrypted < text;
  authenticate < text;

  symbols
  pcd,picc: agent;
  readFile(dataAddress, authenticate): command;
  writeFile(dataAddress, authenticate, encrypted): command;
  noninvertible enc(symmetric_key, message): encrypted;
  noninvertible dec(symmetric_key, message): encrypted;
  noninvertible auth(symmetric_key, symmetric_key): authenticate;
  corrupted: text;
  addressName: dataAddress;
  addressBalance: dataAddress;
  addressNameCorrupted: dataAddress;
  addressBalanceCorrupted: dataAddress;
  nonpublic key1: symmetric_key;
  nonpublic key2: symmetric_key;
  nonpublic none: symmetric_key;
  nonpublic oldBalance: text;
  nonpublic newBalance: text;
  nonpublic name: text;
  ok: text;

  entity Session (A, B: agent) {

  symbols
  fileSystem(dataAddress, dataAddress, symmetric_key, symmetric_key,
        message): fact;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PCD
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    entity PCD (Actor, B: agent) {

      symbols
      Data: message;
      SessionKey: symmetric_key;
      DataAddress: dataAddress;

      body {

        % fresh session key generation
        SessionKey := fresh();

        % read name
        Actor -> B: readFile(addressName, auth(key1, SessionKey));
        B -> Actor: enc(SessionKey, ?Data);

% countermeasure 1:
%%      select {
%%        on(Data = name): {
% end of countermeasure 1

            % PCD should have read "name"
            %% assert name: Data = name;

            % read balance
            Actor -> B: readFile(addressBalance, auth(key1, SessionKey));
            B -> Actor: enc(SessionKey, ?Data);

% countermeasure 1:
%%          select {
%%           on(Data = oldBalance): {
% end of countermeasure 1

                % PCD should have read "oldBalance"
                assert oldBalance: Data = oldBalance;

                % 1. check whether there is enough money
                % 2. subtract the value of the goods
                % 3. write new balance
                Actor -> B: writeFile(addressBalance, auth(key1,
                        SessionKey),  dec(SessionKey, newBalance));
                B -> Actor: ok;

% countermeasure 2:
%%              % check written balance
%%              Actor -> B: readFile(addressBalance, auth(key1, SessionKey));
%%              B -> Actor: enc(SessionKey, ?Data);
%%
%%              % check whether new data were written
%%              select {
%%                on(Data = newBalance): {
% end of countermeasure 2
```

```
                % PICC should have "newBalance" at address "addressBalance"
                assert trueBalance: fileSystem(addressBalance, ?, key1,
                      key1, newBalance);

                % end of protocol

% countermeasure 2:
%%                }
%%              }
% end of countermeasure 2
% countermeasure 1:
%%            }
%%          }
%%        }
%%      }
% end of countermeasure 1
      }
    }

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PICC
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    entity PICC (A, Actor: agent) {

      symbols
      Data: message;
      OldData: message;
      SessionKey: symmetric_key;
      SessionKeyTemp: symmetric_key;
      KeyRead: symmetric_key;
      AuthenticatedKey: symmetric_key;
      Command: command;
      DataAddress, DataAddressCorrupted: dataAddress;
      Encrypted: encrypted;
      UsedSessionKeys: symmetric_key set;

      body {

        % initialization
        AuthenticatedKey := none;
        SessionKey := none;

        % main loop
        while(true) {

          % read command
          A -> Actor: ?Command;

          select {

            %%%%%%%%%%%%%
            %% readFile %%
            %%%%%%%%%%%%%
            on(Command = readFile(?DataAddress, auth(?AuthenticatedKey,
```

```
        ?SessionKeyTemp))): {

  % authentication
  select {
    on(!UsedSessionKeys->contains(SessionKeyTemp) |
            SessionKey = SessionKeyTemp): {
      UsedSessionKeys->add(SessionKeyTemp);
      SessionKey := SessionKeyTemp;

      % read file
      select {
        on(fileSystem(DataAddress, ?, AuthenticatedKey,
                ?, ?Data)): {

