# Lessons learned in the analysis of the EMV and TLS security protocols

**Joeri de Ruiter**

Copyright © Joeri de Ruiter, 2015
ISBN: 978-94-6295-330-7
IPA dissertation series: 2015-11

Typeset using LaTeX

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

This work is part of the research programme Design and Analysis of Secure Distributed Protocols (DASDiP), which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).

# Lessons learned in the analysis of the EMV and TLS security protocols

Proefschrift

ter verkrijging van de graad van doctor aan de

Radboud Universiteit Nijmegen op gezag van
de rector magnificus prof.dr. Th.L.M. Engelen,

volgens besluit van het college van decanen
in het openbaar te verdedigen op donderdag 27 augustus 2015
om 14:30 uur precies

door

Joeri Evert Jan de Ruiter

geboren op 16 april 1986
te Sittard

**Promotor:**

Prof. dr. B.P.F. Jacobs

**Copromotor:**

Dr. ir. E. Poll

**Manuscriptcommissie:**

Dr. K. Bhargavan (Inria Paris - Rocquencourt, Frankrijk)
Prof. dr. W.J. Fokkink (Vrije Universiteit Amsterdam)
Prof. dr. M.D. Ryan (University of Birmingham, Verenigd Koninkrijk)
Prof. dr. F.W. Vaandrager
Prof. dr. E.R. Verheul

# Acknowledgements

Many people contributed in some way to this thesis and I would like to express my gratitude to all of them. First of all, I would like to thank my supervisor, Erik, who guided me through the world of academia. You always had time for our many fruitful discussions or to provide feedback on my writings. It was a pleasure working with you. Furthermore, I would like to thank Bart Jacobs for being my promotor and Karthikeyan Bhargavan, Wan Fokkink, Mark Ryan, Frits Vaandrager and Eric Verheul for being on my reading committee.

During my PhD, I worked together with many great people. I would like to thank David for the interesting discussions at our weekly meetings (if the NS allowed it). My thanks also go to everyone else I worked with over the last years on various interesting projects: Arjan, Fides, Flavio, Georg, Gerhard, Jordi, Matthew, Pim, Roel, Ronny, Stefan, Tim, Tom, Wan and all the others I might have forgotten to mention here.

I had a wonderful time working in the Digital Security group with great colleagues. In particular I would like to mention Bariş, in whom I found a good friend. I would also like to thank my officemates for the nice working environment and the welcome distractions: Roel, Bárbara, Fabian, Freek, Brenda and all the people from the large PhD room. I enjoyed not only the diversity of both people and expertise within the Digital Security group, both also the social aspects, such as the daily coffee breaks.

During my PhD I spent three months at Microsoft Research in Cambridge. This was a great time and I would like to thank Andy and Santiago for giving me the opportunity to work at this inspiring place, as well as my other colleagues there that helped making it an unforgettable experience, in particular Katja, Wudi, Jakob, Martin and Bhaskar.

I am very grateful to my family: my parents, Eef and Maria, my brother, Wouter, and my sister, Lisa. You are always there for me and support me in everything I do.

Finally, I would like to thank Claudia for helping me getting through the last bits of my PhD and supporting me in the choices I made, even though I know this is not always easy. I am happy we found each other.

Joeri de Ruiter

Birmingham (United Kingdom), July 2015

# Summary

Security protocols play an important role in our everyday life. In particular, they are used when paying with bank cards or performing online payments. In this thesis we look at the complex security protocols that are used in these systems: the EMV protocol family, used in bank cards, and the TLS protocol, used to secure network connections. Just the specifications of these protocols can already be hundreds of pages long. We analyse these protocols from different angles. On the one hand we perform a formal analysis of protocol specifications. For this analysis we specify the protocol in F# and make use of the ProVerif tool for the verification. On the other hand we also look at actual implementations of protocols. For this we make use of automated learning techniques to infer state machines from implementations of security protocols.

The first protocol we discuss is EMV, the world's leading standard for payments with smart cards: at the moment over 1.5 billion EMV cards are in use and worldwide almost all bank cards with a chip on them are EMV compliant. The core specifications are over 700 pages and every payment processor has additional proprietary specifications on top of this. Using ProVerif and FS2PV we analysed the EMV specifications, modelled in F#. Though the model was quite large and resulted in an even larger pi-calculus model, we were still able to perform an automated analysis. This revealed known weaknesses for EMV. To analyse implementations of EMV, we performed an analysis using automated learning techniques on EMV bank cards.

For online banking, many banks let their clients use a hand-held smart card reader with a small display and keypad. In combination with a smart card and PIN code, this reader then signs challenges provided on the bank's website, to either log in or confirm bank transfers. Many of these systems use a proprietary standard of MasterCard called the Chip Authentication Program, also known as EMV-CAP. This standard is based on the EMV standard. We analysed a device used for online banking that makes use of this security protocol, the e.dentifier2. Both a manual and a automated analyses were performed of this device that show that it contains a security issue. Based on our observations we present a solution, called the Radboud Reader, to

secure online transactions that provides stronger security guarantees than existing solutions.

When using online banking websites, or visiting other websites for which security is important, the TLS protocol is used. Using the same automated learning techniques as we used in the previous analyses, we analyse eight different TLS implementations. This resulted in the discovery of various security flaws and functional bugs in widely used implementations.

# Samenvatting

Security protocollen spelen een belangrijke rol in ons alledaagse leven. Met name bij het betalen met bankpassen of het gebruik van online bankieren. In dit proefschrift kijken we naar de complexe security protocollen die worden gebruikt in deze system: de EMV protocol familie, gebruikt in bankpassen, en het TLS protocol, gebruikt om netwerkverbindingen te beveiligen. Alleen de specificaties van deze protocollen kan al honderden pagina's lang zijn. We analyseren deze vanuit verschillende kanten. Aan de ene kant voeren we een formele analyse uit van de protocol specificaties. Voor deze analyse specificeren we het protocol in F# en maken we gebruik van de ProVerif tool voor de verificatie. Aan de andere kant kijken we ook naar daadwerkelijke implementaties van protocollen. Hiervoor maken we gebruik van geautomatiseerde leertechnieken om toestandsdiagrammen van implementaties van security protocollen af te leiden.

Het eerste protocol waar we naar kijken is EMV, de grootste standaard wereldwijd voor betalingen met smart cards: momenteel zijn er meer dan 1,5 miljard EMV kaarten in gebruik en bijna alle kaarten met een chip erin zijn EMV compliant. De basis specificaties bevatten meer dan 700 pagina's en elke payment processor heeft bijkomende eigen specificaties die hier bovenop komen. Met behulp van ProVerif en FS2PV hebben we een analyse gemaakt van de EMV specificaties, gemodelleerd in F#. Hoewel dit model behoorlijk groot was en resulteerde in een nog groter pi-calculus model, konden we nog steeds een automatische analyse uitvoeren. Hierbij werden bekende zwakheden van EMV gevonden. Om implementaties van EMV te analyseren hebben we een analyse met behulp van geautomatiseerde leertechnieken gebruikt uitgevoerd van EMV bankpassen.

Om te internetbankieren laten veel banken hun klanten gebruik maken van een paslezer met een klein scherm en toetsenbord. In combinatie met een smart card en PIN code, kan deze lezer challenges op de webstie van de bank ondertekenen om in te loggen of overschrijvingen te bevestigen. Veel van deze systemen maken gebruik van een eigen standaard van MasterCard genaamd Chip Authentication Program,

ook wel bekend als EMV-CAP. Deze standaard is gebaseerd op de EMV standaard. We hebben een lezer geanalyseerd die gebruik maakt van dit security protocol: de e.dentifier2. Zowel een handmatige als een automatische analyse van deze lezer wezen uit dat het een zwakheid bevat. Gebaseerd op onze observaties presenteren we een oplossing, genaamd de Radboud Reader, voor het beveiligen van online transacties die sterker veiligheidsgaranties geeft dan bestaande oplossingen.

Wanneer websites voor internetbankieren, of andere websites waarvoor beveiliging belangrijk is, worden bezocht wordt gebruik gemaakt van het TLS protocol. Met gebruik van dezelfde geautomatiseerde leertechnieken zoals gebruikt in de voorgaande analyses, analyseren we acht verschillende TLS implementaties. Dit resulteerde in de ontdekking van verschillende veiligheidsproblemen en functionele fouten in veelgebruikte implementaties.

# Contents

**Acknowledgements** **vii**

**Summary** **ix**

**Samenvatting** **xi**

**1 Introduction** **1**

1.1 Overview . . . . . . . . . . 3

**2 Preliminaries** **5**

2.1 Smart cards . . . . . . . . . . 5

2.1.1 ISO/IEC 7816 . . . . . . . . . . 5

2.1.2 Attacking smart cards . . . . . . . . . . 6

2.2 Automated analysis of security protocols . . . . . . . . . . 8

2.2.1 Attacker models . . . . . . . . . . 8

2.2.2 ProVerif . . . . . . . . . . 8

2.2.3 FS2PV . . . . . . . . . . 9

2.3 Automated learning . . . . . . . . . . 10

2.3.1 Mealy machines . . . . . . . . . . 10

2.3.2 Inference of Mealy machines . . . . . . . . . . 11

2.3.3 LearnLib . . . . . . . . . . 12

**3 EMV** **13**

3.1 EMV fundamentals . . . . . . . . . . 14

3.2 An EMV protocol session . . . . . . . . . . 16

3.2.1 Initialisation . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.2.2 Card authentication . . . . . . . . . . . . . . . . . . . . . . 19
3.2.3 Cardholder verification . . . . . . . . . . . . . . . . . . . . . 20
3.2.4 The transaction . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.2.5 Script processing . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Known attacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3.1 Relay attack . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.3.2 Cloning SDA cards . . . . . . . . . . . . . . . . . . . . . . . 25
3.3.3 DDA wedge attack . . . . . . . . . . . . . . . . . . . . . . . 25
3.3.4 No-PIN attack . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.3.5 Fallback to plaintext PIN . . . . . . . . . . . . . . . . . . . 26
3.3.6 Pre-play attack . . . . . . . . . . . . . . . . . . . . . . . . . 27

**4 Analysing EMV** **29**
4.1 Formal analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.1 Card and terminal configuration options . . . . . . . . . . . . 30
4.1.2 DOLs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1.3 Adding events to express security requirements . . . . . . . . 31
4.1.4 Security requirements . . . . . . . . . . . . . . . . . . . . . . 32
4.1.5 Experiences using ProVerif . . . . . . . . . . . . . . . . . . . 34
4.1.6 Including the issuer . . . . . . . . . . . . . . . . . . . . . . . 34
4.1.7 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.2 Learning models of bank cards . . . . . . . . . . . . . . . . . . . 35
4.2.1 Test harness . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2.2 Trimming the inferred state diagrams . . . . . . . . . . . . . 39
4.2.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.2.4 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**5 Securing online transactions** **49**
5.1 EMV-CAP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2 What-You-See-Is-What-You-Sign . . . . . . . . . . . . . . . . . . 51
5.3 e.dentifier2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.3.1 Unconnected mode . . . . . . . . . . . . . . . . . . . . . . . 53
5.3.2 Connected mode . . . . . . . . . . . . . . . . . . . . . . . . 54
5.3.3 The SWYS protocol . . . . . . . . . . . . . . . . . . . . . . . 54
5.3.4 The attack . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

5.3.5 Extended length challenges . . . 57
5.3.6 Preventing the vulnerability . . . 58
5.4 Analysing the e.dentifier2 using Lego . . . 59
5.4.1 Lego robot . . . 59
5.4.2 Software . . . 60
5.4.3 Experiments . . . 61
5.4.4 Discussion of obtained models . . . 63
5.4.5 Non-deterministic behaviour . . . 64
5.4.6 Conclusions . . . 65

**6 Radboud Reader 67**
6.1 Security objectives and high-level design . . . 68
6.2 Attacker model . . . 69
6.3 High-level design decisions . . . 69
6.4 Functional requirements . . . 71
6.5 Detailed design . . . 73
6.5.1 Format of data sent to the smart card . . . 74
6.5.2 Format of data sent by the smart card . . . 75
6.6 Prototype reader . . . 75
6.7 Differences with existing devices . . . 77
6.8 Conclusions . . . 79

**7 Analysing TLS 81**
7.1 Related work . . . 81
7.2 The TLS protocol . . . 82
7.3 Learning . . . 83
7.4 Test harness . . . 84
7.5 Results . . . 85
7.5.1 GnuTLS . . . 91
7.5.2 mbed TLS . . . 92
7.5.3 Java Secure Socket Extension . . . 92
7.5.4 miTLS . . . 95
7.5.5 RSA BSAFE for C . . . 97
7.5.6 RSA BSAFE for Java . . . 97
7.5.7 Network Security Services . . . 97
7.5.8 OpenSSL . . . 99
7.5.9 nqsb-TLS . . . 104
7.6 Conclusion . . . 104

**8 Conclusions** **109**
8.1 Future work . . . 110

**Bibliography** **113**

**Index** **123**

**Acronyms** **125**

**Curriculum vitae** **129**

# Chapter 1

# Introduction

Security protocols are employed all the time in everyday life when using electronic communication, often without users being aware of it. A security protocol is a combination of cryptographic computations and exchanges of messages in order to perform some security-sensitive operation between two or more parties. They are invoked when connecting to a website using HTTPS or paying for groceries at the supermarket using your bank card. To make sure your bank card works in every shop and you can visit every website, these protocols are standardised. In this thesis we explore methods for better and more systematic security analyses of security protocols and their implementations. This is done by considering real-life complex case studies from the financial sector and the Internet. Since there are many parties involved in writing these standards they can get very long and ambiguous to allow for flexibility within the standard. For example, for the EMV protocol, which is used throughout this thesis, the core specification consists of 4 books covering over 700 pages. One of the reasons specifications tend to become long is backwards compatibility. Still one can question whether these specifications really have to be this long and complex. Security protocols are notoriously hard to get right, even if they do not need over 700 pages of specifications:

> *"Security protocols are three-line programs that people still manage to get wrong"*
> – Roger Needham

As these specifications will be the basis of any implementation of the protocol, it is important that they are correct and no security weaknesses exist in the protocol itself. This can, for example, be ensured by using formal methods, where a protocol is analysed using rigorous mathematical techniques. An example of this is resolution of Horn clauses as used in the ProVerif tools [Bla01]. Formal methods are mainly used on abstract small protocols. In this thesis it is shown that this technique not only works for these small protocols but also for the ones used in complex real-life systems. A comparable complex protocol that also has been verified is, for example, UMTS [AMR+12].

However, having a correct protocol is only one side of the story. The protocol will be implemented and, especially with large and complex standards, it is not trivial to do this correctly. Where with a 'regular' protocol a mistake in an implementation might result in a system that does not work properly, a mistake in a security protocol might

lead to problems where confidential information is leaked or authentication could be bypassed. It is therefore important to not only check whether protocol specifications are secure, but also to analyse whether implementations of these protocols adhere to the specifications and do not introduce any security problems.

As running case studies in this thesis we look at the EMV protocol family and TLS. EMV is one of the most important group of payment protocols: it is used in over 1.5 billion bank cards in the world and is used in almost all bank cards within Europe [EMV08]. Not only are the core specifications huge, also every payment system has additional proprietary specifications on top of this. Over the years various problems with EMV have been discovered. These discoveries where made in manual ad-hoc ways. We present an automated analysis of the EMV protocol specifications, where we make use of formal methods. Next to analysing the specifications, we also analyse actual implementations of EMV on real bank cards. To analyse these implementations we used automated learning techniques to learn their working. This automated learning results in models of the state machines of these cards, which can easily be inspected manually or using a model-checker. This revealed a wide range of implementation differences, but no security flaws.

When using online banking, many banks provide their customers with handheld smart card readers with a small display and keypad. Many of these systems use a proprietary standard of MasterCard called the Chip Authentication Program, also known as EMV-CAP. This is a proprietary standard on top of the EMV standard. Examples of these devices are the Random Reader from the Rabobank and the e.dentifier2 from ABN AMRO. If a customer wants to perform a transaction, first the PIN code is entered on this device, which is checked by the bank card. After this the customer enters a challenge that is displayed on the bank's website. This challenge is then used by the device to compute a response to confirm the transaction. This response is computed using cryptographic operations on the bank card. We analysed one of these devices, the e.dentifier2, using both manual and automated analysis. This resulted in the discovery of a security flaw in the device. The security guarantees provided by this device are not optimal and constructing the protocols, used in the device, on top of EMV-CAP might not be the best solution. In our automated analysis it became clear that the implementation is more complex than necessary. Based on these observations we designed the Radboud Reader, a minimal handheld smart card reader that can be used to authenticate online transaction.

To secure the connection between a web browser and server, the HTTPS protocol is used. The security in this protocol is provided by the TLS protocol (also known as SSL). This is used by – hopefully – all banks as a first step to secure their online banking. Because it is such a widely used protocol, TLS attracted a lot of attention from researchers. This led to many problems found in both the specification and implementations. Many implementations of TLS exist and we developed an automated analysis that we applied to the most widely used implementations. This analysis led to the discovery of new security bugs.

## 1.1 Overview

**Chapter 2** In this chapter we introduce the necessary technical background on smart cards and the tools used for our formal analysis and automated state machine learning.

**Chapter 3** In this chapter we introduce background on the EMV standard and known attacks. This is a first comprehensive but concise overview of EMV. In this chapter the following publication is discussed, which provides a mitigation for one of the known attacks:

- *A Lightweight Distance-Bounding Contactless EMV Payment Protocol* by Tom Chothia, Flavio Garcia, the author, Jordi van den Breekel and Matthew Thompson [CGR+15]. For this paper my contribution was in providing the necessary background in EMV on how the proposed solution fits within the EMV standard and I was involved in setting up the relay attacks described.

**Chapter 4** In this chapter we perform a formal analysis on EMV, both on the abstract level of the specifications and on actual implementations on bank cards. The formal verification of specifications is done using the existing tools ProVerif and FS2PV. This is the first formal specification of all of EMV and one of the largest formalisations of a security protocol suite. To analyse an actual implementation of EMV, we use automated learning techniques to extract the internal state machines from bank cards. This chapter is based on the following publications:

- *Formal Analysis of the EMV Protocol Suite* by the author and Erik Poll [RP12].
- *The SmartLogic Tool: Analysing and Testing Smart Card Protocols* by Gerhard de Koning Gans and the author [KGR12]. My contribution is in the case study on EMV that we performed together.
- *Formal Models of Bank Cards for Free* by Fides Aarts, the author and Erik Poll [ARP13]. My contribution was in constructing the test harness to communicate with EMV cards necessary for the automated learning and the analysis of the resulting models.

**Chapter 5** After analysis of the 'core' specifications of EMV in Chapter 4 we discuss EMV-CAP in this chapter. This variant of EMV is used to perform online banking transactions using an EMV card and a handheld reader. We look at a USB-connected EMV-CAP reader by Gemalto, the e.dentifier2. To automatically analyse this device, we extend our automated learning technique to be able to handle physical key presses and sending of USB commands. This chapter is based on the following publications:

- *Designed to Fail: A USB-Connected Reader for Online Banking* by Arjan Blom, Gerhard de Koning Gans, Erik Poll, the author and Roel Verdult [BKGP+12]. My contribution was in the analysis of the handheld smart card reader.

- *Automated Reverse Engineering using Lego* by Georg Chalupar, Stefan Peherstorfer, Erik Poll and the author [CPPR14]. My main contribution is in the analysis of the results and fine-tuning of the automated learning process. Georg and Stefan built the original robot, which is controlled using a Raspberry Pi.

**Chapter 6** The problems with the e.dentifier2 discussed in Chapter 5 suggested that during the design the security objectives and attacker were not explicitly taken into account. In this chapter we present our optimal solution for a USB-connected card reader for securing online transactions: the Radboud Reader. This device has a minimal trusted computing base and provides the strongest possible security guarantees. This chapter is based on the following publication:

- *The Radboud Reader: A Minimal Trusted Smartcard Reader for Securing Online Transactions* by Erik Poll and the author [PR13].

**Chapter 7** In this chapter we apply the automated learning technique that was applied previously on bank cards (Chapter 4) and the e.dentifier2 (Chapter 5) to a different kind of security protocol, namely the Transport Layer Security (TLS) protocol. This chapter is based on the following publication:

- *Protocol state fuzzing of TLS implementations* by the author and Erik Poll [RP15].

Other papers published during the author's PhD research, but not directly related to this thesis, are [WRF12, CRP14, Rui14].

# Chapter 2

# Preliminaries

In this chapter we give some background on the methods and tools used in the rest of this thesis. As smart cards play a big role in most of the protocols we will analyse, these are discussed in Section 2.1. After this we discuss the automated analysis of security protocols, which we will use in our analysis of the EMV specifications in Chapter 4. This is followed by the introduction of automated learning techniques, which we use in our analysis of implementations of EMV and EMV-CAP, the e.dentifier2 and TLS in chapters 4, 5 and 7 respectively.

## 2.1 Smart cards

Smart cards are widely used as, for example, bank cards, access cards, passports and public transportation cards. A smart card, also known as Integrated Circuit Card (ICC), is basically a small computer, with its own processor and memory, that can perform computations. As they are mostly used for security purposes, they often contain a cryptographic co-processor as well. Communication between a card and a terminal, like a Point of Sale (POS) or ATM, can be done using either the contacts on the card or contactless via radio waves.

### 2.1.1 ISO/IEC 7816

For terminals and smart cards to be able to communicate they have to speak the same language. To achieve this the international standard ISO/IEC 7816 was introduced [ISOb]. This standard specifies various aspects related to smart cards, ranging from physical characteristics, like dimensions of cards and positions of contacts, to formats of data that is exchanged, and from electrical specifications to a full card application for cryptographic operations. The standard is widely used and most smart cards confirm to it. The standard was originally intended for contact-based smart cards, but the higher level aspects, like the data formats, are also used for contactless cards. For the lower level details in the contactless communication ISO/IEC 14443 can be used [ISOa]. This standard is for example used in passports and contactless bank cards.

ISO/IEC 7816 allows multiple applications to be present on a smart card. To distinguish the different application, each one is identified by an Application Identifier

(AID). Using the so-called SELECT command, that is specified in the standard, the terminal can indicate with which application it wants to interact. AIDs for certain applications have been standardised, and a smart card can have one application that is selected by default.

#### APDUs

For the communication between a card and terminal, the ISO/IEC 7816 standard specifies so-called Application Protocol Data Units (APDUs). The communication between a terminal and card is done using a master/slave protocol. This means the terminal will always send a command to the card first, after which the card will send a response. The terminal sends Command APDUs to the card, to which the card will respond with a Response APDU. A Command APDU is structured as follows:

| CLA | INS | P1 | P2 | (Lc) | (Data) | (Le) |
|---|---|---|---|---|---|---|

The first two bytes, also known as class and instruction byte respectively, indicate the command that the terminal requests the card to perform. The bytes P1 and P2 can be used to specify additional parameters. The Lc byte is used to indicate the length of the data that will follow, if this data is present. Le is used to indicate the length of data the terminal expects in return to the command. The card's Response APDU is formatted as follows:

| (Data) | SW1 | SW2 |
|---|---|---|

The card can optionally return data to the terminal, but the response will always end with two bytes called the Status Word. This Status Word indicates whether the operation was completed successfully (typically by returning 9000) or whether an error occurred. It can also be used to indicate the card has more data available for the terminal.

### 2.1.2 Attacking smart cards

When attacking smart cards, we distinguish between active and passive attacks. These attacks can be either targeted on the communication between terminal and card, or on physical properties of the card, such as power usage or electromagnetic radiation. This second kind of attack is known as side-channel attacks. With passive attacks we only observe the communication between genuine cards and terminals without interfering. This information can then be used, for example, to reverse engineer the protocol that is used or harvest interesting data, such as personal data or PIN codes. An example is a device that can be used for this is the APDU Scanner by Rebel Simcard[1] or the Season Passive 2 Interface by Interesting Devices Ltd[2]. Though these devices are

[1] http://rebelsimcard.com
[2] http://interesting-devices.com

respectively targeted at users that analyse SIM cards to unlock phones or want to share cards used in set-top boxes for digital television, they can be used for other smart cards and readers as well.

With active attacks the communication between the card and the terminal is modified by a Man-in-the-Middle (MitM). This can already be done using very small pieces of hardware called shims. These devices are placed in between the contacts of the card and the terminal and contain a microprocessor to process all communication. As all the communication goes through the shim it is able to modify, drop or add data. This technique is, for example, used in phones, such that they can work with different mobile network operators apart from the one that sold the phone. A more generic tool called the SmartLogic is presented in [KGR12]. This tool can be used to perform MitM attacks or even completely emulate cards. Usually published active attacks are not reproduced because it is a time consuming and tedious job. The SmartLogic lowers the effort needed to mount such an attack and makes it easier to verify published results.

Another option for an active attack is to completely replace the terminal by a modified version. This is what happened at branches of the ABN AMRO bank in The Netherlands during 2008 and 2009 [3]. Criminals replaced e.dentifiers, handheld readers that are used for online banking, in the bank with their own modified versions. The modified devices functioned like the original devices, but had some added functionality. Next to computing the required response that had to be entered on the bank's website, they also stored the customer's Personal Identification Number (PIN) code and data to reconstruct the magnetic stripe of the card. This data could later be collected by the criminals using a so-called 'download card', a specially programmed smart card that was detected by the devices and to which all the saved data would then be transferred. Using this data the criminals could create cloned cards that would contain the correct data on the magnetic stripe and for which they would know the PIN.

Apart from the communication we can also analyse physical properties of the card during security sensitive operations, such as cryptographic computations. An example of these passive side-channel attacks is the analysis of power usage using Simple Power Analysis (SPA) or Differential Power Analysis (DPA), which could reveal key material on the card [KJJ99]. Fault injections are a type of active side-channel attacks where the attacker tries to influence the operation of a smart card [BECN$^+$04]. This could for example be done by glitching the voltage or clock supplied to the card. The goal is then to skip instructions or make them produce wrong results, and use this to circumvent security measures or learn sensitive data.

In this thesis we will focus on active attacks on protocol specifications and implementations, where a possible MitM can change the communication. We research systematic defences against these attacks by using both formal analysis of specifica-

[3] http://deeplink.rechtspraak.nl/uitspraak?id=ECLI:NL:RBROT:2011:BU6141 and http://deeplink.rechtspraak.nl/uitspraak?id=ECLI:NL:RBROT:2011:BU6142

tions and automated learning techniques to analyse implementations.

## 2.2 Automated analysis of security protocols

Designing and analysing security protocols is very error prone and difficult to do by hand. Automated analysis can help here to perform a more systematic analysis and reduce human errors. A typical example of the difficulty of analysis of security protocols is the Needham-Schroeder protocol [NS78]. Though the protocol was published in 1978, it wasn't until 1995 when Lowe found an attack against the protocol and later showed an automated analysis on it that found the attack [Low95, Low96]. For this work Lowe used a generic model checker. Nowadays, a lot of tools exist that are specifically designed to analyse security protocols. One example of a tool that can be used to analyse specifications of security protocols is ProVerif, which we will discuss below [Bla01]. To analyse implementations of security protocols we make use of the automated learning techniques that are introduced in Section 2.3.

### 2.2.1 Attacker models

When analysing security protocols we need to define the attacker model, i.e. the capabilities that an attacker has to attack a protocol. The attacker model that is used most is the so-called Dolev-Yao attacker [DY83]. This attacker is assumed to control the communication medium. This means he can change, drop or inject messages. However, he is computationally bounded as it is assumed that he cannot break the cryptographic operations. More specifically, he cannot perform any cryptographic operation without knowing the correct key.

### 2.2.2 ProVerif

ProVerif is an automated protocol analyser that makes use of resolution of Horn clauses for its analysis [Bla01]. Several input formats can be used to provide the specifications, but we will only consider the extension of the untyped pi-calculus as introduced in [AB05a] and depicted in Figure 2.1.

No built-in cryptographic operations are provided, so these have to be defined using constructors and destructors. This results in cryptographic operations that are assumed to be perfect, i.e. data cannot be decrypted without knowledge of the correct key and neither can signatures be computed without this knowledge. Due to some approximations, the tool can handle an unbounded number of sessions and an unbounded message space. The drawback of this is that ProVerif might return false attacks or is not able to answer queries at all. However, if it reports that there a security property holds this is correct. In the analysis, two types of attackers can be considered: an active attacker, that is equal to a Dolev-Yao attacker, or a passive attacker, that only monitors the public communication channels but cannot modify

| | |
|---|---|
| M, N ::= | term |
| x, y, z | variable |
| a, b, c, k, s | name |
| f($M_1$, ..., $M_n$) | constructor application |
| | |
| P, Q ::= | processes |
| out M(N).P | output |
| in M(x).P | input |
| 0 | nil |
| P—Q | parallel composition |
| !P | replication |
| (new a).P | restriction |
| let x = g($M_1$, ... $M_n$) in P else Q | destructor application |
| let x = M in P | local definition |
| if M = N then P else Q | conditional |

Figure 2.1: Syntax of untyped pi-calculus used by ProVerif [AB05a]

or inject messages. If an attack is found, the tool will try to reconstruct an example of an attack trace [AB05b].

ProVerif supports several types of queries to check security properties of protocols. We will consider two types of queries in this thesis:

- *secrecy*: this is the most basic type of query and it is used to check if it is possible for the attacker to learn the value of a given term.
- *authentication*: this type of query is used to check a relation between different events. Two types of queries exist to check this. The first is of the form

  ```
  ev:EventA() ==> ev:EventB()
  ```

  This query checks that if event `EventA` is executed, then event `EventB` was executed before.

  The second type of query that can be verified is

  ```
  evinj:EventA() ==> evinj:EventB()
  ```

  In this case, for every execution of event `EventB` there exists a distinct executed event `EventA` that was executed before.

### 2.2.3 FS2PV

At Microsoft Research, the tool FS2PV was developed [BFGT08]. This tool can be used to convert F# code – a functional programming language developed by Microsoft

– to the untyped pi-calculus that is used as input to ProVerif. Though only a subset of F# can be used, this already provides much more flexibility by the use of functions and sequential if-statements.

## 2.3 Automated learning

Finite state machines (a.k.a. finite automata) are a very useful formalism to model the behaviour of systems. For security-sensitive systems, they can be used to confirm that actions can only be carried out in the correct order, e.g. that some security-sensitive action is only possible after a successful PIN code check. Implementing a security protocol inevitably involves the implementation of a state machine that checks that messages are only accepted in the correct order. Automatically learning these state machines is a very useful technique that can be used to automatically reverse engineer implementations of security protocols, with the view to find security flaws or confirm their absence. The vulnerabilities that can be discovered using this technique are the ones that occur when performing actions in an unexpected order, e.g. performing a security sensitive operation before having entered a PIN code. On the other hand, we cannot hope to discover carefully hidden backdoors in an implementation. Still, as our experiments will confirm, we can hope to find accidental mistakes in the program logic.

A widely used technique for creating a model from observations is *regular inference*, also known as automata learning [Ang87]. The regular inference algorithm from [Nie03, RSBM09] provides sequences of inputs to a System Under Test (SUT) and observes the responses to infer a state machine as explained in Section 2.3.2. The state machines that are inferred are Mealy machines, a special form of finite state machines that will be explained in Section 2.3.1. In addition to standard learning methods, abstraction techniques for data parameters [AJU10, ASV10] can be used to learn a more detailed model of the system.

### 2.3.1 Mealy machines

We use *Mealy machines* to model the behaviour of smart cards. A Mealy machine is a finite state machine where every transition involves an input and a resulting output. Formally, a *Mealy machine* is a tuple $\mathcal{M} = \langle I, O, Q, q^0, \delta, \lambda \rangle$, where $I$, $O$ and $Q$ are nonempty sets of *input symbols*, *output symbols*, and *states*, respectively; $q^0 \in Q$ is the *initial state*; $\delta : Q \times I \to Q$ is the *transition function*; and $\lambda : Q \times I \to O$ is the *output function*. Elements of $I^*$ and $O^*$ are *input* and *output strings* respectively.

An intuitive interpretation of a Mealy machine is as follows. 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 $i \in I$. The machine then responds by producing an output symbol $\lambda(q, i)$ and transforming itself to the new state $\delta(q, i)$. Let a *transition* $q \xrightarrow{i/o} q'$ in $\mathcal{M}$ denote that $\delta(q, i) = q'$ and $\lambda(q, i) = o$.

The Mealy machines that we consider are *complete* and *deterministic*, meaning that for each state $q$ and input $i$ exactly one next state $\delta(q, i)$ and output symbol $\lambda(q, i)$ is possible.

### 2.3.2 Inference of Mealy machines

Angluin's well-known L* [Ang87] is an active learning algorithm to infer deterministic finite automata. Inspired by work of Angluin, Niese [Nie03] developed an adaptation of the L* algorithm for active learning of deterministic Mealy machines. The algorithm assumes there is a *Teacher*, who knows a deterministic Mealy machine $\mathcal{M} = \langle I, O, Q, q^0, \delta, \lambda \rangle$, and a *Learner*, who initially has no knowledge about $\mathcal{M}$, except for its sets $I$ and $O$ of input and output symbols. Whenever the *Teacher* accepts an input symbol on $\mathcal{M}$, it maintains the current state of $\mathcal{M}$, which at the beginning equals the initial state $q^0$. The *Learner* can ask three types of queries to the *Teacher*:

- An *output query* $i \in I$. Upon receiving output query $i$, the *Teacher* picks a transition $q \xrightarrow{i/o} q'$, where $q$ is the current state, returns output $o \in O$ as answer to the *Learner*, and updates its current state to $q'$.

- A *reset query*. Upon receiving a reset query the *Teacher* resets its current state to $q^0$.

- An *equivalence query* $\mathcal{H}$, where $\mathcal{H}$ is a hypothesised Mealy machine. The *Teacher* will answer *yes* if $\mathcal{H}$ is correct, that is, whether $\mathcal{H}$ is equivalent to $\mathcal{M}$, or else supply a *counterexample*, which is a string $u \in I^*$ such that $u$ produces a different output string for both automata, i.e., $\lambda_{\mathcal{M}}(u) \neq \lambda_{\mathcal{H}}(u)$.

When it comes to answering equivalence queries, there are two possibilities. In a *white-box setting*, the teacher is assumed to know the internal implementation of $\mathcal{M}$, so that he can simply see if the hypothesis automaton is correct. In a *black-box setting*, the teacher cannot access the internal implementation; then the teacher can only resort to black-box testing of the target to see if he can observe a difference. In such cases, the equivalence query can only be approximated, namely by testing to see if a difference between the actual machine $\mathcal{M}$ and the hypothesis $\mathcal{H}$ can be detected. (Note that this is a form of model-based testing.) As we will discuss in Section 2.3.3, there are different strategies to test the equivalence.

Note that in a black-box setting, equivalence queries cannot be answered with certainty: there may be differences between the hypothesis and the real implementation that do not show up in the finite number of tests that are run. For example, we cannot exclude the possibility that if we keep repeating some input many times, the real implementation will do something different than the hypothesis automaton on the millionth time. The automaton that is ultimately inferred might only show a subset of the actual behaviour.

The typical behaviour of a *Learner* is to start by asking sequences of output queries (alternated with resets) until a "stable" hypothesis $\mathcal{H}$ can be built from the answers. After that an equivalence query is made to find out whether $\mathcal{H}$ is correct. If the answer is *yes* then the *Learner* has succeeded. Otherwise the returned counterexample is used to perform subsequent output queries until converging to a new hypothesised automaton, which is supplied in an equivalence query, etc.

### 2.3.3 LearnLib

The LearnLib tool provides a Java implementation of the adapted L* algorithm [RSBM09]. Because LearnLib views the SUT as a black box, equivalence queries can only be approximated. The tool provides several different realisations for checking equivalence queries.

To approximate equivalence checking of the teacher, we use either the random walk method or Chow's W-method [Cho78] as provided by LearnLib. For the random walk method, LearnLib will try to verify a hypothesis by generating random input traces and checking the corresponding outputs from the device against the hypothesis. The maximum number of traces is given, as well as a minimum and maximum trace length. When using the W-method it can be guaranteed that, given an upper bound on the number of states, the correct state machine is found. This does come at a cost of a possibly much longer running time compared to the random walk method and requires an upper bound to be specified. In practice we usually first use the random walk method to get an upper bound for the number of states. This is then used to perform an analysis using the W-method and to verify the results of the random walk method.

# Chapter 3

# EMV

EMV is the world's leading standard for payments with smart cards. The initiative for EMV was taken by Europay, MasterCard and Visa in 1994 and the first version was released in 1996. Since 1999, the EMV specifications are maintained by EMVCo, a company currently jointly owned by American Express, Discover, JCB, MasterCard, UnionPay and Visa. According to EMVCo, the number of EMV cards in use at the moment is over 1.5 billion. In Europe the shift to EMV was partly driven by the Single Euro Payment Area (SEPA), which requires compliance with EMV in their SEPA Cards Framework [Cou09]. In The Netherlands, all bank cards issued are EMV-compliant.

The EMV specification is publicly available and consists of four books [EMV11a, EMV11b, EMV11c, EMV11d], amounting to over 700 pages. These books do not define a single protocol, but a highly configurable toolkit for payment protocols. For instance, it allows a choice between:

- three *Card Authentication Methods (CAMs)*: Static Data Authentication (SDA), Dynamic Data Authentication (DDA), and Combined Data Authentication (CDA), discussed in more detail in Section 3.1;
- five *Cardholder Verification Methods (CVMs)*: none, signature, online PIN, offline plaintext PIN, and offline encrypted PIN;
- on- and off-line transactions.

Moreover, many of these options are again parameterised, as explained in Section 3.1, possibly using proprietary data and formats. All these options and parameterisations make the EMV standard very difficult to comprehend.

The Common Core Definition (CCD) was added to the EMV standard in 2004 and specifies a set of minimum implementation options and configurations for cards that are sufficient to perform an EMV transaction. It is included in the EMV specifications. In addition to CCD a separate specification was released by EMVCo in 2005 called the Common Payment Application (CPA) [EMV05]. In this specification data elements and functionalities for an EMV card application are described that comply with CCD. A card that complies with CPA would be accepted to be used for all international card schemes, such as MasterCard and Visa. Before CPA was released, every payment scheme had its own proprietary specification that cards should comply to. Now,

issuers who want to be able to use the same application for different payment schemes only have to make sure it is compliant with the CPA.

## 3.1 EMV fundamentals

This section explains the basic building blocks of EMV and some central concepts and terminology. In EMV several entities can be identified. First, we have the customer (also known as cardholder in the EMV specifications) who is in possession of his EMV-compliant bank card issued by his bank (also called the issuer). The customer can use the card to perform a transaction at a merchant. The merchant has a terminal, called a POS, that can communicate with the customer's card. The terminal also communicates with the merchant's bank (also called the acquirer), who has to authorise and perform the actual transaction. Another type of terminal is an ATM, where the user can use his card to withdraw cash. For an ATM the acquirer is the bank owning the ATM or the bank of the company owning the ATM. Terminals are trusted to provide the customer with secure input (a keypad that is trusted to enter the PIN code) and output (a display that is trusted to show transaction details).

#### Key setup

The essence of the key setup in EMV is as follows, making use of derived session keys and a Public Key Infrastructure (PKI):

- Every card has a unique symmetric key $K_{AC}$ that it shares with the issuer. This is typically a diversified key derived from a master key known only by the issuer. Using this key a session key $K_{SAC} = enc_{K_{AC}}(ATC)$ can be computed, which is used to authorise transactions. The Application Transaction Counter (ATC) is a counter that is increased on every initialisation. This key derivation method for session keys is not mandatory for EMV and issuers can implement their own methods.

- The Certificate Authority (CA) has a public-private key pair ($PK_{CA}$, $SK_{CA}$). This public key $PK_{CA}$ is known to the terminal.

- The issuer has a public-private key pair ($PK_I$, $SK_I$) and a certificate for this key pair signed by the CA. On all cards, the certificate of its issuer is present.

- Cards that support asymmetric cryptography also have a public-private key pair ($PK_{ICC}$, $SK_{ICC}$) and a certificate for this key pair signed by the issuer. Most cards in use nowadays will have support for asymmetric cryptography. This key pair is used to authenticate a card to terminals and to encrypt PIN codes that are sent to the card.

Payment scheme owners, such as MasterCard and Visa, act as CAs and provide the certificates for the issuers. This key setup is the basis of trust between the different parties.

This setup provides cards with two mechanisms to prove authenticity of data:

- All EMV cards can compute Message Authentication Codes (MACs) on messages, using the symmetric (session) keys shared with the issuing bank. The issuer can check these MACs to verify authenticity of the messages, but the terminal cannot as it does not know the shared symmetric keys $K_{AC}$ or $K_{SAC}$.
- Cards that support asymmetric cryptography can also digitally sign data to prove the authenticity to the terminal, as well as to the issuer.

### Cardholder authentication methods

The EMV standard defines three authentication mechanisms for a card to prove its authenticity to terminals: SDA, DDA, and CDA.

- *SDA (Static Data Authentication)*: the card provides some digitally signed data (e.g. the card number and expiry date) to the terminal to authenticate itself. The data is signed by the issuer and can thus be used by cards that do not support asymmetric cryptography. The terminal can check the authenticity of this data, since it can retrieve the issuer's public key. However, as all the data that is signed is static, this method does not rule out cloning.
- *DDA (Dynamic Data Authentication)*: the card proves its authenticity using a challenge-response mechanism. This requires the card to support asymmetric cryptography and have a public/private key pair. The card signs, using its private key, a challenge chosen by the terminal and a nonce generated by the card. The nonce generated by the card is stored in the terminal for possible later use during the transaction. The challenge from the terminal is required to include an unpredictable number, which should rule out cloning. However, after the data authentication, DDA does not tie the subsequent transaction to a specific card as far as the terminal can determine. In other words, the terminal cannot verify that the transaction was actually carried out by the same card as the data authentication.
- *CDA (Combined Data Authentication)*: repairs this deficiency of DDA. With CDA the card digitally signs all important transaction data, not only authenticating the card, but also authenticating any transaction it performs.

The data that is authenticated with SDA, referred to in the standard as *Static Data to be Authenticated*, is also authenticated with DDA or CDA. For these authentication methods a hash over this data is included in the certificate containing the public key of the card. In this sense DDA and CDA subsume SDA. All Dutch cards we

have seen are capable of performing DDA. Cards that support CDA we have not seen yet, even though any card that can do DDA are technically also capable of performing CDA. Both Visa and MasterCard did not allow new SDA cards to be issued after the beginning of 2011 and no more SDA should be in use anymore from 2015 [Vis11, Mas11].

### Data object lists

An important concept in the EMV specification is that of *Data Object Lists (DOLs)*, that are used to 'configure' various EMV commands. A DOL specifies a list of data elements, and the format of such a list. Example data elements are the card's *ATC*, the transaction amount, the currency, the country code, and card- or terminal-chosen nonces.

An EMV card supplies several DOLs to the terminal. Different DOLs specify which data the card expects as inputs in particular protocol steps (and then also the format that this data has to be in). This is explained in more detail in Section 3.2. The use of these DOLs make EMV highly parameterisable. The choices for DOLs are of crucial importance for the overall security, as they control which data gets signed or MACed, and hence no security analysis is possible without making some assumptions on the DOLs.

## 3.2 An EMV protocol session

An EMV protocol session can roughly be divided into five steps:

1. *initialisation*: selection of the application on the smart card and reading of some data;
2. (optionally) *data authentication*, i.e. authentication of the card, by means of SDA, DDA, or CDA;
3. (optionally) *cardholder verification*, by means of PIN or signature;
4. the actual *transaction*;
5. (optionally) *script processing*: the terminal may send additional Issuer-to-Card scripting commands, that allow the issuer to update cards in the field.

Each of these steps is described in more detail below. Here we use the usual semi-formal Alice-Bob style for security protocols, where square brackets indicate optional (parts of) messages. The card is here denoted by C, and the terminal by T. In Figure 3.1, an overview is given for an EMV session.

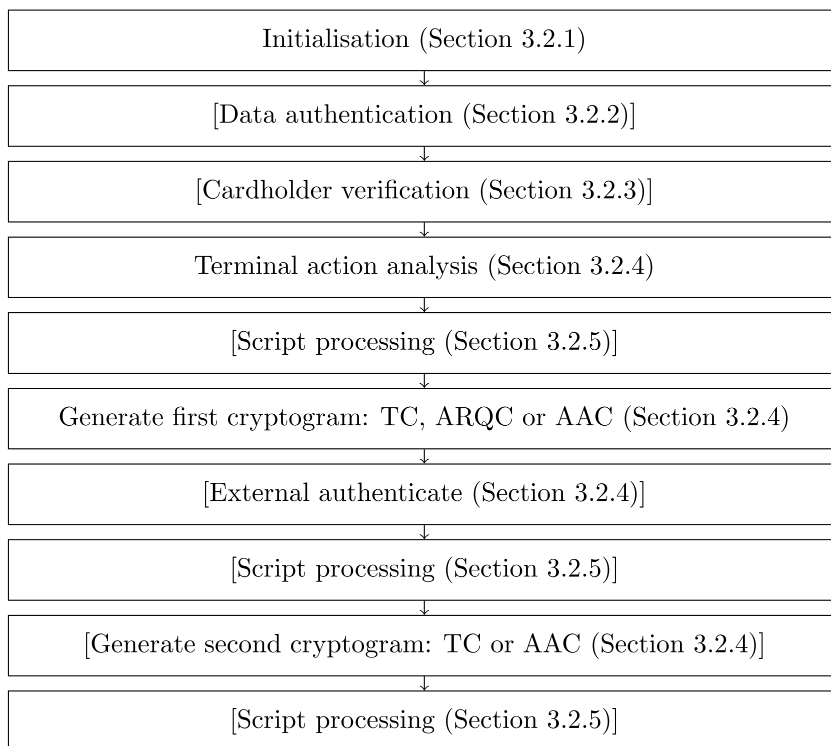


Figure 3.1: Different steps in an EMV session with the corresponding sections. Optional parts are indicated by square brackets.

T → C: `SELECT APPLICATION`
C → T: [PDOL]
T → C: `GET PROCESSING OPTIONS`, [(Data specified by the PDOL)]
C → T: (AIP, AFL)
Repeat for all files in the AFL:
T → C: `READ RECORD`, i
C → T: (Contents of file i)

Figure 3.2: Initialisation of an EMV session

### 3.2.1 Initialisation

In the first phase of an EMV session, the terminal obtains basic information about the card (such as card number and expiry date) and the information about the features the card supports and their configurations. This is information the terminal needs for the subsequent steps in the EMV session. Optionally, the card may require some information from the terminal. In Figure 3.2 the protocol steps are given for the initialisation, which will be explained in more detail below.

The protocol starts by selecting the payment application. In response to the `SELECT APPLICATION` command, the card optionally provides a *Processing Options Data Object List (PDOL)*. The PDOL specifies which data, if any, the card wants from the terminal; this could for instance include the Terminal Country Code or the amount. Note that none of this data is authenticated.

The card then provides its *Application Interchange Profile (AIP)* and the *Application File Locator (AFL)*. The AIP consists of two bytes indicating the supported features (SDA, DDA, CDA, cardholder verification and issuer authentication) and whether terminal risk management should be performed. During the terminal risk management, the terminal will decide whether to perform the transaction online based on, e.g., the amount of the transaction and the last time the card was used in an online transaction.

The AFL is a list identifying the files on the card to be used in the transaction. For each file it is indicated whether it is included in the data authentication as will be explained in Section 3.2.2. The files can be read using the `READ RECORD` command. Some additional data that is not present in the files, like the PIN try counter, can be retrieved using the `GET DATA` command, if supported by the card.

The following data objects are mandatory to be included in the files on the card:

- *Application Expiry Date*,
- *Application Primay Account Number (PAN)*,
- *Card Risk Management Data Object List 1 (CDOL1)*, and
- *Card Risk Management Data Object List 2 (CDOL2)*.

For cards that support SDA the *Signed Static Application Data (SSAD)* is also mandatory.

Note that none of the data provided by the card or by the terminal is authenticated yet at this stage. The process of card authentication, discussed below, will authenticate some of the data provided by the card.

### 3.2.2 Card authentication

As already mentioned, there are three card authentication methods. The AIP informs the terminal which methods are supported by the card, and the terminal should then choose the 'highest' method that both the card and the terminal support.

#### Static Data Authentication (SDA)

On cards that support SDA, the *SSAD* is the signed hash of the concatenation of the files indicated by the AFL, optionally followed by the value of the AIP. Whether the AIP is included is indicated by the optional *Static Data Authentication Tag List*, which can be included in one of the files. The signature in the SSAD is created by the issuer, whose certificate is also on the card. For SDA no additional communication is needed, since the data needed to verify the SSAD was already retrieved in the initialisation phase.

#### Dynamic Data Authentication (DDA)

DDA consists of two steps and requires some additional communication, as can be seen in Figure 3.3. First, the certificate containing the card's public key is checked. This certificate also contains the hash of the *Static Data to be Authenticated*, that is also authenticated with SDA.

Second, a challenge-response protocol is performed. To provide the challenge an `INTERNAL AUTHENTICATE` command is sent to the card. The argument of this command is the data specified by the *Dynamic Data Authentication Data Object List (DDOL)*. The DDOL can be supplied by the card, otherwise a default DDOL should be present in the terminal. The DDOL always has to contain at least a terminal-generated nonce (the *Unpredictable Number (UN)*).

The *Signed Dynamic Application Data (SDAD)* is the signed *ICC Dynamic Data* and hash of the ICC Dynamic Data and the data specified by the DDOL. The ICC Dynamic Data contains at least the *ICC Dynamic Number*, a time-variant parameter like a nonce or a counter. On the bank and credit cards we studied, the DDOL only specified the Unpredictable Number, and the ICC Dynamic Data only consisted of the ICC Dynamic Number.

T → C: `INTERNAL AUTHENTICATE`, (Data specified by DDOL)
C → T: $sign_{SK_{ICC}}$( ICC Dynamic Data,
H(ICC Dynamic Data, Data specified by DDOL))

Figure 3.3: DDA protocol step

**Combined Data Authentication (CDA)**

Card authentication using CDA does not require additional messages, but is combined with the actual transaction, which will be explained in Section 3.2.4. As with DDA, the Static Data to be Authenticated is authenticated using the certificate for the public key of the card.

With CDA, the SDAD is a signature on the ICC Dynamic Data and the UN:

$$SDAD = sign_{SK_{ICC}}(\text{ ICC Dynamic Data}, \mathsf{H}(\text{ICC Dynamic Data, Unpredictable Number}))$$

The ICC Dyamic Data will always include at least the ICC Dynamic Number, the Cryptogram Information Data (CID), the cryptogram and the *Transaction Data Hash Code (TDHC)*. The TDHC is a hash of the elements specified by the PDOL, the elements specified by the CDOL1 or CDOL2 and the elements returned by the card in the response.

### 3.2.3 Cardholder verification

Cardholder verification can be done in several ways: by a PIN, a handwritten signature, or it can simply not be done. The process to decide which CVM is used – if any – is quite involved. The card provides the terminal with its CVM List, that specifies under which conditions which CVMs are acceptable. The CVM List starts with two values that can be used to specify thresholds for transaction amounts. These values are followed by the *CV Rules*, a list specifying which CVM to use under which conditions. These rules can refer to the first two values in the CVM List and are in order of decreasing preference. Examples of possible conditions are 'If terminal supports the CVM' and 'If unattended cash'. The terminal chooses the CVM to be used based on the transaction details, its own capabilities and the card's preferences. In all Dutch bank cards we inspected, the CVM List is included in the Static Data to be Authenticated.