          % send response
          Actor -> A: enc(SessionKey, Data);
        }
      }
    }
  }
}

%%%%%%%%%%%%%%
%% writeFile %%
%%%%%%%%%%%%%%
on(Command = writeFile(?DataAddress, auth(?AuthenticatedKey,
        ?SessionKeyTemp), ?Encrypted)): {

  % authentication
  select {
    on((!UsedSessionKeys->contains(SessionKeyTemp) |
            SessionKey = SessionKeyTemp) &
            Encrypted = dec(SessionKeyTemp, ?Data)): {
      UsedSessionKeys->add(SessionKeyTemp);
      SessionKey := SessionKeyTemp;

      % write file
      select {
        on(fileSystem(DataAddress, ?DataAddressCorrupted,
                ?KeyRead, AuthenticatedKey, ?OldData)): {
          retract(fileSystem(DataAddress, DataAddressCorrupted,
                KeyRead, AuthenticatedKey, OldData));
          fileSystem(DataAddress, DataAddressCorrupted, KeyRead,
                AuthenticatedKey, Data);

          % corrupt the rest of the same file
          select {
            on(fileSystem(DataAddressCorrupted, DataAddress,
                    ?KeyRead, ?AuthenticatedKey, ?OldData)): {
              retract(fileSystem(DataAddressCorrupted,
                    DataAddress, KeyRead, AuthenticatedKey,
                    OldData));
              fileSystem(DataAddressCorrupted, DataAddress,
                    KeyRead, AuthenticatedKey, corrupted);
```

```
                        }
                      }
                    }
                  }

                  % send response
                  Actor -> A: ok;
                }
              }
            }
          }
        }
      }
    }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Session
    body {

      % all possible data locations
      fileSystem(addressName, addressNameCorrupted, key1, key1, name);
      fileSystem(addressBalance, addressBalanceCorrupted, key1, key1,
            oldBalance);
      fileSystem(addressNameCorrupted, addressName, key1, key1, corrupted);
      fileSystem(addressBalanceCorrupted, addressBalance, key1, key1,
            corrupted);

      % new roles
      new PCD(A, B);
      new PICC(A, B);
    }
  }

  % Environment
  body {

    % new session
    new Session(pcd, picc);
  }
}
```

Appendix B

ASLan++ Source of Example 2

```
specification example
channel_model CCM

entity Environment {

  types
  encCommand < text;
  command < text;
  dataAddress < text;
  encrypted < text;
  authenticate < text;

  symbols
  pcd,picc: agent;
  encryptedCommand(authenticate,encrypted): encCommand;
  falseBalance: text;
  readFile(dataAddress): command;
  writeFile(dataAddress, message): command;
  noninvertible enc(symmetric_key, message): encrypted;
  noninvertible auth(symmetric_key, symmetric_key): authenticate;
  corrupted: text;
  addressName: dataAddress;
  addressBalance: dataAddress;
  addressNameCorrupted: dataAddress;
  addressBalanceCorrupted: dataAddress;
  nonpublic key1: symmetric_key;
  key2: symmetric_key;
  nonpublic none: symmetric_key;
  nonpublic oldBalance: text;
  nonpublic newBalance: text;
  nonpublic name: text;
  ok: text;

  entity Session (A, B: agent) {

  symbols
  fileSystem(dataAddress, dataAddress, symmetric_key, symmetric_key,
        message): fact;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PCD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    entity PCD (Actor, B: agent) {

      symbols
      Data: message;
      SessionKey: symmetric_key;
      DataAddress: dataAddress;

      body {

        % fresh session key generation
        SessionKey := fresh();

        % read name
        Actor -> B: encryptedCommand(auth(key1, SessionKey), enc(SessionKey,
                readFile(addressName)));
        B -> Actor: enc(SessionKey, ?Data);

        % PICC should not have "falseBalance" at address "addressBalance"
        assert trueBalance: !fileSystem(addressBalance, ?, key1, key1,
                falseBalance);

        % read balance
        Actor -> B: encryptedCommand(auth(key1, SessionKey), enc(SessionKey,
                readFile(addressBalance)));
        B -> Actor: enc(SessionKey, ?Data);

        % PICC should not have "falseBalance" at address "addressBalance"
        assert trueBalance: !fileSystem(addressBalance, ?, key1, key1,
                falseBalance);

        % 1. check whether there is enough money
        % 2. subtract the value of the goods
        % 3. write new balance
        Actor -> B: encryptedCommand(auth(key1, SessionKey), enc(SessionKey,
                writeFile(addressBalance, newBalance)));
        B -> Actor: enc(SessionKey, ok);

        % PICC should not have "falseBalance" at address "addressBalance"
        assert trueBalance: !fileSystem(addressBalance, ?, key1, key1,
                falseBalance);

        % check written balance
        Actor -> B: encryptedCommand(auth(key1, SessionKey), enc(SessionKey,
                readFile(addressBalance)));
        B -> Actor: enc(SessionKey, ?Data);

        % PICC should not have "falseBalance" at address "addressBalance"
        assert trueBalance: !fileSystem(addressBalance, ?, key1, key1,
                falseBalance);

        % check whether new data were written
```

```
        select {
          on(Data = newBalance): {

            % PICC should have "newBalance" at address "addressBalance"
            assert trueBalance: fileSystem(addressBalance, ?, key1, key1,
                  newBalance);

            % end of protocol
          }
        }
      }
    }

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PICC
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    entity PICC (A, Actor: agent) {

      symbols
      Data: message;
      OldData: message;
      SessionKey: symmetric_key;
      SessionKeyTemp: symmetric_key;
      KeyRead: symmetric_key;
      AuthenticatedKey: symmetric_key;
      EncryptedCommand: encCommand;
      Command: command;
      DataAddress, DataAddressCorrupted: dataAddress;
      Encrypted: encrypted;
      UsedSessionKeys: symmetric_key set;

      body {

        % initialization
        AuthenticatedKey := none;
        SessionKey := none;

        % main loop
        while(true) {

          % read command
          A -> Actor: ?EncryptedCommand;

          select {
            on(EncryptedCommand = encryptedCommand(auth(?AuthenticatedKey,
                    ?SessionKeyTemp), ?Encrypted)): {
              select {

                %%%%%%%%%%%%%%
                %% readFile %%
                %%%%%%%%%%%%%%
                % authentication
                on((!UsedSessionKeys->contains(SessionKeyTemp) |
                        SessionKey = SessionKeyTemp) & Encrypted =
```

```
                    enc(SessionKeyTemp, readFile(?DataAddress))): {
                UsedSessionKeys->add(SessionKeyTemp);
                SessionKey := SessionKeyTemp;

                % read file
                select {
                  on(fileSystem(DataAddress,?,AuthenticatedKey,?,?Data)): {
                    Actor -> A: enc(SessionKey, Data);
                  }
                }
              }

              %%%%%%%%%%%%%%%
              %% writeFile %%
              %%%%%%%%%%%%%%%
              % authentication
              on((!UsedSessionKeys->contains(SessionKey) | SessionKey =
                    SessionKeyTemp) & Encrypted = enc(SessionKeyTemp,
                    writeFile(?DataAddress, ?Data))): {
                UsedSessionKeys->add(SessionKeyTemp);
                SessionKey := SessionKeyTemp;

                % write file
                select {
                  on(fileSystem(DataAddress, ?DataAddressCorrupted,
                          ?KeyRead, AuthenticatedKey, ?OldData)): {
                    retract(fileSystem(DataAddress, DataAddressCorrupted,
                          KeyRead, AuthenticatedKey, OldData));
                    fileSystem(DataAddress, DataAddressCorrupted, KeyRead,
                          AuthenticatedKey, Data);