If cardholder verification is done by means of a PIN, there are three options:

- online PIN, in which case the bank checks the PIN;
- offline plaintext PIN, in which case the card checks the PIN, and the PIN is transmitted to the card in the clear;

T → C: `VERIFY`, pin
C → T: `Success` / (`PIN_failed`, tries_left) / `Failed`

Figure 3.4: Protocol step for offline plaintext PIN verification

T → C: `GET CHALLENGE`
C → T: ICC Dynamic Number
T → C: `VERIFY`, $enc_{PK_{ICC}}((pin, ICCDynamicNumber, random_padding))$
C → T: `Success` / (`PIN_failed`, tries_left) / `Failed`

Figure 3.5: Protocol steps for offline enciphered PIN verification

- offline enciphered PIN, in which case the card checks the PIN, and the PIN is encrypted before it is sent to the card.

Note that the card is only involved in cardholder verification in case of offline – plaintext or encrypted – PIN, as detailed below. Encrypting the PIN requires a card that supports asymmetric crypto. Online PIN verification is performed using a different protocol between the terminal and the bank. In this case the card is not involved in the cardholder verification.

#### Offline plaintext PIN verification

With plaintext PIN verification the PIN code is sent in plain to the card as can be seen in Figure 3.4. In its turn, the card will return an unauthenticated response indicating whether the PIN was correct or how many failed PIN attempts there are left before the card blocks.

#### Offline enciphered PIN verification

If the verification is done using the encrypted PIN, the terminal first requests a nonce from the card. Using the public key of the card, the terminal encrypts the PIN together with the nonce and some random padding created by the terminal. The result of the verification is then returned unauthenticated to the terminal as with the plaintext PIN verification. In Figure 3.5 the protocol steps for this method are given.

### 3.2.4 The transaction

After the optional card authentication and card holder verification, the actual transaction is performed. Transactions can be either offline or online. The terminal chooses which it wants to use, but the card may refuse to do a transaction offline and force the terminal to do an online transaction instead.

For a transaction the card generates one or two cryptograms: one in the case of an offline transaction, and two in the case of an online transaction.

- In an offline transaction the card provides a proof to the terminal that a transaction took place by means of a *Transaction Certificate (TC)*, which the terminal sends to the issuer later.
- In an online transaction the card first provides an *Authorisation Request Cryptogram (ARQC)* which the terminal forwards to the issuer for approval. If the card receives approval, the card then provides a TC as proof that the transaction has been completed.

In both on- and offline transactions the card can also choose to refuse or abort the transaction, in which case an *Application Authentication Cryptogram (AAC)* is provided instead of a TC or ARQC.

Below we first discuss how exceptions that occur can influence the type of cryptogram that is requested and the different types of cryptograms in more detail, before we describe the protocol steps for off- and online transactions.

#### Terminal action analysis

During the terminal action analysis, the terminal decided whether a transaction should be performed online or be aborted in case of the occurrence of an exception. When an exception occurs at the side of the terminal, such as, for example, a failed data authentication or unsuccessful cardholder verification, this is stored in the Terminal Verification Results (TVR). The TVR is a bit-string, where every exception corresponds to a particular bit. When an exception occurs the corresponding bit in the TVR is set to 1 by the terminal. To specify what the terminal should do in case of exceptions EMV makes use of so-called Action Codes. An Action Code is a bit-string similar to the TVR. Both the issuer and the acquirer can indicate their preferred actions specified in the Issuer Action Codes (IACs) and Terminal Action Codes (TACs) respectively. We can distinguish three different types of action codes: Denial, Online and Default. If no TACs are present, they have a default value of all bits set to 0. The default value for *IAC - Denial* is all 0s, whereas the *IAC - Online* and *IAC - Default* are all 1s by default. The Action Codes are processed in pairs by the terminal in the following order:

- *Denial*, if an exception occurred and the corresponding bit is 1 in either the *TAC - Denial* or *IAC - Denial*, the terminal will abort the transaction by requesting an AAC.
- *Online*, if the terminal supports online transaction and an exception occurred for which the corresponding bit is 1 in either the *TAC - Online* or *IAC - Online*, the terminal will force the transaction to be performed online by requesting an ARQC.
- *Default*, if the terminal does not support online transaction or it was unable to go online, and an exception occurs for which the corresponding bit is 1 in either

the *TAC - Default* or *IAC - Default*, the transaction will be aborted by the terminal by requesting an AAC. If the transaction is not aborted, the terminal will try to complete an offline transaction by requesting a TC.

### Cryptograms

Using the `GENERATE AC` command, the terminal can ask the card to compute one of the types of cryptograms mentioned above, i.e. TC, ARQC or AAC.

Arguments of the `GENERATE AC` command tell the card which type of cryptograms to produce and whether CDA has to be used. Additional arguments that have to be supplied by the terminal are specified by CDOL1 and CDOL2. CDA is only performed on TC or ARQC messages, and not on AAC messages.

The response always contains

- the Cryptogram Information Data (CID), indicating the Application Cryptogram (AC) type in the response,
- the Application Transaction Counter (ATC) and
- an Application Cryptogram (AC) or proprietary cryptogram.

Optionally, the response may contain

- the Issuer Application Data (IAD),
- other proprietary data,
- the Signed Dynamic Application Data (SDAD), namely if CDA is requested and the type of the response cryptogram is not AAC.

The cryptogram returned by the card can either be in the format specified in the EMV standard, or in a proprietary format. Additionally, if CDA is used, the card also returns the SDAD over the response using its private key $SK_{ICC}$ so that the terminal can check the authenticity of the complete message. If no CDA is performed, the response to a `GENERATE AC` command consists of the CID, the ATC, the AC and optionally the IAD. When performing CDA, the AC is replaced by the SDAD.

Both the CDOL1 and CDOL2 are required to always include the UN. A Card Risk Management Data Object List (CDOL) might request a *Transaction Certificate Hash*, which is a hash on the elements in the *Transaction Certificate Data Object List (TDOL)*. The TDOL might be provided by the card, or a default can be used that is specified by the payment system.

The `GENERATE AC` command starts with a parameter indicating the type of AC that is requested. The boolean parameter cda_requested specifies whether a CDA signature is requested. This results in the protocol step given in Figure 3.6. For an offline transaction this is the only step to confirm the transaction by requesting a TC.

T → C: `GENERATE AC`, (ac_type, cda_requested, data specified by the CDOL1)
C → T: (CID, ATC, AC, [IAD])

Figure 3.6: Protocol step for a single `GENERATE AC` command

T → C: `GENERATE AC`, (ARQC, cda_requested, data specified by the CDOL1)
C → T: (CID, ATC, AC, [IAD])
T: Communication with issuer to retrieve issuer_authentication_data
T → C: `EXTERNAL AUTHENTICATE`, (issuer_authentication_data)
C → T: `Success`/ `Failed`
T → C: `GENERATE AC`, (TC, cda_requested, data specified by the CDOL2)
C → T: (CID, ATC, AC, [IAD])

Figure 3.7: Protocol steps for an online transaction using the `EXTERNAL AUTHENTICATE` command

After forwarding the ARQC in an online transaction, the terminal might receive the *Issuer Authentication Data* in response from the issuer. If supported by the card, this data is sent to the card using the `EXTERNAL AUTHENTICATE` command to which the card responds whether the issuer authentication was successful (see Figure 3.7). If the `EXTERNAL AUTHENTICATE` command is not supported by the card, the Issuer Authentication Data can still be included in the CDOL2 for the card to authenticate the issuer. How the issuer is exactly authenticated is out of scope of the EMV specifications. After the card returns a TC the transaction is confirmed (and the money or goods handed over to the customer), even though the TC might not have been forwarded to the bank yet.

### 3.2.5 Script processing

A terminal might receive one or more Issuer Scripts from the issuer. This is a list of APDUs, called Issuer Script Commands, to be sent to the card. A script can specify whether this should be done either before or after the final `GENERATE AC` command. These commands are used to perform operations on the card that are not provided by the EMV specifications, for example, to update the PIN code on the card. For security sensitive operations, the EMV specification gives several ways to perform Secure Messaging.

## 3.3 Known attacks

EMV offers a lot of options which gives it great flexibility and introduces a lot of potential security problems at the same time. Some weaknesses of certain EMV configurations are widely known and apparently accepted – at least by some issuers. Next, the currently known attacks will be discussed.

### 3.3.1 Relay attack

With a relay attack, communication between a card and terminal is –as the name already suggests– relayed by the attacker. A terminal, controlled by the attacker, talks to the genuine card and relays the communication to an emulator card that is inserted into a genuine terminal. This means the genuine card and terminal are not necessarily at the same place when a transaction is performed. A customer could, for example, think he is paying for a small item in an attacker's grocery store, while the POS actually relays the communication with the card to a jewellery shop where an accomplice buys some expensive jewellery [DM07]. This attack can be used on almost all smart card based systems, as you basically only 'extend' the connection between the card and terminal. For contact based smart cards you still have to wait for the user to insert his card in a compromised terminal that can relay the communication. For contactless smart cards this becomes easier as you only need to get a reader close to a card to be able to communicate with it. These readers are becoming more common as they are included in many smartphones today. To counter a relay attack using cheap hardware, such as smartphones, we propose a simple modification to the EMV protocol used in these cards in [CGR+15]. This modification relies on the fact that a relay introduces some noticeable overhead in the timings of the communication.

### 3.3.2 Cloning SDA cards

With SDA, only static data on the card is authenticated. This means a terminal won't be able to determine whether the card is actually present or the data is replayed. It is thus possible to copy the data from a genuine card and create a clone. Since the cryptograms are generated using a secret key that is only known to the card and the issuer, the terminal won't be able to determine whether a transaction was successful before it sends the cryptogram to the bank. A cloned SDA card can therefore be used to perform offline transaction. However, as soon as the merchant tries to get his money by sending the cryptogram to the issuer, the issuer will deny the transaction as the cryptogram is not valid.

### 3.3.3 DDA wedge attack

DDA authenticates the card using a challenge-response mechanism as opposed to only static data as with SDA. However, this does not tie the card to the subsequent transaction. After the authentication of the card, an attacker could thus take over the communication and send fake cryptograms. If the transaction is performed offline, the terminal is not able to verify correctness of the cryptograms. For this attack, access to a real card is necessary to successfully complete the card authentication phase. By faking the cryptogram, the merchant will not be able to get his money, as with the cloned SDA cards.

### 3.3.4 No-PIN attack

In [MDAB10], Murdoch et al. describe an attack that works with online transactions. They present a MitM attack that makes use of the fact that the response to the `VERIFY` command for offline verification of the PIN is not authenticated. The attacker will let the transaction proceed as usual by forwarding all commands and responses between the terminal and card except when it sees a `VERIFY` command. In this case, he will not forward the command to the card but respond by sending 9000 as a response, indicating verification of the PIN code was successful. Even though the issuer and the card know that no actual PIN verification took place, the terminal has no way to detect this using the data that is provided in compliance with the EMV specification. The terminal will therefore conclude the verification was successful and will print 'Verified by PIN' on the paper receipt printed by the POS. This attack can be prevented in online transactions. For example, Dutch bank cards include the CVM Results in the ARQC that is forwarded to the bank. Should the issuer detect that no PIN code was entered at all they can decide to abort the transaction.

### 3.3.5 Fallback to plaintext PIN

Barisani et al. presented a MitM attack to force a fallback from encrypted PIN to plaintext PIN, making it possible for an attacker intercepting the traffic to learn the PIN code [BBLF11]. Their attack makes use of the way Issuer Action Codes are used by the terminal. By modifying the CVM List that is sent by the card, they change the preferred CVM to plaintext PIN. If the CVM List is part of the Static Data to be Authenticated, this can be detected by the terminal during the data authentication. However, one of the exceptions that is listed in the Action Codes is whether data authentication failed. This means that the terminal decides what to do after the data authentication failed based on the Terminal Action Codes and Issuer Action Codes. The attacker also changes the IACs that are sent by the card to not deny a transaction if data authentication failed, but to request an online transaction. As the terminal has no way to authenticate the IACs, it will use those provided by the attacker. If failed data authentication is not set in the TACs Denial, the terminal will perform the transaction online and continue with the data modified by the attacker. The terminal will therefore send the PIN in plaintext over the line to the card. Through the CVM Results the issuer will know that the PIN was sent in the clear, but only after the attacker already learnt it.

The fact that the terminal will still continue the transaction after the data authentication failed is a bit weird, even though the transaction needs to be performed online in this case. Apparently availability is considered to be more important than security in this case.

We repeated the attack and used the SmartLogic tool to intercept and modify the communication between a POS terminal in a shop in the Netherlands and a Dutch bank card [KGR12]. After the presentation by Barisani et al. the Dutch banks rolled

out a fix to their terminals to prevent the attack. Our test, however, revealed that even after the Dutch banks took countermeasures, there were still terminals that were susceptible to this attack. Although the transaction was denied by the back-end and the attack was quickly detected by the bank, the modification of the data still resulted in a plaintext PIN code transmission to the card.

### 3.3.6 Pre-play attack

In [BCM+14], Bond et al. show what can happen if the Unpredictable Number generated by the terminal is not really unpredictable. The UN is the only data field that provides freshness for a transaction from the side of the terminal as the other data fields, such as the transaction amount or country code, could be determined in advance. In their research they found several ATMs for which the UN was either completely or partially predictable because, for example, it was based on a counter or timestamps. Knowing the UN enables an attacker to harvest valid ARQCs or TCs, depending on whether the transaction is online or offline respectively. For example, a malicious POS could harvest a number of ARQCs from a card that is used to pay in a shop. Later the attacker can use the harvested data in an Automatic Teller Machine (ATM) for which he can predict the UN to successfully withdraw money. He won't be able to generate a valid TC as this is based on the issuer's response to the ARQC, which cannot as easily be predicted. This might not be a problem as the terminal cannot verify the TC itself and it could be the case that the TC is not immediately sent to the issuer for verification. If CDA is used, this attack would not work for online transactions as the attacker would need a signature on the TC, which would be checked by the terminal. If the customer uses his card before the attacker tries to withdraw money using the customer's data, the last used ATC that is known by the issuer will be higher than the ones in the ARQCs that the attacker harvested. If the attacker now tries to use the harvested data, the issuer is able to detect that data is replayed, as the ATC only increases. Also, depending on the number of ARQCs the attacker harvests, a big gap between the last successfully used ATC and the current one is used. This could be an indication to the issuer that there is a problem, but it can also be caused by failed transactions.

In a Specification Bulletin, EMVCo announced that additional requirements for the UN would be introduced [EMV12]. Terminal vendors are required to produce UNs that are actually unpredictable. One suggestion for this is to follow international standards for random number generation.

# Chapter 4

# Analysing EMV

As seen in the previous chapter, EMV is a complex standard with many options. In this chapter we present our formal analysis of these specifications in Section 4.1 as published in [RP12]. To analyse implementations of EMV we used automated learning techniques on real bank cards, as discussed in Section 4.2 and published in [ARP13].

## 4.1 Formal analysis

In this section we will discuss the formal analysis we performed on the EMV protocol suite [RP12]. For the analysis we make use of the ProVerif tool discussed in Section 2.2.2. Due to the complexity of the EMV specification, we wrote the model in F# and used the tool FS2PV (see Section 2.2.3) to translate our F# model to the untyped pi-calculus that is used by ProVerif. The model grows substantially in size with this translation: whereas the F# model is 370 lines of code, the resulting ProVerif model is 2527 lines of code. The increase in size is caused by the many if-statements in the F# model – which result in duplication of large fragments of code in the untyped pi-calculus – and the (convenient) use of functions in F#. In fact, initially we tried to formalise the EMV protocol in the untyped pi-calculus, but we gave up as the model became too complex to oversee.

Our model includes a card and a terminal. The issuer is not considered in this model as it is not needed for the security requirements we consider in Section 4.1.4. The F# code models a card that performs a single transaction. For most security properties we want to consider cards performing multiple transactions. For this we have to edit the generated ProVerif code, by simply adding `!` for replication in the right place.

We assume that the channel between the card and terminal is public, i.e. a Man-in-the-Middle can intercept and modify all communication. Since the cards are provided by the bank, we assume that there are no dishonest cards. The terminal is assumed to have secure input and output with the customer. Our model makes some assumptions on DOLs based on what we observed in Dutch bank and credit cards. Since we do not know how the cryptogram is computed on the Dutch bank cards, we follow the recommendations from the EMV specification. Here a MAC is computed using the symmetric key $SK_{SAC}$, which is shared between the card and the issuer, on a minimum set of recommended data elements. The recommended minimum set of

elements to be included in the AC is specified in [EMV11b, Section 8.1.1]. It consists of the amount, terminal country, terminal verification results, currency, date, transaction type, terminal nonce, AIP and ATC. The AIP and ATC are data provided by the card, the other elements are provided by the terminal in the CDOL1 and CDOL2.

### 4.1.1 Card and terminal configuration options

For both card and terminal the model includes a number of boolean parameters that describe their configuration parameters. For the card these parameters include

- `sda`, `dda`, `cda`: three booleans that define which card authentication mechanisms are supported;
- `force_online`: a boolean that determines whether the card will force the transaction to go online if the terminal starts an offline transaction.

For the terminal these parameters include

- `pin_enabled`: can the terminal check PIN codes?
- `online_enabled`: is the transaction forced online by the terminal?

When analysing the model we have the choice between fixing certain values of these parameters, or leaving them open. Advantage of the second approach is that properties of multiple configurations can be verified in one go. Disadvantage is that the model may be too complicated for ProVerif to verify (within a reasonable response time), in which case some of these parameters should be fixed. For a bank issuing a specific type of card, it would be fine to fix all the parameters for the card; still, one should then consider all the possible terminal behaviours this card might encounter.

In our model, after the data has been read, the terminal optionally performs card authentication and cardholder verification, and then, for a fresh card-generated nonce and new value of the transaction counter, it provides at most two cryptograms as requested by the terminal and described in Section 3.2.4: either just a TC or AAC, or an ARQC followed by a TC, or an ARQC followed by an AAC.

### 4.1.2 DOLs

Although our model leaves many configuration options for card and terminal open, as described above, the values of the DOLs have to be fixed and hard-coded in the model. Given that the DOLs determine which data is signed or MACed in various protocol steps, we cannot expect to do any security analysis without at least making some minimal assumptions.

The DOLs in our model are based on what we observed on Dutch bank and credit cards:

- PDOL: the empty list
- DDOL: (Unpredictable Number)
- CDOL1: (amount, CVM Results, Unpredictable Number), where the CVM Results contains the result of the cardholder verification.
- CDOL2: (Unpredictable Number)
- TDOL: not used

These are not the precise DOLs of the bank cards we looked at, but rather subsets (or sub-lists) of them. For readability we omitted some of the data elements. For the properties we wanted to verify, omitting these data elements is safe: if with the DOLs above it is impossible to fake messages, then for DOLs with more data elements – which would result in the inclusion of *more* data elements in digital signatures, (signed) hashes or MACs – it is still impossible to fake messages.

### 4.1.3 Adding events to express security requirements

To express interesting security properties, our formal model is augmented with *events* that mark important steps in the protocol by different participants. Without these, all that can be verified are confidentiality properties, as these are 'built-in' to ProVerif.

Events added to the model are

- `CardVerifyPIN(success)`: a failed or successful verification of the PIN code by the card.
- `TerminalVerifyPIN(success)`: a failed or successful verification of the PIN code by the terminal.
- `CardTransactionInit(atc,sda,dda,cda,pan)`: the card starting an EMV session.
- `TerminalSDA(success, pan)`, `TerminalDDA(success, pan)`, `TerminalCDA(success, atc, ac_type)` and `TerminalCDA2(success,atc,ac_type)`: a failed or successful data authentication by the terminal using SDA, DDA or CDA respectively.
- `TerminalTransactionFinish(sda,dda,cda,pan,atc,success)`: the terminal completing a transaction.
- `CardTransactionFinish(sda,dda,cda,pan,atc,success)`: similarly, the card completing a transaction.

Here `success`, `sda`, `dda`, `cda`, are boolean parameters to indicate success or failure and describe the card authentication mechanism used, while `atc` and `pan` are integer parameters for the card's Application Transaction Counter and Primay Account Number.

### 4.1.4 Security requirements

For our model we have verified three types of properties: sanity checks, secrecy requirements, and, most interestingly of all, integrity and authenticity requirements.

#### Sanity checks

A silly mistake in the formal model could cause a deadlock, in some or all the branches of the protocol, preventing these branches from ever being completed, and then making some security properties for these branches trivially true. To detect this, we have checked that all events are triggered, so it is possible to reach all events and hence perform all possible variations of the protocol.

#### Confidentiality

The confidentiality requirements for EMV are the usual ones, namely confidentiality of the private asymmetric keys and the shared symmetric keys.

#### Integrity and authenticity

**Card authentication** If the terminal successfully performs a card authentication, it should be the highest card authentication method supported by both the card and the terminal to rule out a forced fall-back, e.g. from DDA to SDA. To check this, we verified the following two queries:

```
ev:TerminalSDA(True(),pan) ==>
ev:CardTransactionInit(atc,True(),False(),False(),pan)
```

and

```
ev:TerminalDDA(True(),pan) ==>
ev:CardTransactionInit(atc,sda,True(),False(),pan)
```

The PAN is unique for each card, so these queries express that if a terminal successfully performs SDA or DDA this was the highest supported card authentication method. Both queries are evaluated to true by ProVerif for multiple transactions per card, so no fallback can be forced.

The particular EMV configuration that we model, based on our observations of Dutch bank cards, is of importance here. The only reason that rollbacks of the card authentication method are not possible is that the AIP, which contains the data authentication methods that are supported by the card, is included in the data that is authenticated as part of SDA, DDA and CDA.[1]

[1]Not including this AIP in the data that is authenticated is allowed by the EMV specifications, but obviously a bad choice and one that the specification could warn against, or even disallow.

To prevent replay, we also want the card to participate in each data authentication. This is checked by using so-called injectivity in the queries:

```
evinj:TerminalSDA(True(),pan) ==>
evinj:CardTransactionInit(atc,sda,dda,cda,pan)
```

and

```
evinj:TerminalDDA(True(),pan) ==>
evinj:CardTransactionInit(atc,sda,dda,cda,pan)
```

These queries state that if a terminal completes an authentication, the corresponding card (with that PAN) is in fact involved.

ProVerif was not able to prove the first query when considering multiple sessions per card. However, with a single transaction per card the query evaluates to false. If this query does not hold with a single session per card, it will also not hold with multiple sessions per card. This result is as expected, as SDA allows replays as discussed in Section 3.3.2. The second query could again be proved for multiple sessions per card. This query evaluated to true, as the challenge-response mechanism used in DDA prevents replays.

**Customer authentication** If the customer is authenticated using his PIN code, the terminal and card should agree on whether the PIN was accepted or not. To check this the following query is used:

```
evinj:TerminalVerifyPIN(True()) ==> evinj:CardVerifyPIN(True())
```

ProVerif indicates that this query is false. The root cause is that the response of the card to an attempted (offline) PIN verification is not authenticated, so a Man-in-the-Middle attack could fake it. This weakness is exploited in the attack that is explained in Section 3.3.4.

**Transaction authenticity** If a transaction is successfully completed by the terminal, the corresponding card should also agree on having completed the transaction successfully. This is checked using the following query:

```
evinj:TerminalTransactionFinish(sda,dda,cda,pan,atc,True())
==>
evinj:CardTransactionFinish(sda2,dda2,cda2,pan,atc,True())
```

Not surprisingly, this query evaluates to false, as both SDA and DDA do not authenticate the transaction in any way that the terminal can verify (see the attacks in Section 3.3.2 and Section 3.3.3).

When using CDA, the card and terminal should agree on the result of the transaction, i.e. if CDA is successfully performed and the transaction is concluded with a TC message at the terminal side, the card should also successfully have completed the transaction. This is checked using the following queries:

```
evinj:TerminalCDA(True(),atc,DataTC()) ==>
evinj:CardTransactionFinish(sda,dda,cda,pan,atc,True())
```

and

```
evinj:TerminalCDA2(True(),atc,DataTC()) ==>
evinj:CardTransactionFinish(sda,dda,cda,pan,atc,True())
```

For both queries the result is true for cards with multiple transactions. Using CDA thus guarantees that both the card and terminal agree on successful transactions.

### 4.1.5 Experiences using ProVerif

We ran ProVerif on a machine with an Intel Core i5-M540 processor at 2.53 GHz and 4GB of RAM memory. The running times for the final model range from 5 seconds for the sanity checks to around 5 minutes for the more complex queries.

When constructing the model, usually ProVerif could verify the properties we were interested in in a few minutes. Occasionally, it would take hours. To reduce the time needed to verify, we removed types from the functions used and tried to reduce the number of if-statements. Small changes in the F# model could result in quite different ProVerif code. For example using a boolean condition $b$ in an if-statement resulted at one point in ProVerif not being able to compute the result within hours. Changing the condition to $b = true$ resulted in code being verified by ProVerif within minutes.