                    % corrupt the rest of the same file
                    select {
                      on(fileSystem(DataAddressCorrupted, DataAddress,
                              ?KeyRead, ?AuthenticatedKey, ?OldData)): {
                        retract(fileSystem(DataAddressCorrupted,
                              DataAddress, KeyRead, AuthenticatedKey,
                              OldData));
                        fileSystem(DataAddressCorrupted, DataAddress,
                              KeyRead, AuthenticatedKey, corrupted);
                      }
                    }
                  }
                }

                % send response
                Actor -> A: enc(SessionKey, 0);
              }
            }
          }
        }
      }
    }
  }
}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Session
    body {

      % all possible data locations
      fileSystem(addressName, addressNameCorrupted, key1, key2, name);
      fileSystem(addressBalance, addressBalanceCorrupted, key1, key1,
            oldBalance);
      fileSystem(addressNameCorrupted, addressName, key1, key2, corrupted);
      fileSystem(addressBalanceCorrupted, addressBalance, key1, key1,
            corrupted);

      % new roles
      new PCD(A,B);
      new PICC(A,B);
    }
  }

  % Environment
  body {

    % new session
    new Session(pcd,picc);
  }
}
```

Appendix C

ASLan++ Source of Example 3

```
specification example
channel_model CCM

entity Environment {

  types
  encCommand < text;
  command < text;
  dataAddress < text;
  encrypted < text;
  authenticate < text;
  initializationVector < text;

  symbols
  pcd,picc: agent;
  encryptedCommand(authenticate,encrypted): encCommand;
  readFile(dataAddress): command;
  writeFile(dataAddress, message): command;
  noninvertible enc(symmetric_key, initializationVector, message): encrypted;
  noninvertible auth(symmetric_key, symmetric_key): authenticate;
  nextIV(initializationVector): initializationVector;
  zeroIV: initializationVector;
  corrupted: text;
  addressName: dataAddress;
  addressBalance: dataAddress;
  addressNameCorrupted: dataAddress;
  addressBalanceCorrupted: dataAddress;
  nonpublic key1: symmetric_key;
  key2: symmetric_key;
  nonpublic none: symmetric_key;
  nonpublic oldBalance: text;
  nonpublic newBalance: text;
  nonpublic name: text;
  ok: text;

  entity Session (A, B: agent) {

  symbols
  fileSystem(dataAddress, dataAddress, symmetric_key, symmetric_key,
```

```
        message): fact;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PCD
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    entity PCD (Actor, B: agent) {

      symbols
      Data: message;
      SessionKey: symmetric_key;
      DataAddress: dataAddress;
      IV: initializationVector;

      body {

        % fresh session key generation
        SessionKey := fresh();

        % read balance
        Actor -> B: encryptedCommand(auth(key1, SessionKey), enc(SessionKey,
                nextIV(zeroIV), readFile(addressBalance)));
        B -> Actor: enc(SessionKey, next(nextIV(zeroIV)), ?Data);

        % 1. check whether there is enough money
        % 2. subtract the value of the goods
        % 3. write new balance
        Actor -> B: encryptedCommand(auth(key1, SessionKey), enc(SessionKey,
                next(nextIV(nextIV(zeroIV))),
                writeFile(addressBalance, newBalance)));

% refinement 1:
%%      B -> Actor: ?;
% OR refinement 2:
%%      B -> Actor: enc(?, ?, ok);
% OR refinement 3:
%%      B -> Actor: enc(SessionKey, next(nextIV(nextIV(nextIV(zeroIV)))),
%%              ok);
% end of refinement

        % PICC should have "newBalance" at address "addressBalance"
        assert trueBalance: fileSystem(addressBalance, ?, key1, key1,
                newBalance);

        % end of protocol
      }
    }

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PICC
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    entity PICC (A, Actor: agent) {

      symbols
      Data: message;
```

```
OldData: message;
SessionKey: symmetric_key;
SessionKeyTemp: symmetric_key;
KeyRead: symmetric_key;
AuthenticatedKey: symmetric_key;
EncryptedCommand: encCommand;
Command: command;
DataAddress, DataAddressCorrupted: dataAddress;
Encrypted: encrypted;
UsedSessionKeys: symmetric_key set;
LastIV: initializationVector;
IV: initializationVector;

body {

  % initialization
  AuthenticatedKey := none;
  SessionKey := none;