### 4.1.6 Including the issuer

When extending the model with the issuer we can check not only whether the card and terminal agree on the details of the transaction, but also whether the issuer agrees. This is for example interesting for online transactions, where the terminal relies on the response by the issuer in accepting or denying the transaction. In an extension of the previously discussed model we included the issuer in the code for the terminal, as we assume the communication between the terminal and the issuer to be secure. Since we did not know how the MACs in the cryptograms are actually computed we used the recommended minimum set of elements to be included as specified in the EMV standard [EMV11b, Section 8.1.1]. These included elements are amount, terminal country, Terminal Verification Results, currency, date, transaction type, terminal generated nonce, Application Interchange Profile and Application Transaction Counter. Notice that the type of the cryptogram is not included in this set of data

elements. This resulted in a new weakness found by ProVerif in our model. If a transaction is declined by the card by sending an AAC to the terminal, a MitM could change the type of the message to a TC. This would result in the terminal and issuer accepting the transaction, though the card denied the transaction. This attack does not work if CDA is used, as changing the message type would invalidate the signature on the message. This is a theoretical weakness, as we do not know how the banks implemented the actual construction of the cryptograms.

### 4.1.7 Conclusions

Our model covers all the important options for card and terminal, including all card authentication methods (SDA, DDA, CDA) and transaction types (on- and offline). For some of the configuration parameters, the so-called DOLs, minimal assumptions are hard-coded in the model; these can easily be changed, but analysis of EMV without making assumptions about the DOLs is clearly impossible.

Given the size complexity of the EMV specifications, the formal model is surprisingly small. The model still fits on 5 pages. The use of F# as modelling language was crucial to keep the formalisation tractable. Our initial attempts to model EMV in untyped pi-calculus failed, and the use of if-statements and function in F# were a huge improvement to keep the model comprehensible. Indeed, whereas the F# is 370 lines, the generated ProVerif code by FS2PV is 2527 lines. Admittedly, a handwritten ProVerif model might be smaller, but this comes at the cost of readability.

Of course, our model abstracts from some of the low-level details that are in the 700-odd pages EMV specs, e.g. about byte- and bit-level encodings of data. Such abstraction seems crucial to keep an overview and understand the standard as a whole. The EMV specs make very little attempt at providing useful levels of abstractions, apart from use of a standard Tag-Length-Value (TLV) encoding.

We had to come up with the security requirements to verify ourselves, as these are at best very implicit in the official specifications. We expected this might be hard, but the rather generic security requirement – that after a transaction all parties agree on all that transaction's parameters – captures the essential security requirement in an intuitive and easy way.

## 4.2 Learning models of bank cards

In this section we describe our work on the automated reverse engineering of EMV bank cards using the model inference techniques explained in Section 2.3 [ARP13]. Such automated reverse-engineering, which only observes the smart card as a black box, takes little effort and is fast. The finite state machine models obtained provide a useful insight into decisions (or indeed mistakes) made in the design and implementation, and would be useful as part of security evaluations – not just for bank cards

but for smart card applications in general – as they can show unexpected additional functionality that is easily missed in conformance tests.

Software for bank or credit cards will be developed using a very strict and regimented software engineering process. After all, this software is highly security-critical and patching is usually not an option. The software will be subjected to rigorous compliance tests and security certifications, possibly even costly Common Criteria certifications.

Establishing security here is often more difficult than just establishing correctness, or compliance with a standard. In checking compliance (e.g. for interoperability) the emphasis tends to be on the *presence of required functionality*: if some functionality is missing, the implementation is incorrect and it will not work correctly in all circumstances. Security on the other hand is also concerned with the *absence of unwanted functionality*; if an implementation provides more functionality than what is required, then it may be considered compliant – after all, it does what it is supposed to do – but it might be insecure, as it does *more* than what it is supposed to do, and this additional functionality may be a source of insecurity. This makes it hard to test for security bugs, and to discover them in the field: unlike functional bugs, security bugs may never show up under normal circumstances.

Testing of security applications using model-based testing techniques seems an interesting approach to test for security vulnerabilities [FAZB11], as a generalisation of fuzzing. It does however require formal models that specify the intended behaviour of the system, and in practice these are often not available, because creating them is time-consuming, and possibly complex and error-prone. Constructing these models automatically would therefore be extremely useful. A potential approach is to use program analysis to construct models from source code. Of course, in many cases, access to source code is restricted. The method discussed in this section constructs models just by observing the smart card's external behaviour.

For our tests on EMV bank cards we used a collection of MasterCard and Visa branded debit and credit cards from the Netherlands, Germany, Sweden and the UK. All the MasterCard credit cards contain a MasterCard application, whereas on the bank cards there is a Maestro application. Both these applications are used for payments in shops and to withdraw cash from ATMs. The Dutch bank cards also contain a SecureCode Aut application, which is used for online banking with a handheld EMV-CAP reader provided by the bank (see Section 5.1). The Visa branded debit card contains the Visa Debit application.

We used authentic bank cards as *SUT*/*Teacher*. Access to the smart cards was realised via a standard smart card reader and a testing harness, which we discuss in Section 4.2.1. We connected the *SUT* to the LearnLib library (see Section 2.3.3) which served as *Learner*, see Figure 4.1.

In our experiments we used a random walk method with 1000 test traces of length 10 to 50 as equivalence oracle. We verified our results with the W-method by Chow [Cho78] to check if it will find at least one more state than the random test suite.

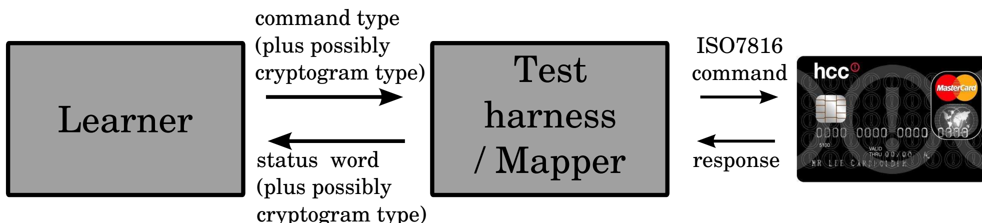


Figure 4.1: Set-up

### 4.2.1 Test harness

As illustrated in Figure 4.1, our test harness translates the abstract command (from the input alphabet of our Mealy machine model) to a concrete Command APDU, and translates a Response APDU to a more abstract response (in the output alphabet of the Mealy machine model). The following commands are supported by the test harness:

- `SELECT APPLICATION`
- `GET PROCESSING OPTIONS`
- `READ RECORD`
- `GET DATA`
- `INTERNAL AUTHENTICATE`
- `GET CHALLENGE`
- `VERIFY`
- `GENERATE AC`

The test harness is just over 300 lines of Java code. Most of this code is generic code to set up a connection to the smart card reader. A regular smart card reader was used, and communication was performed using the standard Java Smart Card I/O library. The code specific to EMV is just over 100 lines of code, and consists of 15 methods that define a particular Command APDU to be sent to the card. The input alphabet corresponds to these 15 methods.

For many parameters of these commands the test harness uses some fixed value, for instance for the random number sent as argument of the command `INTERNAL AUTHENTICATE`, the payload data for the cryptograms generated by the card, and the (correct) guess for the PIN code. One would not expect a different random number to affect the control flow of the application in any meaningful way, so by fixing values here we are unlikely to miss interesting behaviour. Note that we have two different

payloads when requesting the cryptograms due to the difference between the first and second request for a cryptogram. As these payloads are different, both a correct and an incorrect payload is used when requesting cryptograms. Obviously, entering an incorrect PIN code would affect the control flow, but learning about the behaviour in response to incorrect PIN guesses is very destructive as it will quickly block the card.

For several commands different variants are provided by the test harness:

- For the commands `GET DATA`, `READ RECORD` and `GET PROCESSING OPTIONS`, both a variant with correct arguments and one with incorrect arguments is provided. E.g., for `GET DATA` we have variants requesting a data element that is present or one that is not.
- For the `GENERATE AC` command 6 variants are provided, as there are 3 cryptogram types, each of which can be used with one of two sequences of arguments (one for the first and one for the second cryptogram).

The test harness does not output the entire response of the smart card to the learner. It only returns the 2 byte status word, but not any additional data returned by the card. For most commands, like `GET PROCESSING OPTIONS`, this additional data returned will always be the same, so there is not much interest in learning it. The only exception to this is the `GENERATE AC` command: here the test harness does return the type of cryptogram that was returned by the card (but not the cryptogram itself; as this is computed using a cryptographic function on the input and the card's ATC, the response will never be the same and there is nothing we could hope to learn from it).

A limitation of our test harness is that we do not know the bank's secret cryptographic keys that might be needed to complete one 'correct' path of the protocol, namely the path where the card produces an ARQC as first cryptogram and a TC as second. For this a correct reply to the first ARQC is needed, which requires knowledge of the cryptographic keys used by the bank's back end.

To be able to include the `VERIFY` command in the learning, the PIN code of the corresponding card has to be known. We did not try to learn the behaviour of the card in response to incorrect PIN codes, to avoid blocking the card. The cards we used are real bank cards for which we cannot reset the PIN. (With access to functionality to reset the PIN, which the issuing bank might have, one could also try to learn the behaviour in response to incorrect PINs.) The German card only supported encrypted PIN verification. Since the public key of MasterCard is needed to be able to encrypt the PIN and this public key is not published, we were unfortunately not able to use the `VERIFY` command with this card.

The Visa branded card can perform the `GET DATA` command to retrieve the current value of the ATC. This functionality is used for a so-called *mapper* component [AJU10, ASV10] to be able to learn the transitions where a counter is increased. The mapper is integrated in the test harness of the *SUT*, and keeps track of the value

of the counter. The `GET DATA` command only returns the current value of the ATC if the Visa Debit application is selected. Since the mapper depends on the value of the ATC, the Visa Debit application is automatically selected by the test harness before an output query is performed by LearnLib. The mapper retrieves the value of the ATC after each output query and adds the difference with the stored value of the ATC to the response, e.g. '+1' on an edge indicates the ATC was increased by one in this transition.

### 4.2.2 Trimming the inferred state diagrams

The state diagrams returned by LearnLib as .dot file look quite unintelligible at first sight, because there are so many transitions: for each state, one for *every* possible command. However, many transitions from a given state are errors and simply return to the same or an error state (e.g. the 'Selected' or 'Finished' state). By simply collapsing all these transitions into one transition marked 'Other', and drawing multiple transitions between the same states with different labels as one transition with a set of labels, we obtain simple automata such as figures 4.2, 4.3, 4.4 and 4.5. In these figures the responses are omitted for readability. We simply obtained these by manually editing the .dot files. This could easily be automated. At the same time we chose meaningful names for the different states.

The transition labels for `GENERATE AC` commands indicate (i) if it is the 1st or 2nd request for a cryptogram in this session (i.e. whether the argument is for the first or second request), (ii) the type of cryptogram that was requested (ARQC, AAC, or TC), and (iii) the type of cryptogram that was returned. E.g. `GENERATE AC 1st ARQC ARQC` means the type requested was ARQC, with the arguments supplied for the first request, and the type returned was an ARQC. We have combined arrows if different parameters yield the same response; e.g. `GENERATE AC 2nd TC/AAC AAC` means that requests for a TC or AAC, with the arguments for the second request, both result in an AAC.

### 4.2.3 Results

We learned models of EMV applications on bank cards issued by several Dutch banks (ABN AMRO, ING, Rabobank) and one German bank (Volksbank), on MasterCard credit cards issued by Dutch and Swedish banks (SEB, ABN AMRO, ING) and on one Visa debit card from the United Kingdom (Barclays). The Dutch bank cards contain two EMV applications, one for internet banking (SecureCode Aut) and one for ATMs and Point-of-Sales (Maestro). All cards resulted in different models, with as only exception that the Maestro applications on all Dutch bank cards were identical, as were the SecureCode Aut applications. An educated guess would be that these implementations come from the same vendor.

To learn the models LearnLib performed between 855 and 1695 membership queries for each card and produced models with 4 to 8 states. The total learning

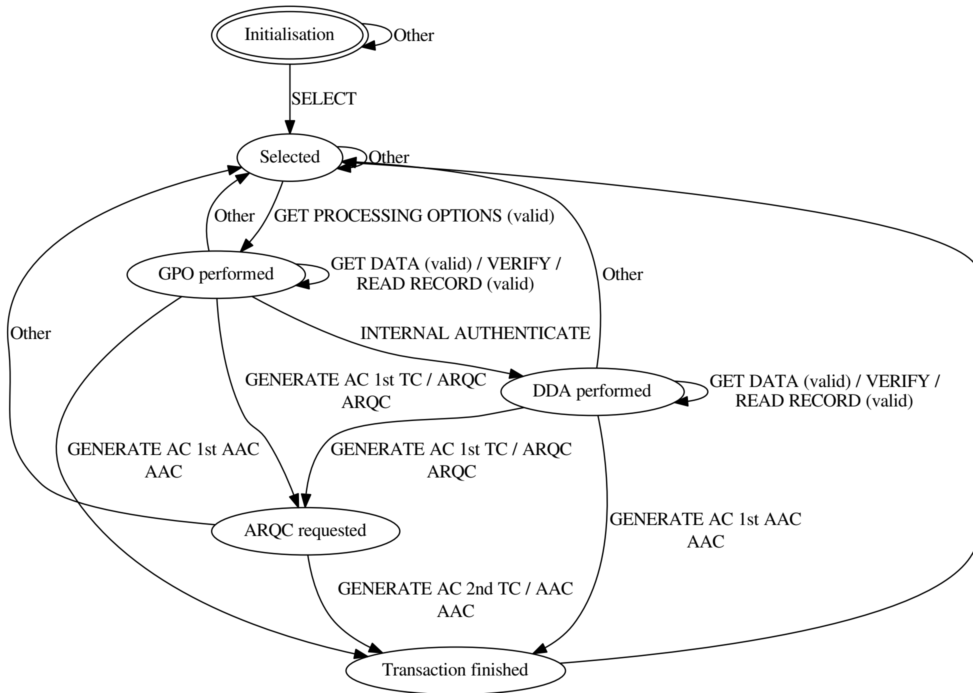


Figure 4.2: Automaton of Dutch Maestro application. Just to highlight one observation that can be made from this diagram: the `VERIFY` instruction, i.e. the verification of the PIN code by the smart card, is optional; this makes sense because the terminal may check the PIN code with the bank (so-called online PIN verification), or choose not to verify the PIN at all.

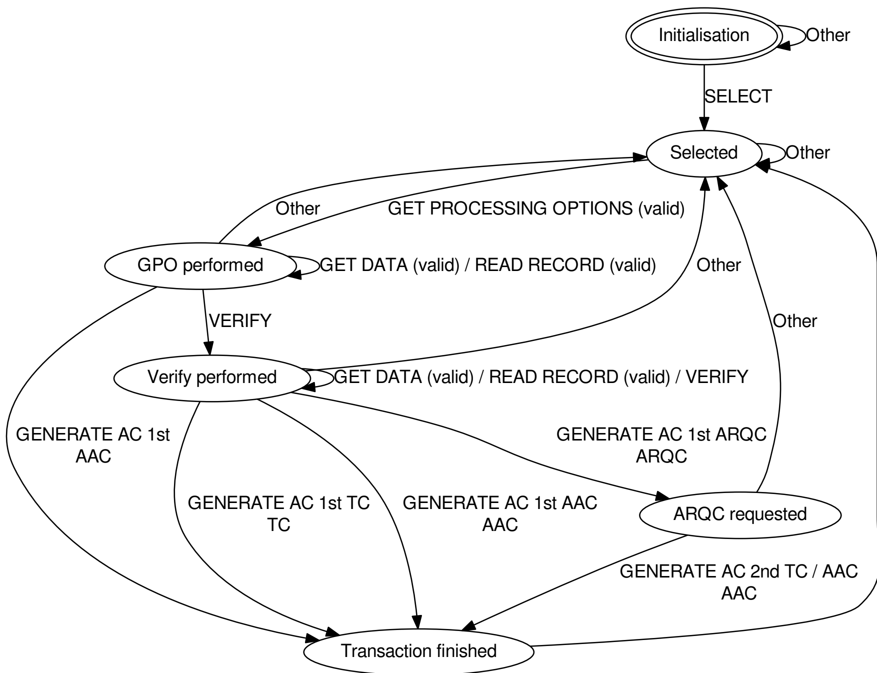


Figure 4.3: Automaton of Dutch SecureCode Aut application. Note that here the `VERIFY` operation – i.e. verification of the PIN code – must be passed successfully before cryptograms can be generated, except for the AAC cryptogram to abort the session.

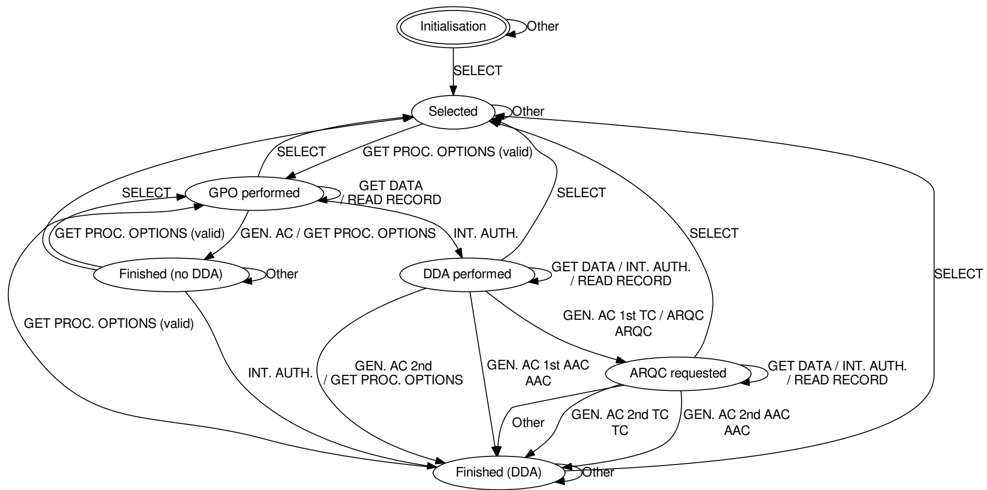


Figure 4.4: Automaton of Maestro application on a German Volksbank bank card. Note that it is very different from the Dutch Maestro card in Figure 4.2.

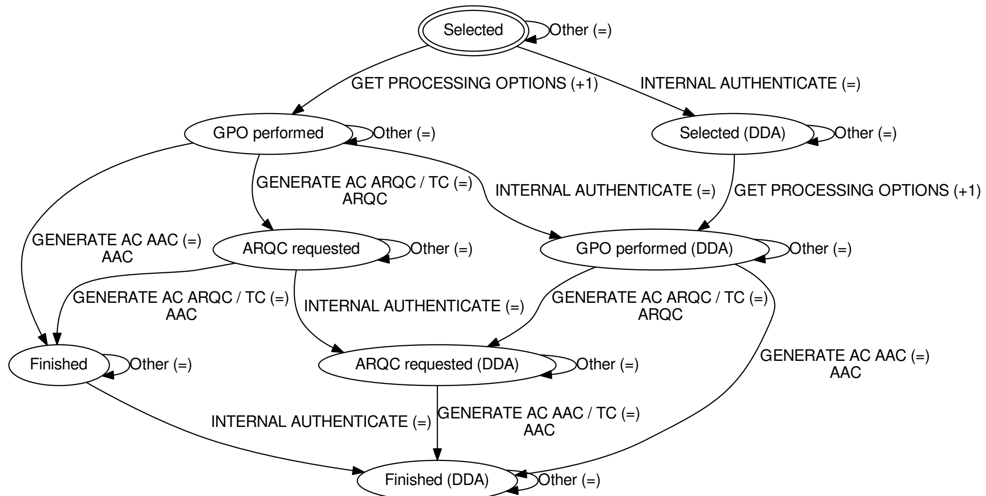


Figure 4.5: Automaton of Visa Debit application on Barclays card. Note that the `INTERNAL AUTHENTICATE` can be performed at any stage of the protocol.

time depended on the algorithm and corresponding parameters used for equivalence approximation. The time needed to construct the final hypothesis was less than 20 minutes for every card. Using the random walk method, it took between 9 and 35 minutes to generate a model. With the W-method by Chow it varied from 20 to 65 minutes. However, both algorithms led to the same models.

When analysing the state diagrams for the different categories, we made the following observations.

The state diagrams for the ABN AMRO and ING credit cards are very similar. There are only a few subtle differences, e.g in the initial state different error codes are returned in response to some instructions. Also the handling of the `INTERNAL AUTHENTICATE` instruction differs: both cards respond with the error `6D00` ('Instruction code not supported or invalid'), indicating that the instruction is not supported, but for the ING card this does not have any influence on the state, whereas the ABN AMRO card is 'reset' to the 'Selected' state.

Comparing the Maestro (Figure 4.2) and the SecureCode Aut application (Figure 4.3) on the Dutch bank cards, we can observe the following:

1. In both applications, if data that is not available is requested, either using the `READ RECORD` or the `GET DATA` instruction, the application returns to the 'Selected' state. This seems a bit strict, as the terminal has no way of knowing whether certain data that can be retrieved using `GET DATA` is available. Apparently, here the developers have chosen a 'safe by default' approach. Though this seems a sensible approach, one can imagine this can lead to compatibility problems with terminals that expect certain data to be present on the card while it is not, as the card will reset to a state that the terminal might not expect.

2. With the SecureCode Aut application it is possible, after successfully verifying the PIN code, to request a TC cryptogram using the `GENERATE AC` instruction. This is surprising, as this does not have any meaning in EMV-CAP: in an EMV-CAP session the terminal must always first ask for an ARQC. One would expect that requesting a TC cryptogram type would result in an error (as e.g. happens when a second ARQC is requested) or in an AAC being returned to abort the session (as e.g. happens when any type of cryptogram is requested before PIN verification). Still, it does not seem that this spurious TC cryptogram can be exploited to cause a security vulnerability, at least insofar as we know the EMV-CAP protocol, which is discussed in Section 5.1.

3. The error code that is given in response to the `INTERNAL AUTHENTICATE` instruction is different depending on the state in the SecureCode Aut application. In those states where it is possible in the Maestro application to perform this action, the error code is `6987` ('Expected secure messaging data objects missing'), while in the other states, an error code `6985` ('Usage conditions not satisfied') is returned.

Compared to the cards considered before, the Volksbank card handles things a bit differently (see Figure 4.4):

1. Where the other cards return to the 'Selected' state when an error occurs, the Volksbank card goes into a 'Finished' state. From a 'Finished' state there is one transition using the `SELECT APPLICATION` command to get to the 'Selected' again, and one to get to the 'GPO performed' state using a valid `GET PROCESSING OPTIONS` command.

2. Data authentication using DDA is also handled differently with this card. First, the card forces DDA to be performed, i.e. if no `INTERNAL AUTHENTICATE` command is given, transactions cannot be performed: the `GENERATE AC` command will then always return an error. Also, it is possible to perform DDA even if the card is in a 'Finished' state. This suggests that the `INTERNAL AUTHENTICATE` command is handled separately from the other commands and keeps its own state to indicate whether it is already performed. Below we compare this with what the MasterCard's specifications say.

3. If in the first `GENERATE AC` a TC is requested, the card indicates it wants to go online by returning an ARQC. However, after an ARQC is returned the first time, when requesting a TC in the second `GENERATE AC`, this is actually returned. This seems odd since one would expect this request to fail (i.e. an AAC to be returned), as we did not provide a valid response from the bank.

An interesting observation considering all cards is that the state diagrams can be used as fingerprints for the cards. Since the state diagrams are different for each type of card, and indeed each type of application on them, it is possible to determine a sequence of commands that gives a different result for each of the categories. As a matter of fact, just looking at the response to the `GET DATA` and `READ RECORD` instructions in the initial state one can already determine which application one is dealing with. However, using these fingerprints to identify cards is not so interesting for an attacker, as the cards already reveal plenty of uniquely identifying information. For some smart card applications, such as electronic passports, such fingerprints are unwanted though [RMP08].

### Difference with MasterCard's specifications

The Maestro and MasterCard-branded applications should all conform to MasterCard's Paypass-M/Chip specification[2]. This specification specifies a state diagram, which has only 5 states, whereas the models we obtained for Maestro cards have 6 or 7 states.

[2]This specification is for dual interface (contact and contactless) cards, rather than contact-only cards, but states that the state diagram for contact-only cards is the same, except that it has one transition less [Mas05, p. 98].

In the state diagram specified by MasterCard the operation `INTERNAL AUTHENTICATE` has no effect on the state, meaning that this operation – i.e. performing DDA – is optional and can be done *any number of times*. In contrast, the model learned for the Dutch Maestro card says that this operation can be done *at most once* before cryptograms can be generated, and the model for the Volksbank Maestro card says that it must be done *exactly once* before cryptograms can be generated.

Another difference between the state diagram of the Volksbank card and the one specified by MasterCard is the presence of the 'Finished (no DDA)' state, which seems to be a spurious dead-end in the behaviour of the Volksbank card, as it does not lead to a normal protocol run which ends where one or two cryptograms are generated.

As these cards carry the Maestro or MasterCard logo, they must have undergone some certification. Assuming that their certification has not missed potential compatibility problems caused by these deviations from MasterCard's specification, this suggests that this process does not include checking for implementation of the exact state machine.

**Different choices in the Visa branded card**

In the models of the MasterCard applications there exists an 'Initialisation' state from which the applications can be selected on the smart card. Since with the Visa branded card the test harness automatically selects the Visa Debit application, this initialisation state is not included in the learned models and the initial state is 'Selected'.

The Visa branded card is quite different from the others. For example, with the Visa card the commands `GET DATA`, `READ RECORD` and `VERIFY` are allowed in all states, even before the transaction is initialised with `GET PROCESSING OPTIONS` and after the actual transaction is started with a `GENERATE AC` command or even finished. These commands are thus apparently completely independent from the state of the card. Also, DDA can be performed, by an `INTERNAL AUTHENTICATE`, completely independent of any other actions, again even during and after a transaction.

In the model it can be seen from the additional information added by the mapper that only two transitions increase the ATC. This indicates that the ATC is increased when performing a successful `GET PROCESSING OPTIONS` command (i.e. `9000` is returned as the status word).

### 4.2.4 Conclusions

We have demonstrated that after defining a simple test harness/mapper component, we can easily obtain useful state machine models for banking smart cards using learning and simple abstraction techniques [RSBM09, AJU10]. After some trimming, the models obtained are easy to understand for anyone familiar with the EMV standard, and clearly highlight some of the central decisions taken in an implementation.

Differences in the models obtained for different cards may be inconsequential differences that exploit the implementation freedom allowed by the under-specification in the EMV specifications, but can really affect the security conditions imposed (for example, the difference between figures 4.2 and 4.3 in requiring PIN code verification). To determine which is which, we have relied on ad-hoc manual work and human intelligence - the models obtained are easy to inspect visually. This step could even be automated if security conditions are expressed as temporal logic formulae.

Although we get these formal models for free – i.e. without very little effort –, it does still require a human expert to see if the models obtained are correct and secure.

Differences in the state diagrams do not necessarily mean that implementations are not secure or that they cannot be regarded as compliant to the standard. The diagrams are a helpful aid in deciding whether this is the case. However, this decision then inevitably relies on an informal understanding of the standard and the essential security requirements. One would like to see more objective criteria for this, especially as security protocols are notoriously brittle and deciding what constitutes a secure refinement of the specification is not always easy.

The complexity of the standards involved makes such models very valuable. In fact, finite state machine models such as we obtain would be a useful addition to the official specifications. Despite the length of the EMV specifications [EMV08], state diagrams describing the smart card are conspicuously absent. A state diagram is specified in MasterCard's specification [Mas05], but most of the cards we analysed actually did not conform to it. The differences between e.g. figures 4.2 and 4.4 show the considerable leeway there is between different implementations of the same spec. One would expect (and hope?) that engineers developing, testing, or certifying EMV smart cards do have such state diagrams, either in the official documentation or just scribbled on a whiteboard.

The models learnt did not reveal any security issues. Indeed, one would not expect to find any in smart cards such as we considered, which should have undergone rigorous security evaluations and tests. Still, we do notice some peculiarities (notably that the Volksbank card is still willing to return a TC even after failed issuer authentication). We believe that our approach would be useful as part of security evaluations, because it increases the rigour and confidence provided and it can save a lot of expensive and boring manual labour.

Here it helps that LearnLib learns the behaviour blindly, in a completely haphazard way, without any of the preconceptions or expectations about what the 'normal' behaviour is that a human tester or code reviewer might have. The tool learns about *all* the possible behaviour. This is an advantage for security, as security bugs often occur under unusual conditions, when someone does something unexpected.

Still, the hand-coded test harness we developed does make some assumptions about the functionality that the card provides. The test harness implements the basic operations for EMV, and LearnLib then only learns all the possible behaviours given these operations. A deliberately introduced backdoor would thus not be detected,

but we conjecture that any mistake in the implementation of the internal state and the associated control flow in the smart card code would.

# Chapter 5

# Securing online transactions

For internet banking, many banks let their clients use a handheld smartcard reader with a small display and keypad. In combination with a smartcard and PIN code, this reader then signs challenges provided on the bank's web page, to log in or to confirm bank transfers. Many of these systems use a proprietary standard of MasterCard on top of the EMV standard, called the Chip Authentication Program, also known as EMV-CAP. In this chapter, first EMV-CAP is discussed in more detail. We then introduce the concept of What-You-See-Is-What-You-Sign (WYSIWYS) and analyse a device used for online banking that makes use of this security property, the e.dentifier2. Both a manual and an automated analysis of this device were performed. Finally, we will present a solution to secure online transactions that provides stronger security guarantees than most existing solutions.

## 5.1 EMV-CAP

Though the EMV-CAP standard is confidential and only available under a non-disclosure agreement, it has been largely reverse engineered [DMA09, ST11] and an informative description is leaked (apparently accidentally) in [Che, Appendix 1]. To use EMV-CAP, an additional application is included on the bank card next to the regular EMV application.

For an EMV-CAP session, a full EMV transaction is performed by first requesting an ARQC followed by a request for an AAC. A session starts by requesting the PIN from the user. The user is optionally presented with a challenge and/or other transaction-related data by the bank. After the PIN verification, this data is either entered on the handheld reader by the user or transferred to it using, for example, a USB connection. The ARQC is used in combination with the ATC to generate a response for the bank, while the AAC is only used to leave the card in a clean state by correctly finishing the transaction. The response is typically computed by applying the Issuer Proprietary Bitmap (IPB) on the concatenation of the PAN Sequence Number (PSN), ATC, ARQC and IAD, thus selecting only part of the bits from these values. The PSN is optionally included in the bitfilter and whether it should be included is indicated in the Issuer Authentication Flag on the card. The result of this bitfilter is then converted to digits, making it easier for the user to enter them again on the bank's website. Several variations exist within the standard, defining

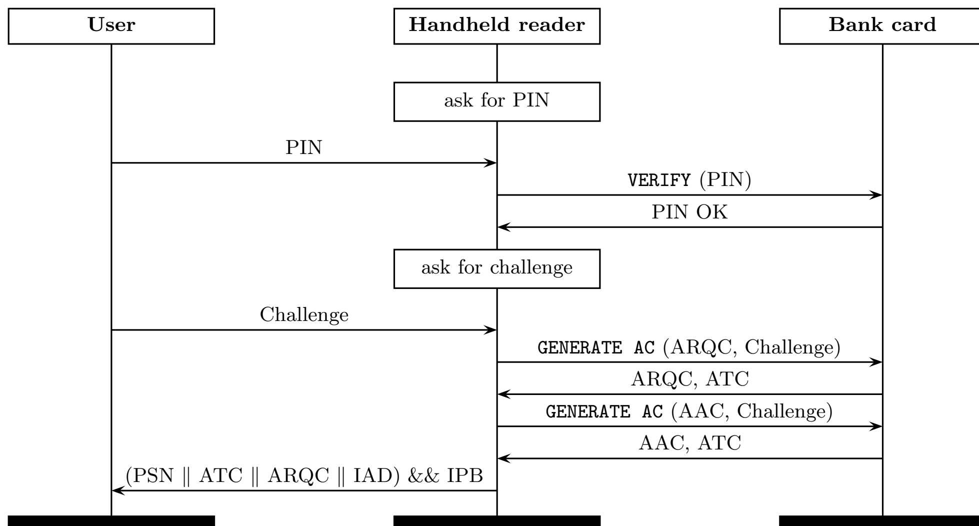


Figure 5.1: A typical EMV-CAP session using Mode 1 without amount or currency code

what data is sent to the card in the `GENERATE AC` commands and how the response is computed.

In *Mode 1* the bank always provides a challenge to the user. This challenge is then sent to the card with the `GENERATE AC` command in the field for the Unpredictable Number, as specified in the corresponding CDOL. Next to this, the amount and currency code can also be included in the command. If a value is not included it is replaced with a constant value of all zeroes. The response is computed by applying the bitfilter as discussed before. A typical example of Mode 1 without amount or currency code can be seen in Figure 5.1. This variant, where no amount or currency code is set, is also referred to as *Mode 3*.

*Mode 2* is identical to Mode 1, except that no challenge or additional data is included in the `GENERATE AC` commands at all. The response value is therefore only dependent on the ATC, which makes it possible to harvest these responses in advance from a card. This would only be detected by the bank if an authorisation using a higher ATC already took place before the harvested responses are used. As an extension to this mode *Mode 2 with Transaction Data Signing (TDS)* was introduced. In this mode, additional data from the bank is used in the authorisation. The different data fields from the bank are concatenated and used as input to a MAC using DES. The key that is used for this MAC is the ARQC, and the resulting MAC is used instead of the ARQC when applying the bitfilter. As with the regular Mode 2, no transaction related data is sent to the card. So again, data can be harvested in advance and be used to compute the correct response when the bank presents the challenge and additional data to be signed. By using this last mode any guarantee that the

card was present is lost, which you can get by using a smart card for security-critical computations in calculating the response to a challenge.

Usually, the PIN code used for the EMV-CAP application on a bank card is identical to the one used for the regular EMV transaction. This is convenient for the user, as he only needs to remember one PIN code but introduces a security risk as well [DMA09]. For example, when using a rogue POS terminal to pay for your groceries, this terminal could at the same time contact your bank and set up an online transfer. Having access to all necessary data to complete an EMV-CAP successfully – including the necessary PIN code – it can compute the correct response to authorise the transaction. A typical measure banks take to prevent this kind of attack is by asking the user for additional data that is not present on the chip of the bank card when authorising an action, for example when logging in to the bank's website. This additional data could, for example, be a username/password combination or a card number.

Another risk is type confusion for the values provided by the bank [DMA09]. An example of this is when Mode 1 with the transaction amount included is used for both log in and signing transactions. When logging in, the transaction amount is set to 0. As a result of this, logging in and transferring 0 euro will result in an identical response code. An attacker could therefore try using social engineering to convince a user to perform a transaction of 0 euro and reveal the response code, thus enabling the attacker to log in on the user's account using this response code.

The main limitation of EMV-CAP is that the user, in general, does not know what he is exactly signing. With Mode 1, when the user is provided with a challenge and amount of the transaction, the destination account of the transaction is not included in the authorisation. The user can therefore never be sure where he is sending the money to. This is even worse when the amount is not included in the authorisation, as he cannot even be sure if he is not transferring all his money to the attacker. When using Mode 2 with TDS, additional data might be included in the authorisation. However, the meaning of these data fields is not standardised and it depends on the implementation of the handheld reader how clear their meaning is to the user. For example, with the Random Reader of the Rabobank there is no functionality to indicate the meaning of the data fields. The only way to indicate the meaning is therefore through the website, which might be under control of the attacker in, for example, a Man-in-the-Browser attack.

## 5.2 What-You-See-Is-What-You-Sign

A two-factor authentication as EMV-CAP, which requires access to a smart card and a PIN code, is of course much stronger than a traditional password. Still, a serious and fundamental limitation of these handheld readers is that the challenges have to be quite short, usually 8 digits, for the user to be able to easily enter them on the reader. Therefore they do not offer much scope for a message that is meaningful to

the user. As the challenge is often just random the user has no real idea what he is authorising. This means that a Man-in-the-Browser attack, where the attackers controls the browser on the client's PC, can still let someone unwittingly approve unwanted transactions.

This risk can be mitigated by letting the user sign additional challenges, for instance the amount or say the last eight digits of the bank account, which are meaningful to the user. Some online banking sites use such additional challenges for transfers with high amounts or transfers to bank accounts not used before by a customer. The downside of this is more hassle for the user, typing the extra challenges and responses. Also, a compromised browser could *still* trick users into signing these additional challenges, as a user cannot tell if an apparently random challenge is not in fact an amount or the last 8 digits of the attacker's bank account. Users could be asked to type in more than eight digits, possibly using devices with a bigger display, but clearly there will be a limit on how much hassle users are willing to accept.

Connecting the reader to the PC with, for example, a USB cable can solve the problem, or at least drastically reduce the risk and impact of Man-in-the-Browser attacks. The user no longer has to retype challenges and responses, making longer challenges acceptable. Moreover, the display can be alphanumeric (even if the keyboard is numeric only), allowing more meaningful challenges for the user. Not only security is improved, but also user-friendliness, as users do not have to retype challenges and responses.

Having such a setup, it is possible to implement so-called WYSIWYS functionality. This means the information that is shown on the screen is actually included in the signature generated to authenticate the action and this signature is only computed after explicit approval by the user. If the challenge displayed includes, for example, the destination account and the amount, this provides protection against Man-in-the-Browser (MitB) attacks as the user knows what will be signed and has to approve this. Depending on the actual displayed messages, WYSIWYS therefore can provide very strong security guarantees for both the user and the bank.

## 5.3 e.dentifier2

The e.dentifier2 is a handheld reader to secure online banking transactions originally developed by Todos AB, which is now owned by Gemalto. It provides a screen and numeric keyboard, and it can be used with or without a USB connection. If used with the USB connection, the device provides WYSIWYS functionality – called Sign-What-You-See (SWYS) by Gemalto. As this sounded promising and we were interested in possible other uses, the device was largely reverse engineered by Arjan Blom [Blo11]. On the website of the manufacturer the device was promised to be "the most secure "sign-what-you-see" end-user device ever seen", though this was certainly not the case in the first version as shown in [Blo11, BKGP+12]. An attacker that controls the user's PC is able to let the user sign a transaction without explicit approval, i.e

WYSIWYS would be broken. As the source code of the device is not available, it was necessary to reverse engineer the device to discover how it operates. This was done by looking at the browser plugin that is used to communicate with the device, the USB traffic between the PC and the device, and the smart card communication between the device and the bank card.

### 5.3.1 Unconnected mode

Without USB connection, the device behaves like a standard EMV-CAP reader. The device offers 5 different modes in its menu:

- *Log on*, in this mode no challenge is entered by the user. The user is asked for a PIN, and is presented a response code to login to the bank's website. Mode 2 of EMV-CAP is used for, so no challenge is sent to the card.
- *Send transact.*, this mode is used to confirm a transaction. The user can enter a challenge of up to 8 digits. This mode does not quite follow the known EMV-CAP modes, as the data sent and received from the card is first mangled using some hashing and/or encryption. A challenge is sent to the card in the `GENERATE AC` command, as with Mode 3 of EMV-CAP. This challenge is not the same as the one entered by the user but it is dependent on this value. Also, the ARQC is not used directly in the application of the bitfilter. Instead it is used to mangle some data, which includes at least the challenge entered by the user. Changing the least significant bits of the ARQC does not change the response code, which is a strong indication it is used as a key in some DES operation as these bits are used as parity bits in DES and not used in the actual cryptographic operations.
- *Check acc. nr.*, this mode seems to be identical the previous mode. The mangling of the data sent to the card and received from it seems to be different, indicating the mode is somehow included in the mangling of the data.
- *Check input*, again this mode seems to be identical to the two previous modes except for the data mangling.
- *SecureCode*, this mode provides EMV-CAP Mode 1, including the amount, currency and challenge. As opposed to the other modes of the e.dentifier2, the user is first asked to enter the transaction details and challenge before entering the PIN code. The user has four options for the currency: Euros, Dollars, Pounds and other (which results in a currency code of 999). Here, the challenge is sent to the card without any additional mangling and the bitfilter is applied directly on the ARQC.

The additional mangling in some of the modes means that there are unknown functions hard-coded in the device. The secrecy of these functions is not crucial for the security, but it does prevent interoperability with EMV-CAP readers issued by other

banks, and it prevents the construction of a software emulation of the device, as is available for other EMV-CAP readers[1].

### 5.3.2 Connected mode

More interesting functionality is provided when the device is connected to a PC using a USB connection. To use this connected mode, customers have to install a special driver (only available for Windows and MacOS). The browser then interacts with this driver via JavaScript and a browser plugin. The browser plugin checks whether it is connected to either the domain `abnamro.nl` or `abnamro.com` using TLS. If this is not the case, the plugin will not function. The Firefox plugin also allows local files to use the plugin. This might introduce security risks as, for example, this means that attachments in emails could also use the plugin.

In connected mode, an internet banking session starts with the reader reading the bank account number and the card number from the smart card and supplying it to the browser. This way the user does not have to type this in, making the system more user-friendly.

To log in, the reader first prompts the user for his PIN code. It then displays a message saying that the user is about to log in and asks the user to confirm this by pressing OK.

To confirm a bank transfer, or a set of bank transfers, the reader will again prompt the user for his PIN code. It then displays a message giving the number of transfers the user is about to approve and the total amount of the transactions combined, and asks the user to approve this by pressing OK.

The additional security of the connected mode over the unconnected mode here is that you see what you sign, even if the browser or the PC it runs on is controlled by malware. Connecting the e.dentifier2 to a possibly infected PC by USB does introduce a new attack vector: malicious code on the PC could try to interfere with the device. Still, given that the device is so simple and offers so little functionality, it should be possible to design and implement it so that it is secure against such attacks.

The device can display some predefined messages, such as "Transactions sent successfully." or "Please follow the instruction on your computer.". This last message would of course be ideal for a phishing attack. Users might be easier to convince to enter sensitive information on a fake page if their e.dentifier2 displays such a message.

### 5.3.3 The SWYS protocol

The reverse engineering of online banking with the e.dentifier2 in connected mode was done by eavesdropping on the USB and smart card communication, and replaying of modified traffic to determine dependencies between data elements [Blo11]. This

[1] Available from http://sites.uclouvain.be/EMV-CAP

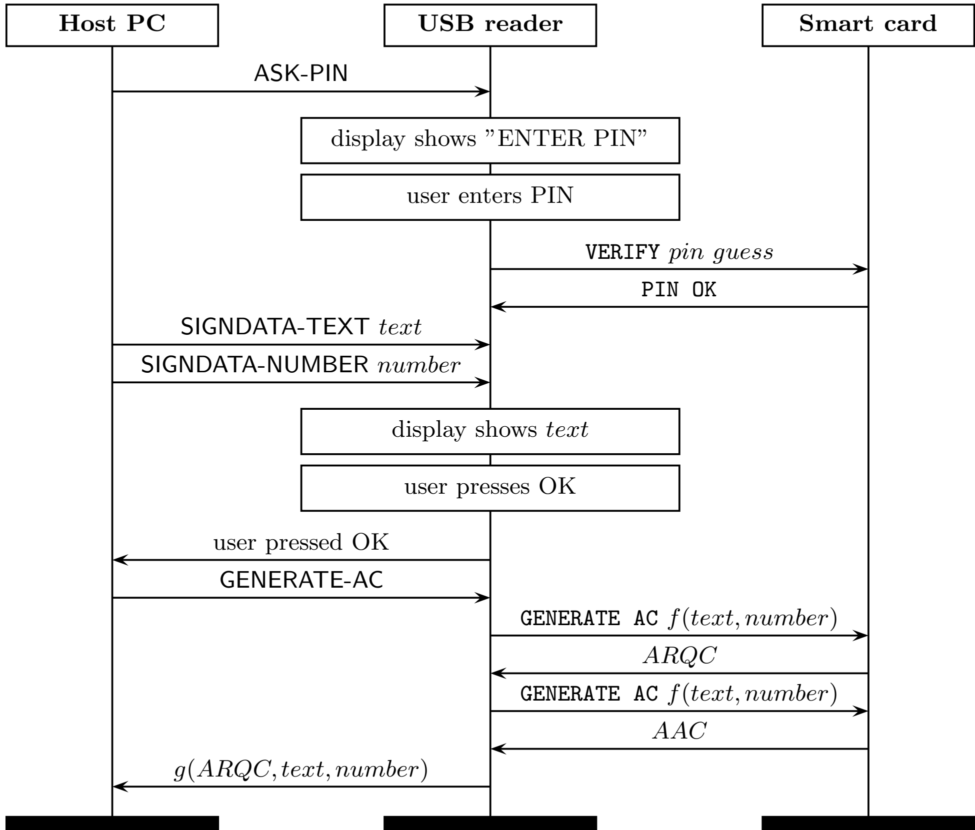


Figure 5.2: SWYS protocol for log-in and for transactions

process revealed the protocol, as it is used by ABN AMRO, shown in Figure 5.2. This figure outlines the abstract SWYS protocol for logging in or confirming a transaction. In either case the web browser sends so-called *signdata* to the reader. This signdata consists of two parts, namely some text (that is shown on the display) and some number (which is not):

- In the case of a login, the signdata text specifies the account number and bank card number; in case of a transaction, it specifies the number of transactions and the total amount.
- In the case of a login, the signdata number consists of two values: a 4 byte value giving the current UNIX time and a seemingly random 8 byte value; in case of a transaction, it is a 20 byte apparently random value.

In the protocol, the reader asks the smart card to produce two cryptograms, using the `GENERATE AC` command. Included in these commands is the Unpredictable

Number. As is usual in EMV-CAP transactions, the first cryptogram is an ARQC, the second an AAC.

By replaying earlier transactions, and varying the text and number parts of the signdata, we could confirm that the UN does depend on both the text and the number. Hence this payload is written as $f(text, number)$ in Figure 5.2. It has to depend on the text to make sure that 'we sign what we see'; including the number is useful to diversify the input and prevent replays, though the computation of the cryptograms will also involve the card's transaction counter. We do not know what the function $f$ is.

To finish a transaction, the reader sends a response back to the web browser, which is based on the cryptograms generated by the smart card and the complete signdata. Again, we do not precisely know how this response is computed. However, as with the unconnected mode it seems that the ARQC reported by the smart card is used as the key in a Data Encryption Standard (DES) operation. Hence it is written as $g(ARQC, text, number)$ in Figure 5.2.

Analysis of the browser plugin and corresponding JavaScript libraries revealed the device also supports Mode 1 and 2 of EMV-CAP via the USB connection. As there is no challenge or additional data in Mode 2, the user is not asked to confirm the action but only to enter his PIN. With Mode 1, the user is asked to confirm the amount and currency of the transaction. As opposed to the protocol as depicted in Figure 5.2, the user is asked to first confirm the transaction details and only after this to enter his PIN code. For both EMV-CAP modes, the responses that are returned to the PC are identical to the ones that would be displayed on the device when used in unconnected mode.

### 5.3.4 The attack

As can be seen in Figure 5.2, the reader sends a message to the host PC indicating the user pressed OK. After this the host PC sends a command to generate the cryptograms to the reader. This seems strange, as the reader would be expected to generate the cryptograms automatically after OK has been pressed. The driver on the host PC should not play a role in this.

This weakness can be exploited: by sending the request over the USB line to generate the cryptograms without waiting for the user to press OK, the cryptograms are generated and the reader returns the response over the USB line, without the user getting a chance to approve or cancel the transaction. To make matters worse, a side-effect of giving this command is that the display is cleared, so the transaction details only appear on the display for less than a second. We demonstrated this attack in an actual internet banking session.

This means that an attacker controlling an infected PC can let the card sign messages that the user did not approve, thus defeating one of the key objectives of WYSIWYS. The user still has to enter his PIN, but this is entered at the start

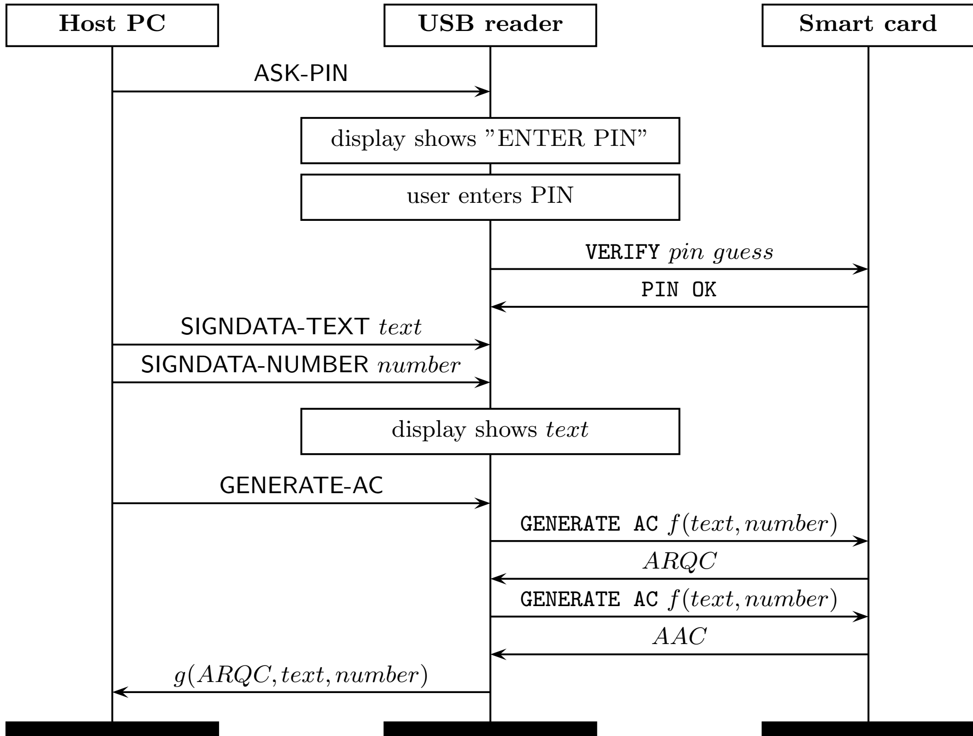


Figure 5.3: Attack on SWYS protocol; the difference with Figure 5.2 is that the driver directly gives the instruction to generate the cryptograms, without waiting for the user to press OK

of a transaction before any transaction details are shown, and after this no more interaction is needed from the user to sign malicious transactions.

### 5.3.5 Extended length challenges

Initially it was mind-boggling to us how this vulnerability could ever have been introduced. Subsequently it became clear that the vulnerability may have been introduced as a side-effect of having additional functionality in the device, where the device shows several messages for the user to approve. Presumably this functionality is included for more complex transactions where more than 68 characters are needed to display transaction data.

We never observed such transactions in our use of the online banking system, but using our own software we could see that the device is capable of doing this. Several messages can be sent to the reader in turn, with the next message being sent after the user presses OK, and then after the final message the driver can give the command to generate the response. We could observe that the 4 byte payload sent as challenge to

the smart card depended on all the texts that were sent to the reader and displayed there. Presumably the unknown function $f$ implemented in the device hashes these texts together to compute this challenge.

For this variant of the protocol the reader needs to communicate with the driver after the user presses OK, namely to request the next part of the message. This might explain why the weakness has been introduced in the first place, and why it was missed in security reviews.

Note that this variant of the protocol results in an overloaded semantics for the OK button: it can mean an instruction (to the driver) to 'send more data to the display' or an instruction (to the smart card) to 'go ahead and generate a cryptogram'. Since the reader is not aware of the meaning of the button, the host PC has to determine this, providing a possible origin of the vulnerability.

### 5.3.6 Preventing the vulnerability

The attack that we found is something that should and could have been detected. After all, the attack does not involve some detailed cryptanalysis and does not rely on a cleverly crafted MitM attack or exploit some subtle low level coding mistake. A patent application describing an e.dentifier2-like solution has been filed by the company that produces it [Gul10]; the higher level description of the protocol given there does not include the vulnerability that we found.