  % main loop
  while(true) {

    % read command
    A -> Actor: ?EncryptedCommand;

    select {
      on(EncryptedCommand = encryptedCommand(auth(?AuthenticatedKey,
              ?SessionKeyTemp), ?Encrypted)): {
        select {

          %%%%%%%%%%%%%
          %% readFile %%
          %%%%%%%%%%%%%
          % authentication
          on((!UsedSessionKeys->contains(SessionKeyTemp) & Encrypted =
                  enc(SessionKeyTemp, nextIV(zeroIV),
                  readFile(?DataAddress))) | (SessionKey =
                  SessionKeyTemp & Encrypted = enc(SessionKeyTemp,
                  nextIV(lastIV), readFile(?DataAddress)))): {
            if(SessionKey = SessionKeyTemp): {
              lastIV := next(nextIV(lastIV));
            } else {
              lastIV := next(nextIV(zeroIV));
            }
            UsedSessionKeys->add(SessionKeyTemp);
            SessionKey := SessionKeyTemp;

            % read file
            select {
              on(fileSystem(DataAddress, ?, AuthenticatedKey, ?,
                      ?Data)): {
                Actor -> A: enc(SessionKey, lastIV, Data);
              }
            }
```

```
                }

                %%%%%%%%%%%%%%
                %% writeFile %%
                %%%%%%%%%%%%%%
                % authentication
                on((!UsedSessionKeys->contains(SessionKeyTemp) & Encrypted =
                        enc(SessionKeyTemp, nextIV(zeroIV),
                        writeFile(?DataAddress, ?Data))) | (SessionKey =
                        SessionKeyTemp & Encrypted = enc(SessionKeyTemp,
                        nextIV(lastIV), writeFile(?DataAddress, ?Data)))): {
                  if(SessionKey = SessionKeyTemp): {
                    lastIV := next(nextIV(lastIV));
                  } else {
                    lastIV := next(nextIV(zeroIV));
                  }
                  UsedSessionKeys->add(SessionKeyTemp);
                  SessionKey := SessionKeyTemp;

                  % write file
                  select {
                    on(fileSystem(DataAddress, ?DataAddressCorrupted,
                            ?KeyRead, AuthenticatedKey, ?OldData)): {
                      retract(fileSystem(DataAddress, DataAddressCorrupted,
                            KeyRead, AuthenticatedKey, OldData));
                      fileSystem(DataAddress, DataAddressCorrupted, KeyRead,
                            AuthenticatedKey, Data);

                      % corrupt the rest of the same file
                      select {
                        on(fileSystem(DataAddressCorrupted, DataAddress,
                                ?KeyRead, ?AuthenticatedKey, ?OldData)): {
                          retract(fileSystem(DataAddressCorrupted,
                                DataAddress, KeyRead, AuthenticatedKey,
                                OldData));
                          fileSystem(DataAddressCorrupted, DataAddress,
                                KeyRead, AuthenticatedKey, corrupted);
                        }
                      }
                    }
                  }

                  % send response
                  Actor -> A: enc(SessionKey, lastIV, ok);
                }
              }
            }
          }
        }
      }
    }
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Session
```

```
    body {

      % all possible data locations
      fileSystem(addressName, addressNameCorrupted, key1, key2, name);
      fileSystem(addressBalance, addressBalanceCorrupted, key1, key1,
          oldBalance);
      fileSystem(addressNameCorrupted, addressName, key1, key2, corrupted);
      fileSystem(addressBalanceCorrupted, addressBalance, key1, key1,
            corrupted);

      % new roles
      new PCD(A,B);
      new PICC(A,B);
    }
  }

  % Environment
  body {

    % new session
    new Session(pcd,picc);
  }
}
```