We have no insight in the procedures followed in design, implementation, or testing, or indeed any of the associated documentation, so we can only speculate how the vulnerability could possibly have been missed. Still, we can consider how it could have been spotted, or, better still, prevented in the first place.

Firstly, the attack breaks one of the central security objectives of WYSIWYS. This security objective should be used as a basis for considering abuse cases. An obvious abuse case that could already have been identified in the early design phase is: malicious code on the PC that tries to get the reader to sign without the user's approval. It seems unlikely that the vulnerability could have gone unnoticed in design, implementation and testing if this abuse case was made explicit, as it would then have been considered by e.g. a reviewer of the specifications or tester of the implementation. In fact, one would then expect someone developing the specification or implementing it to notice the problem long before any post-hoc security evaluation.

As discussed in Section 5.3.5, it seems that the semantics of pressing the OK button is overloaded. This overloading could be spotted in a high-level specification that only considers the interaction with the user, if in such a specification one tries to make the semantics of pressing OK explicit.

Finally, even if the problem went unnoticed in the design, the problem could still have been detected by more systematic testing. Exhaustive testing is of course impossible, even for a system as simple as this: the messages over the USB line are small but too long for exhaustively testing all possible contents. Still, the number of

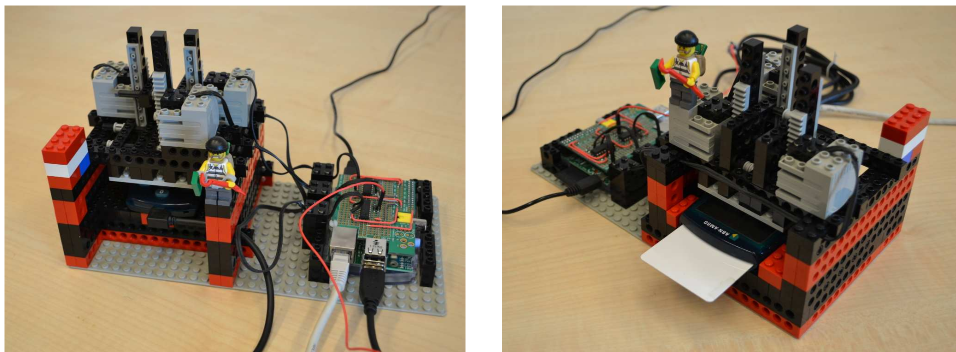

Figure 5.4: Our Lego robot is capable of pressing three buttons on the e.dentifier2. The learning setup includes the Lego robot, circuitry to power the engines, and a Raspberry Pi that interacts with the e.dentifier2 and controls the engines.

different types of messages over the USB line is very small. The number of internal states of the reader, which correspond to particular states of the simple protocol it implements, is very small too. This makes it a feasible option to apply automated learning techniques as will be shown in the next section.

## 5.4 Analysing the e.dentifier2 using Lego

In this section we describe our analysis of a handheld smart card reader for online banking – the e.dentifier2 that was introduced in Section 5.3 – using automated learning techniques. This section is based on [CPPR14]. The original robot was built by Georg Chalupar and Stefan Peherstorfer. The goal of this research was to see if we could automate the analysis described in Section 5.3. At the moment of this research a new version of the e.dentifier2 was released where the problem that we discovered was fixed. This gave us two different devices to test our approach on.

To prevent people from opening and changing or analysing handheld readers, these often contain tamper-proof features that prevent the reader from functioning after it has been opened. To avoid breaking the device by opening it, a robot was constructed using Lego and controlled by a Raspberry Pi to press the buttons. Controlling all this from a laptop, we then can use LearnLib to learn the behaviour of handheld readers.

### 5.4.1 Lego robot

To press the buttons on the e.dentifier2 we found a cheap and flexible solution by using Lego motors (see Figure 5.4). A small gear on the motors is used to move a Lego bar via a toothed rack.

In our experiments, we used a smart card that we programmed ourselves to provide

the required EMV support. An advantage of this was that we could fix some of the behaviour of the card, and make it produce fixed responses for each EMV command. For a real bank card the ATC is included in the Application Cryptogram and increased after each request, and therefore the responses to the `GENERATE AC` will be different each time. The resulting change in the output would have to be filtered out to learn the state machine with LearnLib.

The robot needs to be able to enter PIN codes and press the OK and Cancel buttons. As we used our own smart card, we let it accept all PIN codes so that only one 'finger' is needed to enter the same digit four times. In total only three fingers were needed and three Lego motors to move them up and down.

We choose a Raspberry Pi to actuate the motors as it offers General-purpose Input/Output (GPIO) pins which are easy to program and, like the Lego motors, it is also powered by 5 Volt. Additionally, the Raspberry Pi contains a USB port, which is used to communicate with the e.dentifier2, and an Ethernet port, which is used to control the Raspberry Pi from the PC running LearnLib.

When setting the timing for moving the fingers up and down, there is the risk that on long test runs, taking several hours and thousands of key presses, errors in the timing would accumulate and cause the fingers not to press down enough. We therefore set the time for moving the finger down a small fraction too long, so that we are certain that the buttons are always pressed down completely.

### 5.4.2 Software

Our test harness for the e.dentifier2 is written in Python and runs on the Raspberry Pi. This script controls the motors and resets the device. Additionally, it sends the USB commands to the e.dentifier2 using the PyUSB library. LearnLib runs on a separate laptop and the queries are sent to the test harness via a TCP socket. The input alphabet for LearnLib consists of command names that represent either USB messages or commands for the Lego robot. For the equivalence checking, we used either the random walk method or the W-method.

Especially in longer experiments, we experienced problems with non-deterministic behaviour due to, for example, buttons not always correctly being pressed. At one point we got a lot of non-deterministic behaviour, which LearnLib cannot handle and results in an endless loop. After some debugging it turned at that the numeric button used for the PIN code got stuck once in a while because it was worn down too much by all the button presses. After changing to another digit, the robot functioned again. To counter this kind of problems, we made use of an alternative version of the software using majority voting where all queries are executed twice. If the results are different, the query is executed a third time and this output is given if it is equal to one of the first two outputs. If all three outputs are different, the software will throw an exception.

### 5.4.3 Experiments

**Input and output symbols**

The input symbols are based on the USB commands discovered by the previous reverse engineering discussed in Section 5.3. A normal transaction starts with the SHIELD command, after which the e.dentifier2 displays the logo of the bank (a "shield") three times. Next, INSERT_CARD sets the language (in our case "EN" for English) and waits until the bank card is inserted. The PIN command causes the device to ask for the PIN and send it to the card. After that, SIGNDATA sends binary data to be included in the cryptogram. The DISPLAY_TEXT command encodes a text to be included in the cryptogram, that is shown on the screen and has to be confirmed by the user with the OK button. Finally, GEN_CRYPTOGRAM tells the e.dentifier2 to get a cryptogram from the smart card.

In addition to these USB commands, the input symbols ROBOT_OK and ROBOT_CANCEL instruct the robot to press the OK or Cancel button respectively. To accelerate the learning process by keeping the number input symbols and states low, we defined some input symbols that combine two or more inputs.

- COMBINED_INIT consists of SHIELD followed by INSERT_CARD;
- COMBINED_DATA combines SIGNDATA and DISPLAY_TEXT;
- COMBINED_PIN translates to the PIN USB command plus the robot pressing four times a digit followed by the OK button.

Moreover, we added the option to automatically issue the COMBINED_INIT command after every reset of the e.dentifier2.

The USB responses of the e.dentifier2 are used as output symbols for the learning process. The smart card communication and messages on the screen were not observed.

After a robot action or sending a USB command, the Raspberry Pi waits 1 second for a response via USB. We assumed that USB messages are sent in chunks of 8 bytes, as the previous reverse engineering indicated. Although the USB responses seem to encode whether more chunks will be sent, our script attempts to read data via USB until no more data is received for 1 second. While our smart card is programmed to always send the same cryptogram, the USB responses to the GEN_CRYPTOGRAM command can vary because the e.dentifier2 manipulates the cryptogram based on the data supplied by SIGNDATA and DISPLAY_TEXT. Therefore, we classify every message starting with a particular prefix as a cryptogram. Optionally, we distinguish an EMPTY_CRYPTOGRAM, which is the response when neither SIGNDATA nor DISPLAY_TEXT has been sent before the cryptogram is requested, and a VALID_CRYPTOGRAM, which is the response after sending the original commands as used in a valid transaction (i.e. one SIGNDATA and one DISPLAY_TEXT).

### Learning parameter and timings

At the beginning of every input sequence, the e.dentifier2 has to be reset to an initial state. As we want to be sure all previous operation have finished and it is ready to accept commands after this, we included some waiting times. On average a reset command therefore takes 5.6 seconds. Pressing a single button takes 2.5 seconds on average, including waiting for the device to get ready to accept a button press and waiting for a possible response. Combining multiple key presses when entering a PIN code takes on average 4.4 seconds. A command that only requires a single USB message takes 1.1 seconds on average, mainly due to the waiting for a possible response.

We first learned models using coarser-grained combined actions, and then we learned more detailed models using more fine-grained actions:

- First, we learned a coarse-grained model where COMBINED_INIT is performed after every reset and with a reduced input alphabet consisting of COMBINED_DATA, COMBINED_PIN, ROBOT_OK, and GEN_CRYPTOGRAM. Moreover, we did not distinguish between the EMPTY_CRYPTOGRAM and other CRYPTOGRAM responses in this setup. We learned a model with 3 states for the old version of the e.dentifier and 4 states for the new version using 85 and 65 output queries respectively. The equivalence checking using the random walk method with 50 queries with a length of 4 or 5 input symbols for each model could not find a counterexample. The whole process took 45 and 30 minutes respectively.

  These coarse models, shown in figures 5.5 and 5.6, already have enough detail to show the flaw in the old version and reveal differences between the old and new version, as discussed in more detail in Section 5.4.4.

- Next, we learned a more fine-grained model for the new version using a simple reset and the commands COMBINED_INIT, COMBINED_PIN, ROBOT_CANCEL, ROBOT_OK, SIGNDATA, DISPLAY_TEXT, and GEN_CRYPTOGRAM as input alphabet (see Figure 5.7). This time we distinguished between the EMPTY_CRYPTOGRAM and other CRYPTOGRAM responses. After equivalence checking found counterexamples for two hypothesis models, the final model with 8 states has been verified with 500 random queries with a length of 6 or 7 input symbols. In total, 578 output queries and 894 queries for equivalence testing have been performed and the process took 7:15 hours. For a more extensive test using the test harness implementing majority voting with equivalence queries of length 10 to 15 and a maximum of 1000 equivalence queries, the random walk method took almost 23 hours for 580 membership and 1091 equivalence queries. Using the same setup, we ran the W-method for a maximum of 8 states, which took about 88 hours. In this period, 578 membership queries and 7530 equivalence queries were executed to determine the final model.

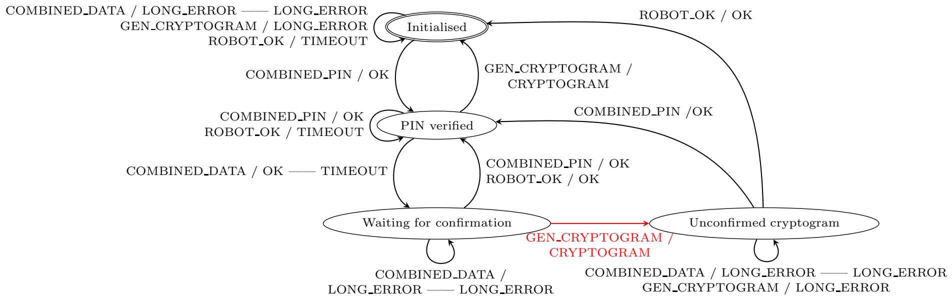


Figure 5.5: The learned coarse-grained model of the old version of the e.dentifier2. The initial state is marked with a double ellipse.

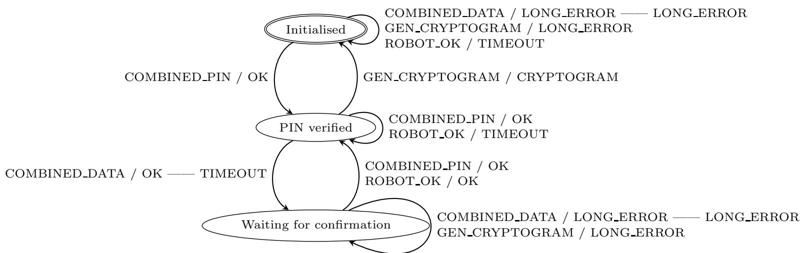


Figure 5.6: The learned coarse-grained model of the new version of the e.dentifier2.

### 5.4.4 Discussion of obtained models

Figure 5.5 shows the generated state diagram of the old version of the e.dentifier2. The normal path through the application in case of this input alphabet is: COMBINED_PIN, COMBINED_DATA, ROBOT_OK, GEN_CRYPTOGRAM. It is also possible to get a cryptogram with the combination COMBINED_PIN, GEN_CRYPTOGRAM, which is an empty cryptogram.

The security vulnerability discussed in Section 5.3 occurs in the state *waiting for confirmation*, where it is possible to get a cryptogram without pressing the OK button for the transaction with the following inputs: COMBINED_PIN, COMBINED_DATA, GEN_CRYPTOGRAM, leading to the state *unconfirmed cryptogram*.

The state diagram of the new version of the e.dentifier2 (see Figure 5.6) does not have this additional state *unconfirmed cryptogram*. So the device lacks this (superfluous) state which causes the security vulnerability. This shows that the vulnerability present in the old version has been fixed: the only paths through the application that lead to a valid cryptogram are the legitimate ones that are expected.

To generate a more detailed state model, we refined the inputs as described in

Section 5.4.3. This leads to more states and paths as shown in Figure 5.7. In this state diagram, the COMBINED_INIT command is used independently from the RESET command which leads to several uninitialised states. As visible in the model, the *initialised* state and the *error* state are almost the same, except that pressing the OK or the CANCEL button gives different behaviour. After COMBINED_PIN, both states end up in the *PIN verified* state. The normal way through the application from the *PIN verified* state is: SIGNDATA, DISPLAY_TEXT, ROBOT_OK and then back to the *initialised* state by generating a cryptogram with GEN_CRYPTOGRAM. By repeating the DISPLAY_TEXT and the ROBOT_OK command, more text can be added to the signed data for the cryptogram. This is necessary if the user should confirm more text than fits on the display. Additionally, in the *ready to sign* state it is possible to add data for the cryptogram with the SIGNDATA command. The COMBINED_PIN leads to the *PIN verified* state, no matter if the current state is the *waiting for confirmation* or the *ready to sign* state. In the *waiting for confirmation* state, the user is prompted to confirm the data on the display. If the Cancel button is pressed, the e.dentifier2 is reset to the *initialised* state as expected.

The fact that there are the states *uninitialised1* and *uninitialised2* shows that there is still some strange behaviour but at least it is not possible to generate a cryptogram by bypassing the confirmation of the user. Also, the *error* state and the *initialised* state could be combined to one state.

When looking at the more detailed model of the old device, we not only discovered the known bug but also additional strange behaviour. The COMBINED_INIT command does not seem to influence a regular protocol run (COMBINED_PIN, SIGNDATA, DISPLAY_TEXT, ROBOT_OK and GEN_CRYPTOGRAM). It is possible to start with a protocol run before issuing the COMBINED_INIT command and issue this command at any point before GEN_CRYPTOGRAM is executed. This will still result in a valid cryptogram and the device still displays the text and asks for the PIN code as usual. There is however no response returned over the USB line yet before the COMBINED_INIT command. This behaviour is no longer present in the new device.

### 5.4.5 Non-deterministic behaviour

One problematic issue in our experiments was dealing with non-deterministic behaviour of the system under test. LearnLib cannot cope with non-deterministic behaviour, and will fail to terminate if it encounters non-determinism.

In one instance, non-determinism was traced to one of the buttons of the e.dentifier2 keyboard intermittently malfunctioning, namely the digit button used to enter the PIN code, probably due to it having been pressed thousands of times by our robot. We solved this by switching to a different digit for the PIN code.

More problematic was non-deterministic behaviour on the old e.dentifier2 that showed up in some tests. Randomly, there are 8 unexpected bytes, usually at the end of an expected response or between two different expected USB responses. This

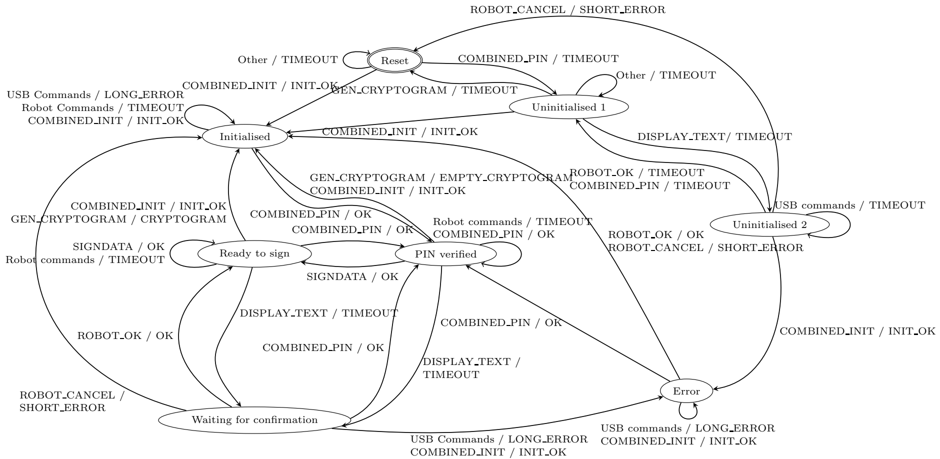


Figure 5.7: The learned fine-grained model of the new version of the e.dentifier2 using the W-method for 8 states.

additional byte sequence normally looks similar to this one: `0281010100000000`. The new e.dentifier2 shows no such behaviour, which leads to the conclusion that there was not only a security bug fix but also an update of the USB part of the firmware. The byte sequence might be some error code of the old USB stack. The Python script on the Raspberry Pi was modified to filter out such 'error' bytes to learn the state machine for the old e.dentifier2.

### 5.4.6 Conclusions

Using the LearnLib library for state machine learning and our Lego robot, we can automatically reverse engineer the state machines of handheld smart card readers for online banking.

The state machines obtained for the different versions of the e.dentifier2 device immediately reveal differences between different versions, and can be used to spot the security flaw in the older version. Despite the fact that our Lego robot is rather slow, it can learn this difference within 45 minutes per device. The state machines obtained for the new version of the e.dentifier2 show that the security vulnerability has been fixed there, also when we study the fine-grained model. This confirms the usefulness of state machine learning as a technique to automatically finding security flaws in these type of devices.

As the set-up is fully automated, it can be used to perform very thorough tests to look for unwanted behaviour; here we can learn more detailed behaviour, for example to check for the presence of insecure or unneeded behaviour in the newer, patched version. Of course, there are limits to what can be done with such automated state

machine inference: we cannot hope to find a well-hidden malicious backdoor, but we can expect to find accidental flaws in the programming logic.

Although the new device does not contain the old flaw, the more detailed state machine obtained for the new device (see Figure 5.7) is still surprisingly complex. To reduce the potential for things going wrong (as they clearly have done in the past), we wonder whether it would not be better, already from the early design phase, to try and keep the protocol state machine as simple as possible.

Apart from confirming the security fix in the new version of the e.dentifier2, the differences in presence of non-deterministic behaviour between the old and new version suggest that the new firmware not only contains the security fix but also improvements in the USB driver, as the new version no longer generates intermittent errors in the USB traffic.

# Chapter 6

# Radboud Reader

In this section the design of a device for securing online transactions is presented [PR13]. This device could be used, for example, for internet banking, where it can protect against PC malware, including Man-in-the-Browser attacks.

A fundamental problem in securing online transactions is the lack of a trustworthy device to communicate with the human end user. The software on laptops and PCs, but also smart phones, is so large and complex that security vulnerabilities are inevitable. This means that these devices are not trustworthy as input or output channel to communicate to the user (via the display, keyboard or mouse), as malware can manipulate the display and eavesdrop on any input.

Smart cards are a potential improvement in that they provide a secure computing platform. The chances of exploitable security vulnerabilities in the software on a smart card are *much* lower than for a PC or laptop, given the very small size and the very limited functionality of the code. The code on a smart card is in the order of a few kilobytes and offers a very restricted interface, whereas the code of a laptop or mobile phone operating system is in the order of many megabytes and presents a huge attack surface for an attacker. However, a serious and fundamental limitation of smart cards is the lack of a display and keyboard. This means that a smart card requires some terminal with a display for output and a keyboard for user input, and this terminal has to be trusted. Still, the first smart cards with keyboard and display are in commercial use. For these smart cards the size is a limiting factor, i.e. the size of the display is quite limited and the same holds for the keys that can be used.

Given the issues above, the more secure solutions that use smart cards to secure online transactions rely on a dedicated smart card reader with a numeric keyboard and display. The most prominent example is the EMV-CAP standard for internet banking discussed in Section 5.1. Some of these devices are connected by USB to the computer, which can both increase user convenience – as the user does not have to type over numbers to and from the device – and security, by providing a What-You-See-Is-What-You-Sign guarantee.

In this section an alternative design is presented for a solution, the Radboud Reader, where we rigorously stick to the design principle that the device should be as simple as possible. Where possible functionality is moved to the smart card rather than the reader and the smart card is given control as much as possible. Although these principles are rather obvious, and the resulting system is quite a natural solution,

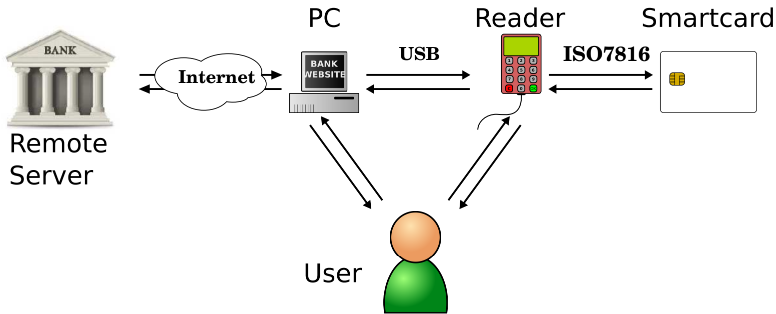


Figure 6.1: Set-up

we are not aware of anyone else proposing a system like this. The resulting system has some interesting advantages over existing solutions:

- The device is *simpler* than alternative solutions: it contains no secrets, and does not need to support any cryptographic operations or even hashing. Keeping the device simple also reduces the chance of security vulnerabilities in the device.
- Our solution gives *stronger security guarantees* than most other solutions: the device provides a trusted display which only displays text approved by the smart card, so that the smart card can check say a digital signature on any text before it is displayed.
- The device is *generic* and can be used in combination with different smart cards for different purposes.

## 6.1 Security objectives and high-level design

The basic set-up, shown in Figure 6.1, is the same as with any USB-connected smart card reader for internet banking: a web browser on a PC communicates with a remote web server over the internet and with a smart card reader via a USB cable. The smart card reader has its own display and numeric keypad with two additional buttons marked 'OK' and 'Cancel', so the user can interact with both the PC and the reader for input and output, via their respective displays and keyboards.

The objective is that, unlike the PC, the reader can be a trusted input and output device, meaning that we rely on

**S1** authenticity of whatever is displayed on the display,

**S2** confidentiality of anything that is entered on the keyboard,

**S3** non-repudiation of any transactions confirmed or declined by pressing 'OK' or 'Cancel'.

Although a complete transaction might be performed using only the reader, this is not very user friendly given its limited display and numeric-only keyboard. Therefore, the PC can still be used to set up the connection to the service provider and enter the transaction details.

## 6.2 Attacker model

The attacker is assumed to be in full control of the network and of the PC, including the USB connection to the reader, but not of the reader or the smart card, and the communication between them. If one abstracts away from the PC then this is equivalent to assuming a Dolev-Yao attacker, as discussed in Section 2.2.1, on the communication between the remote server and the reader.

Our main concern is an online attacker, with possibly total control over the PC, but without physical access to the reader or the smart card. Attackers with physical access are only a secondary concern. This means that physical tamper-resistance or tamper-evidence of the reader is not crucial: they are nice properties to have, but in practice one will only want to spend a limited amount to realising them to some degree. Of course, one would want to include protection against shoulder surfing, e.g. by not echoing say a PIN code on the display as it is entered on the reader.

## 6.3 High-level design decisions

For the reader to be a trusted input and output device, we will ensure that

- all output on the display comes from the smart card, and
- all input to the device is only sent to the smart card,

as illustrated in Figure 6.2. This means that the smart card can ensure authenticity of the output to the screen, and confidentiality of the input from the keyboard.

By only allowing the smart card to output to the display, we get a trusted display, where authenticity of displayed messages can be enforced by the smart card. Allowing the PC to display text would allow malware on the PC to control the display, even though the device could still offer the guarantee of WYSIWYS.

So *physically* the reader is between the smart card and the PC, but *logically* – i.e. considering the information flows – the smart card is between the PC and the reader. One could even consider using physically different contacts on the smart card for communication with the display and communication with the PC. Two of the contacts of smart cards are reserved for future use in the ISO/IEC 7816 standard, so this is possible. However, this would be a more expensive solution requiring non-standard smart cards.

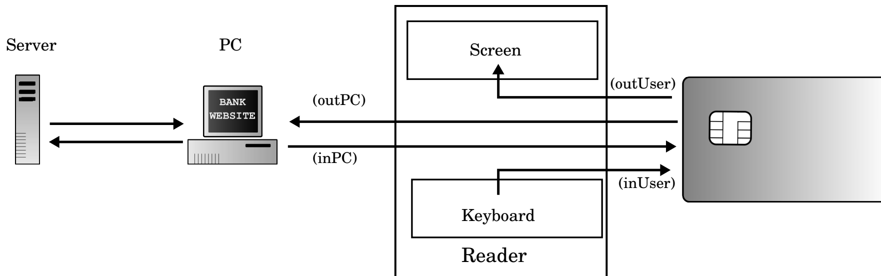


Figure 6.2: Information flows in the Radboud Reader. All I/O the device offers to the user is with the smart card, not the PC, and all communication between the user and remote server has to pass through the smart card.

For the input the Radboud Reader provides a numeric keyboard as well as an 'OK' and 'Cancel' button. Output can be displayed on its display consisting of 80 characters (4 x 20).

Note that it is completely up to the smart card to decide what functionality it provides, and how the data communicated with the PC, and via the PC with the remote server, is secured. Different strategies are possible here:

- The smart card could set up a secure tunnel to the remote server, analogous to TLS, ensuring integrity and confidentiality of the entire communication session between smart card and the remote server. In such a set-up, one could let the smart card *only* communicate with the back-end, and not communicate with the PC at all.

  Many smart cards already provide such a secure tunnelling mechanism, called Secure Messaging in smart card jargon. The ISO/IEC 7816 standard [ISOb] already describes it, as do many other smart card standards, including the ICAO standard for electronic passports [ICA08], the EMV standard [EMV08], and the Global Platform standard [GP06].

- Alternatively, one could choose to sign and/or encrypt parts of messages exchanged between the smart card and the remote server on a more piece-meal basis.

The former approach provides simpler and more robust security. An advantage of the latter approach might be that the browser can see and display some parts of the communication between smart card and PC, which in the former approach would require additional communication between browser and the back-end.

To login to a website using the reader in combination with a smart card, one could of course let the smart card generate a credential to login, using some challenge-response protocol. It is even possible to let the smart card supply the username (or

say, in the case of a bank account, the bank account number) to the remote server when logging in over a secure tunnel, in which case malware on the PC would not even be able to learn this. Paper receipts for credit card transaction no longer show the complete card number but only the last four digits. Similarly, when using the Radboud Reader in combination with a smart card to log-on to some website, there is no reason to show the actual login name on the PC's display if leaking this could give useful information to an attacker.

Inserting a smart card into a smart card reader that is attached to a (potentially infected) PC is of course dangerous. Malware on the PC could try to access functionality of the smart card in unwanted ways, for example by sending PIN code guesses to the card, with a small chance of guessing it right, and a big chance of blocking the card with 3 incorrect guesses. This leads to another security requirement, namely that access to functionality of the smart card is strictly limited, which we will realise by ensuring that

**S4** Input from the PC is forwarded to the smart card in a specific format so that it cannot address arbitrary functionality on the card.

Effectively, the reader should provide a firewall between the smart card and the PC, which only lets minimal functionality through.

That the device uses a USB-connection to a PC to connect to the internet and the back-end is not essential; any means of connecting the device to the internet could be used. USB is the obvious choice in that it is widely available, and indeed existing smart card readers use it. An alternative to this would be to connect with the PC via Bluetooth. Another (more expensive) alternative would be to simply equip the device with GSM and let it by-pass the PC completely, though that requires additional measures to link a session on the device with a session on a PC.

## 6.4 Functional requirements

To achieve the properties discussed previously, the Radboud Reader needs to provide the following operations:

1. Starting a session, by selecting the desired application on the smart card; a smart card can hold several applications, and one has to be selected at the start of a session.

2. Forwarding data received from the PC to the smart card.

3. Carrying out instructions received from the smart card; these instructions can tell the reader to

   a) display text and wait for the user to press 'OK' or 'Cancel';
   b) display text and wait for user input;

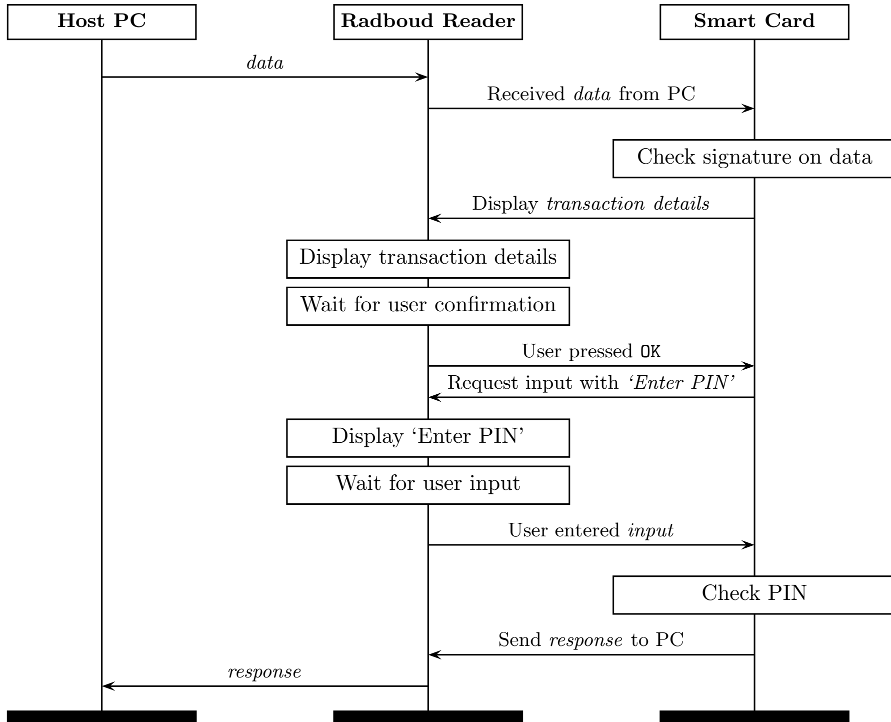


Figure 6.3: Typical usage scenario

c) forward data from the smart card back to the PC, and then wait for new data from the PC.

4. Sending user input from the keyboard to the smart card, following 3a) or 3b).

Using these operations the smart card can then construct any interaction with PC and the user that it wants. A typical usage scenario would be that the card receives digitally signed information over the internet and displays this on the device (ensuring that only genuine messages are displayed) and sends out digitally signed responses (to authenticate transactions). Of course, data received and sent can also be encrypted as well as signed to ensure confidentiality. It is up to the smart card to check or add digital signatures and to en- or decrypt.

Figure 6.3 illustrates such a scenario, where the data from the PC is forwarded to the card, who subsequently shows transaction data on the display. If the user agrees with this information, the card requests the PIN code from the user as input. If the PIN code is correct, the card generates a signature which is sent to the reader to be forwarded to the PC.

## 6.5 Detailed design

To implement the functional requirements described above, several decisions have to be taken:

(i) How is the applet selected on the smart card?

(ii) How do we forward data to the smart card, when data is received from the PC (i.e. inPC in Figure 6.2)?

(iii) How is the distinction made between output from the smart card that is (a) destined for the display, (b) a request for input from the user, and (c) output destined for the PC (i.e. between outUser and outPC in Figure 6.2)?

(iv) How is input by the user forwarded to the smart card (i.e. inUser in Figure 6.2)? And how can the card distinguish between data coming from the keyboard and from the PC (i.e. between inPC and inUser)?

These operations have to be realised at the level of the communication between the smart card and the reader, as laid down in the ISO/IEC 7816 standard and discussed in Section 2.1.1. The reader will have to present data to the smart card in such a way that the smart card can distinguish between inPC and inUser in Figure 6.2. Similarly, the smart card will have to provide its response in such a way that the reader can distinguish between options 3a), 3b) and 3c) listed earlier. Because we want the smart card to be in control as much as possible, these responses from the smart card will in fact be instructions for the Radboud reader. To avoid confusion with the standard terminology of commands for messages from the reader to the smart card, we will call such messages from the smart card to the reader *instructions* and not commands.

**Selecting an application on the smart card** Selecting the desired application could be done in several ways: by fixing a unique AID that is selected and hard-coding this in the reader, by hard-coding a list of AIDs that the reader attempts to select, or by letting the reader select the default applet. Letting the PC choose the applet to select is of course not a good option, as it could be abused by malware on the PC.

For simplicity, we choose to use a fixed AID to select the application on the smart card. We choose an AID here that is not already used for an existing standard application. As soon as the smart card is inserted in the reader, the application is selected.

**Forwarding PC communications to the smart card** One security-critical piece of functionality of the device is to provide a kind of firewall to shield the smart card from malicious actions by the PC. One way of doing this would be to block all traffic except a minimal white-list or a single instruction. Another would be to prepend data sent to the smart card with a fixed prefix.

The former approach is a classical method for firewalls. The PC can send APDUs to the reader, who will forward them to the card only when they are allowed according to its rules. These rules would have to be fixed for all possible applications, as they are stored in the reader. (One could make this APDU firewall configurable, with configuration controlled by the smart card, but this introduces a lot of complexity.) Therefore it needs to be decided in advance what APDUs might be necessary and can be allowed to pass through.

We choose the latter approach, where the data received from the PC is wrapped as payload in an APDU with a dedicated instruction. Using this approach, the reader does not have to process the data it receives in any way: it simply forwards communication received from the terminal with a fixed prefix.

This approach does not work with existing smart card applications, but it is possible to reuse these applications with only minor modifications. However, this should be done with care, since just unwrapping the APDU received by the smart card and executing it without any checking of the operation poses a serious security risk.

We are free to choose the prefix, but we should make sure this prefix does not have a meaning already in the ISO/IEC 7816 standard. Otherwise the reader could accidentally trigger functionality in a smart card that was not designed to be used with the USB reader.

Ideally, the display and keyboard are disconnected before USB connection is connected, and only reconnected after the USB connection has been disconnected, to avoid any way for the USB connection to influence or observe the display or keyboard.

### 6.5.1 Format of data sent to the smart card

As explained above, the reader will simply forward data from the PC to the smart card – i.e. inPC in Figure 6.2 – with a fixed prefix. For user input on the device – inUser– the reader will use a different prefix, so that the card can distinguish between them. Figure 6.4 gives the data formats for this, which use the constants `INS_DATA` and `INS_USER_INPUT` as the instruction byte (the second byte of the command APDUs):

- `INS_DATA` indicates that the APDU contains data forwarded from the PC. The data that is received from the PC is wrapped in an APDU and forwarded as it is.

- `INS_USER_INPUT` indicates that the APDU contains user input entered on the keyboard. The first byte of the data indicates whether the 'OK' or 'Cancel' button was pressed and, in case that 'OK' was pressed, the rest of the data gives the user input.

| CLA | INS_DATA | 00 | 00 | Le | $data_0$ | $data_1$ | ··· |
|---|---|---|---|---|---|---|---|

| CLA | INS_USER_INPUT | 00 | 00 | Le | OK / Cancel | $input_0$ | $input_1$ | ··· |
|---|---|---|---|---|---|---|---|---|

Figure 6.4: Data formats for the Command APDUs to pass PC data and user input to the card.

### 6.5.2 Format of data sent by the smart card

The three instructions that the smart card can give to the reader use the data formats presented in Figure 6.5, where the first byte specifies the instruction:

- `RESP_DISPLAY` instructs the reader to display the text returned by the smart card. The text to be displayed should not be longer than 80 characters for our prototype and is supplied after the `RESP_DISPLAY` instruction. When displaying data following a `RESP_DISPLAY` instruction, the reader waits for the user to either press 'OK' or 'Cancel'. This input is then sent to the smart card using the `INS_USER_INPUT` command.
- `RESP_REQUEST_INPUT` instructs the reader to request user input. The second byte indicates whether input should be echoed, i.e. if the user can see the input on the screen or only masked characters. The third byte indicates the maximum input length. A string is appended to this that will be shown before the input. The length of this string together with the maximum input length cannot be longer than 80 characters for our prototype. After receiving a `RESP_REQUEST_INPUT` instruction, the reader lets the user input data using the keypad. The reader uses a `INS_USER_INPUT` instruction to return the result to the smart card.
- `RESP_DATA` instructs the reader to forward the data returned by the smart card to the PC and wait for new data from the PC.

After each command sent to the smart card, the card can respond with a new instruction, making it possible to perform multiple `RESP_DISPLAY` and `RESP_REQUEST_INPUT` instructions before finally returning data to the PC using the `RESP_DATA` instruction.

## 6.6 Prototype reader

To show the functionality of the Radboud Reader, we constructed a prototype using the Arduino platform[1] (see Figure 6.6). The code is approximately 350 lines and the components cost around 50 Euro. This prototype demonstrates the feasibility of the

[1] http://www.arduino.cc

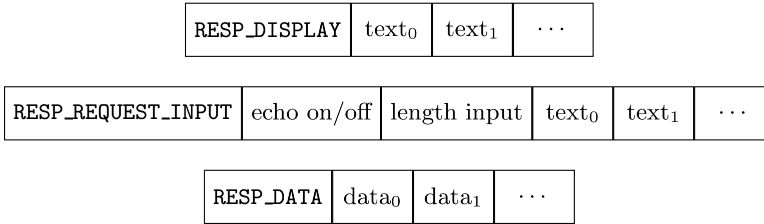


Figure 6.5: Data formats for the Response APDUs that provide instructions of the smart card to the reader

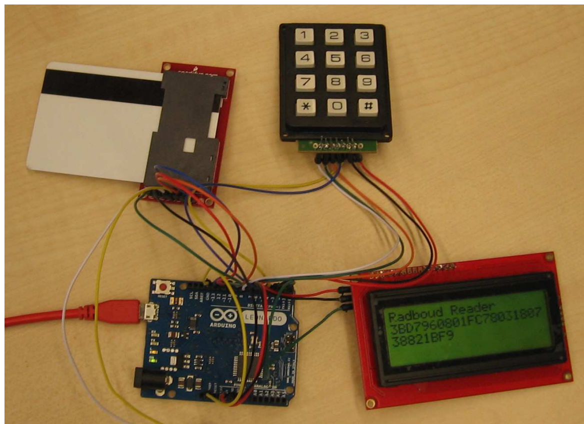


Figure 6.6: Prototype based on Arduino platform

approach and developing it was a useful exercise to make sure we resolved all implementation and design choices that have to be made in realising an implementation. Using this prototype protocols can be developed and tested.

We implemented one sample protocol on an actual smart card. The protocol provides integrity and confidentiality of communication between the smart card and the issuer. For this, the smart card contains a symmetric key, that is shared with the issuer, an asymmetric key pair and the public key of the issuer. The issuer knows the shared symmetric key, the public key of the smart card and its own asymmetric key pair.

The issuer starts the protocol by sending a command to initialise the protocol. The smart card responds by sending a nonce and his identity, encrypted using the public key of the issuer. In response the issuer sends the smart card's nonce together with its own nonce, both encrypted using the public key of the smart card, back. The smart card will then generate a random session key, which it encrypts using its shared symmetric key. To ensure integrity of the session key, as it is a random string, a MAC is computed over the encryption.

After the session key is established, so-called Secure Messaging is used to provide confidentiality and integrity of the communication session between the smart card and issuer, as in an Secure Shell (SSH) or TLS tunnel. The Session Sequence Counter (SSC), that is used to prevent replay, is initialised to the concatenation of the nonces of the issuer and the smart card. With Secure Messaging, first the data is encrypted using the session key. Subsequently a MAC is computed over the encrypted data. To prevent replays, the first block that is processed in the computation of the MAC is the SSC, whose value is increased by one after each message.

## 6.7 Differences with existing devices

To compare our approach with existing devices, several security features can be taken into consideration:

1. The device may guarantee What-You-See-Is-What-You-Sign (i.e. guarantee that the transaction details shown on the display are in fact what is signed), but it may also provide a trusted display that can guarantee that any data shown on the display is authentic (i.e. originates from the service provider).
2. The device can support cryptographic operations, increasing the complexity of the device.
3. The device can include secrets, for instance in the form of secret keys (in case that the device can do cryptographic operations). But it is also possible that the functionality of the device is (partly) secret, in which case we may not be able to tell if it contains say a secret key or if it simply contains some unknown use of a hash function.
4. The device may be unconnected, and not provide any means of communication between it and the PC, or, in case it does, this communication may be bidirectional (e.g., in the case of USB), or uni-directional (e.g., in the case of optic communication from the PC to the device).
5. Finally, there is the question of how generic the device is, i.e. whether the device can be used for several purposes and users.

For some of these characteristics one can still argue whether they improve or weaken security. E.g., having secrets in the device makes it harder to produce a fake version or a software implementation of the devices, but makes generic use harder.

Table 7.2 gives a comparison of different devices based on these features. Below we discuss these devices in more detail.

Manufacturers typically do not publish technical details about the devices they produce, (though they do sometimes apply for patents, e.g. [Gul10]). This means that for some devices we do not know all the features, unless the working has been

reverse-engineered. Ideally vendors would provide this kind of information publicly so the clients can make a better comparison between different devices.

Unconnected smart card readers with a display and keyboard are widely used for online banking. Many of these systems use EMV-CAP, discussed in Section 5.1. This has advantages, namely that existing EMV smart card implementations, which may have undergone costly security certifications, can be re-used, but also introduces the risk of ambiguities in different meanings of the same protocol messages [DMA09].

Some banks already use a USB-connected reader for internet banking. Companies supplying solutions for this include VASCO and Gemalto. As far as we know, none of these devices provide a trusted display: they display data that is received over the USB cable in plaintext without any integrity checks. This means that malware on an infected PC could show anything it wants, and could even use the display of the reader as part of a sophisticated phishing attack. Even if these devices do not guarantee that what is shown on the display is authentic in any way, they can provide WYSIWYS by sending the displayed text to the card to be signed.

The vulnerability in the e.dentifier2, discussed in Section 5.3, demonstrates once again the danger in relying on closed, proprietary solutions. It also provides further support for our design philosophy of keeping the device as simple as possible.

The Zone Trusted Information Channel, or ZTIC[2] from IBM comes close to our approach in the security it provides [WKH+08]. The ZTIC is a small USB-connect smart card reader with a small display and allows user input by means of two buttons (for OK and Not OK) and a wheel that can be turned to input numbers. Unlike the Radboud Reader, the ZTIC has cryptographic capabilities and the keys and certificates to set up a secure TLS tunnel between the ZTIC and a remote web server. Every device has its own certificates for mutual authentication with the issuer.

One difference between the ZTIC and our proposal is that the ZTIC is a more complicated device, capable of storing keys and doing crypto. Unlike our solution, which is generic and can be used in conjunction with any smart card with a suitable application, the ZTIC needs to have the certificate for each service provider in order to communicate with it. Another difference, and possible advantage of the ZTIC, is that the common functionality to provide a secure tunnel is provided by the ZTIC, and need not be provided by each smart card used with the Radboud Reader: using the ZTIC it is guaranteed to have a secure tunnel, using the Radboud Reader this still depends on the smart card.

The FINREAD project proposed a standardised trusted card reader [FIN04]. This idea never became a success, probably because the FINREAD card reader would be too expensive and complex; they were meant to be tamper-resistant and included a PKI support for controlling applications on them.

[2]See http://www.zurich.ibm.com/ztic. Originally, ZTIC stood for Zurich Trusted Information Channel.

| | Trusted display | Crypto | Secrets | Connected | Generic |
|---|---|---|---|---|---|
| EMV-CAP reader | ✗ | ✓ | ✗ | ✗ | ✗ |
| e.dentifier2 | ✗ | ✗ | ✓[a] | ↔ | ✗ |
| ZTIC | ✓ | ✓ | ✓[b] | ↔ | ✗ |
| FINREAD | ✓ | ✓ | ✓[b] | ↔ | ✓ |
| AGSES | ? | ✓ | ✓[b,c] | → | ✗ |
| Radboud Reader | ✓ | ✗ | ✗ | ↔ | ✓ |

[a] Unknown functionality
[b] Secret key
[c] Uses fingerprints rather than PIN code

Table 6.1: Comparison with existing readers

The AGSES card reader[3] does not use a USB connection, but receives data via a flickering barcode on the PC screen. There is no communication back from this reader to the PC, so the user has to manually type in the response again. We do not know if the data sent using the flickering bar code is checked for authenticity before it is displayed.

Nowadays, smart cards with integrated keyboard and display are also commercially available. These displays are however very limited, for example, the smart card offered by NagraID[4] only contains 6 characters.

A solution without even a reader is mTAN. Here the user receives an SMS on his mobile phone with transaction details and a code to confirm it. Here the phone could be seen as a trusted display. However, as mobile phones become more and more complex, they are now becoming popular targets for malware and attacks just like PCs.

## 6.8 Conclusions

The number and importance of online transactions is rapidly growing, and protecting such transactions in the face of ever more sophisticated malware is a serious challenge. This is not only an issue in online banking, but also in e-government services, or indeed any online transactions where one would really want the online equivalent of a handwritten signature.

[3]http://www.agses.net
[4]http://www.nagraid.com

In this chapter the design of a simple and generic device for securing online transactions was presented, which protects against any malware on the PC, including Man-in-the-Browser attacks. The device provides a trusted communication channel between a user and any remote service provider, by means of a smart card issued by that provider.

The essence of our solution, as illustrated in Figure 6.2, is quite simple: namely, make sure that all communication with the user passes through the smart card. We are not aware of any solutions that use this approach, even though conceptually it is quite simple. This solution allows a very simple device, which does not need any cryptographic capabilities or need not store any keys or other secrets.

Because there are no secrets in the Radboud Reader, e.g. in the form of secret keys or secret protocols, anyone can make one. This can be considered a disadvantage (an attacker could make or market fake devices) but also an advantage, as anyone can implement their own device. Note that there is little incentive for manufacturers of smart card readers to come up with solution like ours, where there is no intellectual property or secret in the device, thus allowing anyone to manufacture it and not having any risk of vendor lock-in.

As discussed in detail in Section 6.7, the Radboud Reader offers stronger security than many existing USB-connected smart card readers with display and keyboard used for internet banking, except IBM's ZTIC, as these solutions do not offer provide a trusted display, i.e. they can not guarantee that what appears on the display is an authentic message from the remote server.

Unlike other solutions, our solution is completely generic. The functionality hardcoded in the reader only provides some basic building blocks and the smart card is in charge of using these to build the scenario that some service requires. So the same device can be used by different smart cards for different purposes. As the number of online services that require a high-security solution to secure transactions increases, investing in a single reader that can be used for all of them may prove a practical and economical solution. It may then also become an option to go for a slightly more expensive device, with a larger display; one could even imagine an e-reader for digitally signing electronic documents that operates in the same way as the Radboud Reader, though the limited bandwidth of communication with the smart card could then become a bottleneck.

# Chapter 7

# Analysing TLS

In this chapter we discuss the analysis of TLS implementations using protocol state fuzzing, making use of LearnLib for the inference of the state machines. This chapter is based on [RP15].

TLS, short for Transport Layer Security, is widely used to secure network connections, for example in HTTPS. Being one of the most widely used security protocols, TLS has been the subject of a lot of research and many issues have been identified. These range from cryptographic attacks (such as problems when using RC4 [ABP+13]) to serious implementation bugs (such as Heartbleed [Cod]) and timing attacks (for example, Lucky Thirteen and variations of the Bleichenbacher attack [AP13, MSW+14, Ble98]).

## 7.1 Related work

Various formal methods have been used to analyse different parts and properties of the TLS protocol [Pau99, DCVP04, HSD+05, OF05, GMP+08, MSW10, KL11, JKSS12, KPW13]. However, these analyses look at abstract descriptions of TLS, not actual implementations, and in practice many security problems with TLS have been due to mistakes in implementation [MS14]. To bridge the gap between the specification and implementation, formally verified TLS implementations have been proposed [BFCZ08, BFK+13].

Existing tools to analyse TLS implementations mainly focus on fuzzing of individual messages, in particular the certificates that are used. These certificates have been the source of numerous security problems in the past. An automated approach to test for vulnerabilities in the processing of certificates is using Frankencerts as proposed by Brubaker et al. [BJR+14] or using the tool x509test[1]. Fuzzing of individual messages is orthogonal to the technique we propose as it targets different parts or aspects of the code. However, the results of our analysis could be used to guide fuzzing of messages by indicating protocol states that might be interesting places to start fuzzing messages.

Another category of tools analyses implementations by looking at the particular

[1] https://github.com/yymax/x509test

configuration that is used. Examples of this are the SSL Server Test[2] and sslmap[3].

Finally, closely related research on the implementation of state machines for TLS was done by Beurdouche et al. [BBFK+15]. We compare their work with ours in Section 7.6.

## 7.2 The TLS protocol

The TLS protocol was originally known as Secure Socket Layer (SSL), which was developed at Netscape. SSL 1.0 was never released and version 2.0 contained numerous security flaws [TP11]. This lead to the development of SSL 3.0, on which all later versions are based. After SSL 3.0, the name was changed to TLS and currently three versions are published: 1.0, 1.1 and 1.2 [DA99, DR06, DR08]. The specifications for these versions are published in RFCs issued by the Internet Engineering Task Force (IETF).

To establish a secure connection, different subprotocols are used within TLS:

- The *Handshake* protocol is used to establish session keys and parameters and to optionally authenticate the server and/or client.
- The *ChangeCipherSpec* protocol – consisting of only one message – is used to indicate the start of the use of established session keys.
- To indicate errors or notifications, the *Alert* protocol is used to send the level of the alert (either warning or fatal) and a one byte description.

In Figure 7.1 a normal flow for a TLS session is given. In the ClientHello message, the client indicates the desired TLS version, supported cipher suites and optional extensions. A cipher suite is a combination of algorithms used for the key exchange, encryption, and MAC computation. During the key exchange a premaster secret is established. This premaster secret is used in combination with random values from both the client and server to derive the master secret. This master secret is then used to derive the actual keys that are used for encryption and MAC computation. Different keys are used for messages from the client to the server and for messages in the opposite direction. Optionally, the key exchange can be followed by client verification where the client proves it knows the private key corresponding to the public key in the certificate it presents to the server. After the key exchange and optional client verification, a ChangeCipherSpec message is used to indicate that from that point on the agreed keys will be used to encrypt all messages and add a MAC to them. The Finished message is finally used to conclude the handshake phase. It contains a keyed hash, computed using the master secret, of all previously exchanged handshake messages. Since it is sent after the ChangeCipherSpec message

[2]https://www.ssllabs.com/ssltest/
[3]https://www.thesprawl.org/projects/sslmap/

it is the first message that is encrypted and MACed. After the handshake phase, application data can be exchanged over the established secure channel.

To add additional functionality, TLS offers the possibility to add extensions to the protocol. One example of such an extension is the – due to Heartbleed [Cod] by now well-known – Heartbeat Extension, which can be used to keep a connection alive using HeartbeatRequest and HeartbeatResponse messages [STW12].

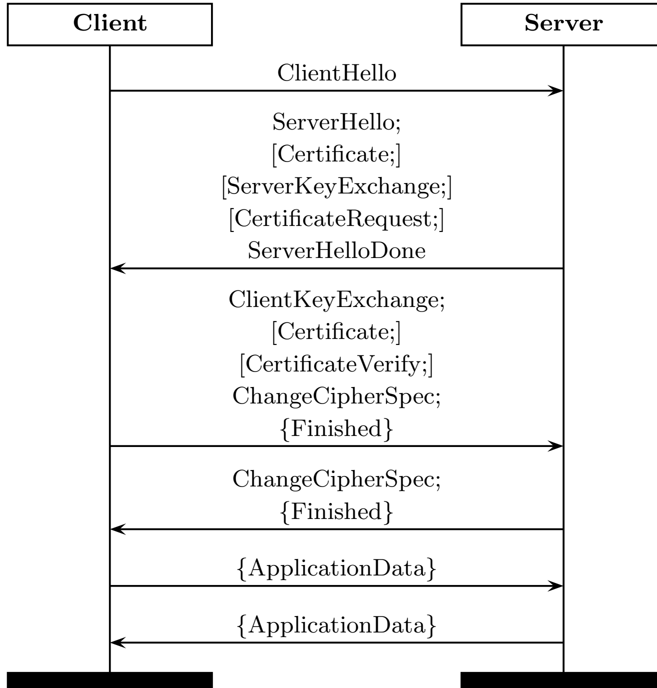


Figure 7.1: A regular TLS session. An encrypted message m is denoted as {m}. If message m is optional, this is indicated by [m].

## 7.3 Learning

To infer the state machines of implementations of the TLS protocol we used LearnLib. For the equivalence checking we use an improved version of Chow's W-method [Cho78]. The W-method is guaranteed to be correct given an upper bound for the number of states. For LearnLib we can specify a depth for the equivalence checking: given a hypothesis for the state machine, the upper bound for the W-method is set to the number of found states plus the specified depth. The W-method is very powerful but comes at a high cost in terms of performance. Therefore we improved the algorithm to take advantage of a property of the system we learn, namely that once a

connection is closed, all outputs returned afterwards will be the same (namely *Connection closed*). So when looking for counterexamples, extending a trial trace that results in the connection being closed is pointless. The W-method, however, will still look for counterexamples by extending traces which result in a closed connection. We improved the W-method by adding a check to see if it makes sense to continue searching for counterexamples with a particular prefix, and for this we simply check if the connection has not been closed. This simple modification of the W-method greatly reduced the number of equivalence queries needed, as we will see in Section 7.5.

## 7.4 Test harness

To use LearnLib, we need to fix an input alphabet of messages that can be sent to the SUT. This alphabet is an abstraction of the actual messages sent. In our analyses we use different input alphabets depending on whether we test a client or server, and whether we perform a more limited or more extensive analysis. To test servers we support the following messages: ClientHello (RSA and DHE), Certificate (RSA and empty), ClientKeyExchange, ClientCertificateVerify, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest and HeartbeatResponse. To test clients we support the following messages: ServerHello (RSA and DHE), Certificate (RSA and empty), CertificateRequest, ServerKeyExchange, ServerHelloDone, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest and HeartbeatResponse.

We thus support all regular TLS messages as well as the messages for the Heartbeat Extension. The test harness supports both TLS version 1.2 and, in order to test older implementations, version 1.0. The input alphabet is not fixed, but can be configured per analysis as desired. For the output alphabet we use all the regular TLS messages as well as the messages from the Alert protocol that can be returned. This is extended with some special symbols that correspond with exceptions that can occur in the test harness:

- *Empty*, this is returned if no data is received from the SUT before a timeout occurs in the test harness.
- *Decryption failed*, this is returned if decryption fails in the test harness after a ChangeChipherSpec message was received. This could happen, for example, if not enough data is received, the padding is incorrect after decryption (e.g. because a different key was used for encryption) or the MAC verification fails.
- *Connection closed*, this is returned if a socket exception occurs or the socket is closed.

LearnLib uses these abstract inputs and outputs as labels on the transitions of the state machine. To interact with an actual TLS server or client we need a test harness

that translates the abstract input messages to actual TLS packets and the responses back to abstract responses. As we make use of cryptographic operations in the protocol, we needed to introduce state in our test harness, for instance to keep track of the information used in the key exchange and the actual keys that result from this. Apart from this, the test harness also has to remember whether a ChangeCipherSpec was received or sent, as we have to encrypt and MAC all corresponding data after this message. Note that we only need a single test harness for TLS to then be able to analyse any implementation. Our test harness can be considered a 'stateless' TLS implementation.

When testing a server, the test harness is initialised by sending a ClientHello message to the SUT to retrieve the server's public key and preferred ciphersuite. When a reset command is received we set the internal variables to these values. This is done to prevent null pointer exceptions that could otherwise occur when messages are sent in the wrong order.

After sending a message the test harness waits to receive responses from the SUT. As the SUT will not always send a response, for example because it may be waiting for a next message, the test harness will generate a timeout after a fixed period. Some implementations require longer timeouts as they can be slower in responding. As the timeout has a significant impact on the total running time we varied this per implementation.

To test client implementations we need to launch a client for every test sequence. This is done automatically by the test harness upon receiving the *reset* command. The test harness then waits to receive the ClientHello message, after which the client is ready to receive a query. Because the first ClientHello is received before any query is issued, this message does not appear explicitly in the learned models.

## 7.5 Results

We analysed the nine different implementations listed in Table 7.1. We used demo client and server applications that came with the different implementations except with the Java Secure Socket Extension (JSSE). For JSSE we wrote simple server and client applications. For the implementations listed the models of the server-side were learned using our modified W-method for the following alphabet: ClientHello (RSA), Certificate (empty), ClientKeyExchange, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest. For completeness we learned models for both TLS version 1.0 and 1.2, when available, but this always resulted in the same model.

Due to space limitations we cannot include the models for all nine implementations in this paper, but we do include the models in which we found security issues (for GnuTLS, Java Secure Socket Extension, and OpenSSL), and the model of RSA BSAFE for Java to illustrate how much simpler the state machine can be. The models

| Name | Version | URL |
|---|---|---|
| GnuTLS | 3.3.8<br>3.3.12 | http://www.gnutls.org/ |
| Java Secure Socket Extension (JSSE) | 1.8.0_25<br>1.8.0_31 | http://www.oracle.com/java/ |
| mbed TLS (previously PolarSSL) | 1.3.10 | https://polarssl.org/ |
| miTLS | 0.1.3 | http://www.mitls.org/ |
| RSA BSAFE for C | 4.0.4 | http://www.emc.com/security/rsa-bsafe.htm |
| RSA BSAFE for Java | 6.1.1 | http://www.emc.com/security/rsa-bsafe.htm |
| Network Security Services (NSS) | 3.17.4 | https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS |
| OpenSSL | 1.0.1g<br>1.0.1j<br>1.0.1l<br>1.0.2 | https://www.openssl.org/ |
| nqsb-TLS | 0.4.0 | https://github.com/mirleft/ocaml-tls |

Table 7.1: Tested implementations

| **Alphabet** | **Algorithm** | **Time (hh:mm)** | **#states** | **Membership queries** | **Equivalence queries** |
|---|---|---|---|---|---|
| regular | modified W-method | 0:09 | 7 | 456 | 1347 |
| full | modified W-method | 0:27 | 9 | 1573 | 4126 |
| full | original W-method | 4:09 | 9 | 1573 | 68578 |

Table 7.2: Analysis of the GnuTLS 3.3.12 server using different alphabets and equivalence algorithms

and the code of our test harness. We wrote a Python application to automatically simplify the models by combining transitions with the same responses and replacing the abstract input and output symbols with more readable names. Table 7.3 shows the times needed to obtain these state machines, which ranged from about 9 minutes to over 8 hours.

A comparison between our modified equivalence algorithm and the original W-method can be found in Table 7.2. This comparison is based on the analysis of GnuTLS 3.3.12 running a TLS server. It is clear that by taking advantage of the state of the socket our algorithm performs much better than the original W-method: the number of equivalence queries is over 15 times smaller for our method when learning a model for the server.

When analysing a model, we first manually look if there are more paths than expected that lead to a successful exchange of application data. Next we determine whether the model contains more states than necessary and identify unexpected or superfluous transitions. We also check for transitions that can indicate interesting behaviour such as, for example, a 'Bad record MAC' alert or a *Decryption failed* message. If we come across any unexpected behaviour, we perform a more in-depth analysis to determine the cause and severity.

An obvious first observation is that all the models of server-side implementations are very different. For example, note the huge difference between the models learned for RSA BSAFE for Java in Figure 7.9 and for OpenSSL in Figure 7.11. Because all the models are different, they provide a unique fingerprint of each implementation, which could be used to remotely identify the implementation that a particular server is using.

Most demo applications close the connection after their first response to application data. In the models there is then only one ApplicationData transition where ap-

| | #states | Timeout | Time (h:mm) | #membership queries | #equivalence queries |
|---|---|---|---|---|---|
| GnuTLS 3.3.8 | 12 | 100ms | 0:45 | 1370 | 5613 |
| GnuTLS 3.3.12 | 7 | 100ms | 0:09 | 456 | 1347 |
| mbed TLS 1.3.10 | 8 | 100ms | 0:39 | 520 | 2939 |
| OpenSSL 1.0.1g + | 16 | 100ms | 0:31 | 1016 | 4171 |
| OpenSSL 1.0.1j + | 11 | 100ms | 0:16 | 680 | 2348 |
| OpenSSL 1.0.1l + | 10 | 100ms | 0:14 | 624 | 2249 |
| OpenSSL 1.0.2 + | 7 | 100ms | 0:06 | 350 | 902 |
| JSSE 1.8.0_25 | 9 | 200ms | 0:41 | 584 | 2458 |
| JSSE 1.8.0_31 | 9 | 200ms | 0:39 | 584 | 2176 |
| miTLS 0.1.3 | 6 | 1500ms | 0:53 | 392 | 517 |
| NSS 3.17.4 | 8 | 500ms | 3:16 | 520 | 5329 |
| RSA BSAFE for Java 6.1.1 | 6 | 500ms | 0:18 | 392 | 517 |
| RSA BSAFE for C 4.0.4 | 9 | 200ms | 8:16 | 584 | 26353 |
| nqsb-TLS 0.4.0 + | 8 | 100ms | 0:15 | 399 | 1835 |

+ Without heartbeat extension

Table 7.3: Results of the automated analysis of server implementations for the regular alphabet of inputs using our modified W-method with depth 2

plication data is exchanged instead of the expected cycle consisting of an Application-Data transition that allows server and client to continue exchanging application data after a successful handshake.

In the subsections below we discuss the peculiarities of models we learned, and the flaws they revealed. Correct paths leading to an exchange of application data are indicated by thick green transitions in the models. If there is any additional path leading to the exchange of application data this is a security flaw and indicated by a dashed red transition.

### 7.5.1 GnuTLS

Figure 7.2 shows the model that was learned for GnuTLS 3.3.8. In this model there are two paths leading to a successful exchange of application data: the regular one

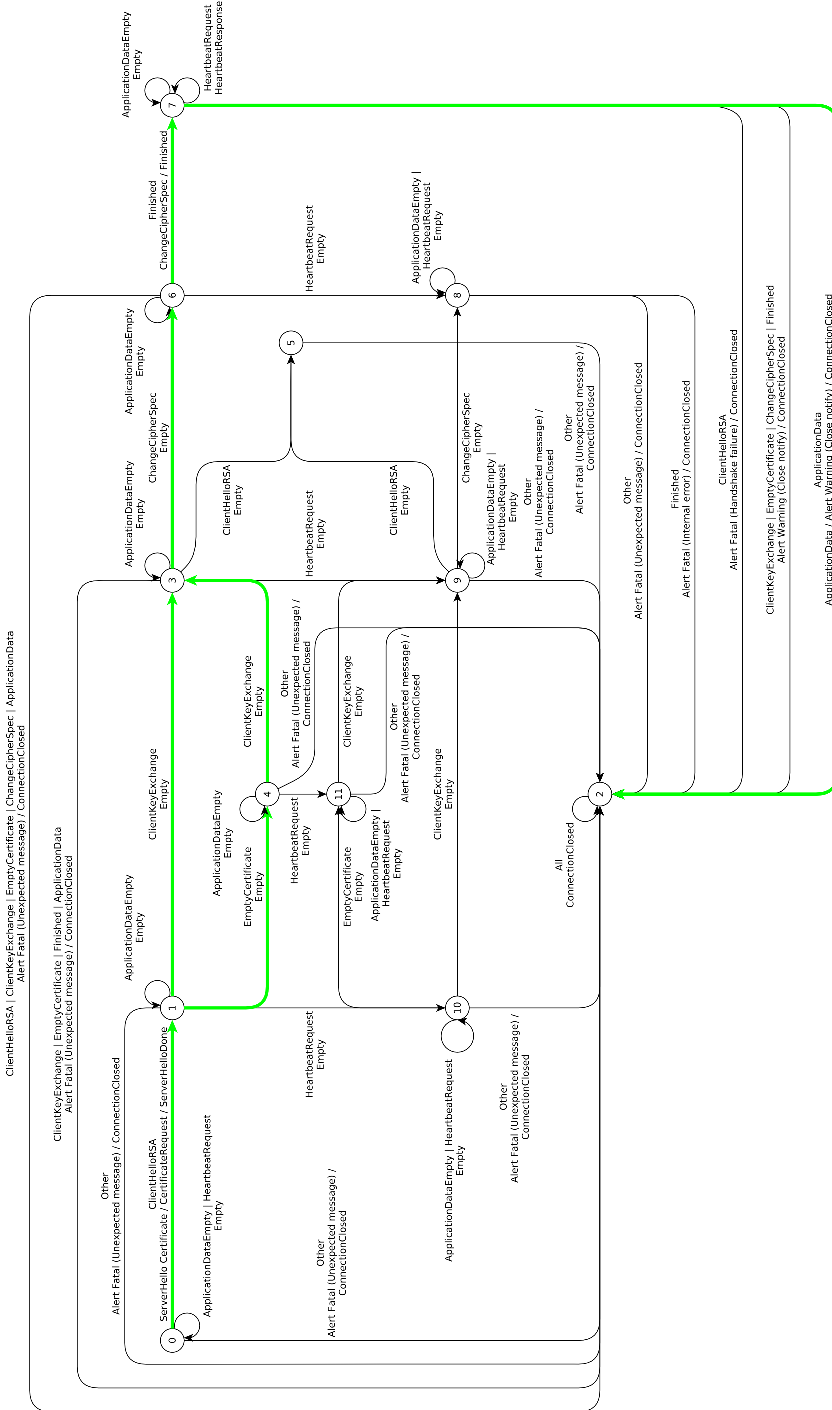


Figure 7.2: Learned state machine model for GnuTLS 3.3.8

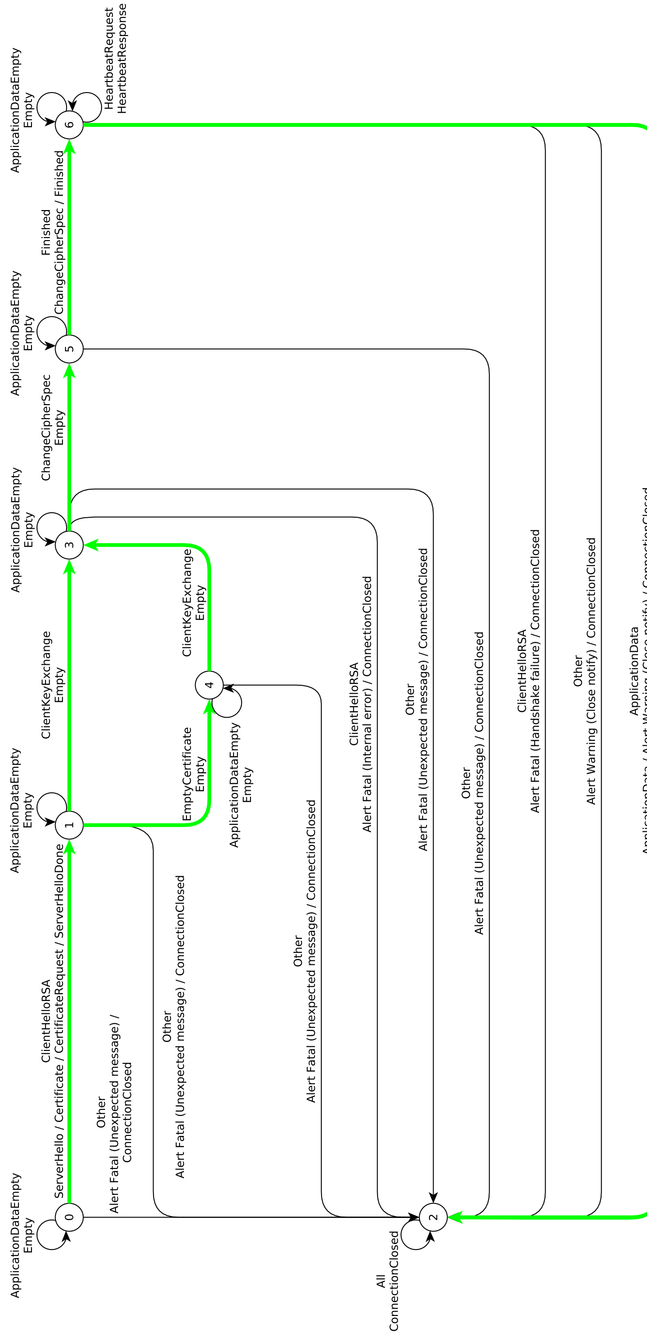


Figure 7.3: Learned state machine model for GnuTLS 3.3.12. A comparison with the model for GnuTLS 3.3.8 in Figure 7.2 shows that the superflous states (8, 9, 10, and 11) are now gone, confirming that the code has been improved.

without client authentication and one where an empty client certificate is sent during the handshake. As we did not require client authentication, both are acceptable paths. What is immediately clear is that there are more states than expected. Closer inspection reveals that there is a 'shadow' path, which is entered by sending a HeartbeatRequest message during the handshake protocol. The handshake protocol then does proceed, but eventually results in a fatal alert ('Internal error') in response to the Finished message (from state *8*). From every state in the handshake protocol it is possible to go to a corresponding state in the 'shadow' path by sending the HeartbeatRequest message. This behaviour is introduced by a security bug, which we will discuss below. Additionally there is an redundant state *5*, which is reached from states *3* and *9* when a ClientHello message is sent. From state *5* a fatal alert is given to all subsequent messages that are sent. One would expect to already receive an error message in response to the ClientHello message itself.

**Forgetting the buffer in a heartbeat** As mentioned above, HeartbeatRequest messages are not just ignored in the handshake protocol but cause some side effect: sending a HeartbeatRequest during the handshake protocol will cause the implementation to return an alert message in response to the Finished message that terminates the handshake. Further inspection of the code revealed the cause: the implementation uses a buffer to collect all handshake messages in order to compute a hash over these messages when the handshake is completed, but this buffer is reset upon receiving the heartbeat message. The alert is then sent because the hashes computed by server and client no longer match.

This bug can be exploited to effectively bypass the integrity check that relies on comparing the keyed hashes of the messages in the handshake: when also resetting this buffer on the client side (i.e. our test harness) at the same time we were able to successfully complete the handshake protocol, but then no integrity guarantee is provided on the previous handshake messages that were exchanged.

By learning the state machine of a GnuTLS client we confirmed that the same problem exists when using GnuTLS as a client.

This problem was reported to the developers of GnuTLS and is fixed in version 3.3.9. By learning models of newer versions, we could confirm the issue is no longer present, as can be seen in Figure 7.3.

To exploit this problem both sides would need to reset the buffer at the same time. This might be hard to achieve as at any time either one of the two parties is computing a response, at which point it will not process any incoming message. If an attacker would successfully succeed to exploit this issue no integrity would be provided on any message sent before, meaning a fallback attack would be possible, for example to an older TLS version or weaker cipher suite.

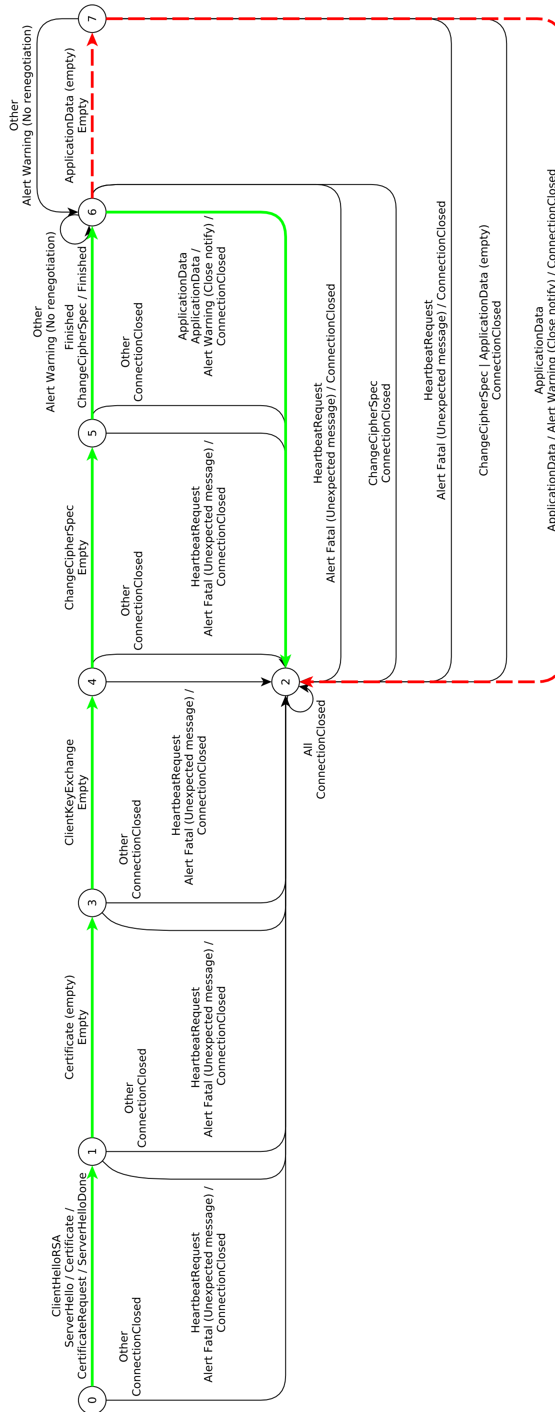


Figure 7.4: Learned state machine model for mbed TLS 1.3.10

### 7.5.2 mbed TLS

For mbed TLS, previously known as PolarSSL, we tested version 1.3.10 (see Figure 7.4). We saw several paths leading to a successful exchange of data. Instead of sending a regular ApplicationData message, it is possible to first send one empty ApplicationData message after which it is still possible to send the regular ApplicationData message. Sending two empty ApplicationData messages directly after each other will close the connection. However, if in between these message an unexpected handshake message is sent, the connection will not be closed and only a warning is returned. After this it is also still possible to send a regular ApplicationData message. While this is strange behaviour, it does not seem to be exploitable.

### 7.5.3 Java Secure Socket Extension

For Java Secure Socket Extension we analysed Java version 1.8.0_25. The model contains several paths leading to a successful exchange of application data and contains more states than expected (see Figure 7.5). This is the result of a security issue which we will discuss below.

As long as no Finished message has been sent it is apparently possible to keep renegotiating. After sending a ClientKeyExchange, other ClientHello messages are accepted as long as they are eventually followed by another ClientKeyExchange message. If no ClientKeyExchange message was sent since the last ChangeCipherSpec, a ChangeCipherSpec message will result in an error (state *7*). Otherwise it either leads to an error state if sent directly after a ClientHello (state *8*) or a successful change of keys after a ClientKeyExchange.

**Accepting plaintext data** More interesting is that the model contains *two* paths leading to the exchange of application data. One of these is a regular TLS protocol run, but in the second path the ChangeCipherSpec message from the client is omitted. Despite the server not receiving a ChangeCipherSpec message it still responds with a ChangeCipherSpec message to a plaintext Finished message by the client. As a result the server will send its data encrypted, but it expects data from the client to be unencrypted. A similar problem occurs when trying to negotiate new keys. By skipping the ChangeCipherSpec message and just sending the Finished message the server will start to use the new keys, whereas the client needs to continue to use its old keys.

This bug invalidates any assumption of integrity or confidentiality of data sent to the server, as it can be tricked into accepting plaintext data. To exploit this issue it is, for example, possible to include this behaviour in a rogue library. As the attack is transparent to applications using the connection, both the client and server application would think they talk on a secure connection, where in reality anyone on the line could read the client's data and tamper with it. Figure 7.6 shows a protocol run where this bug is triggered. The bug was report to Oracle and is identified by

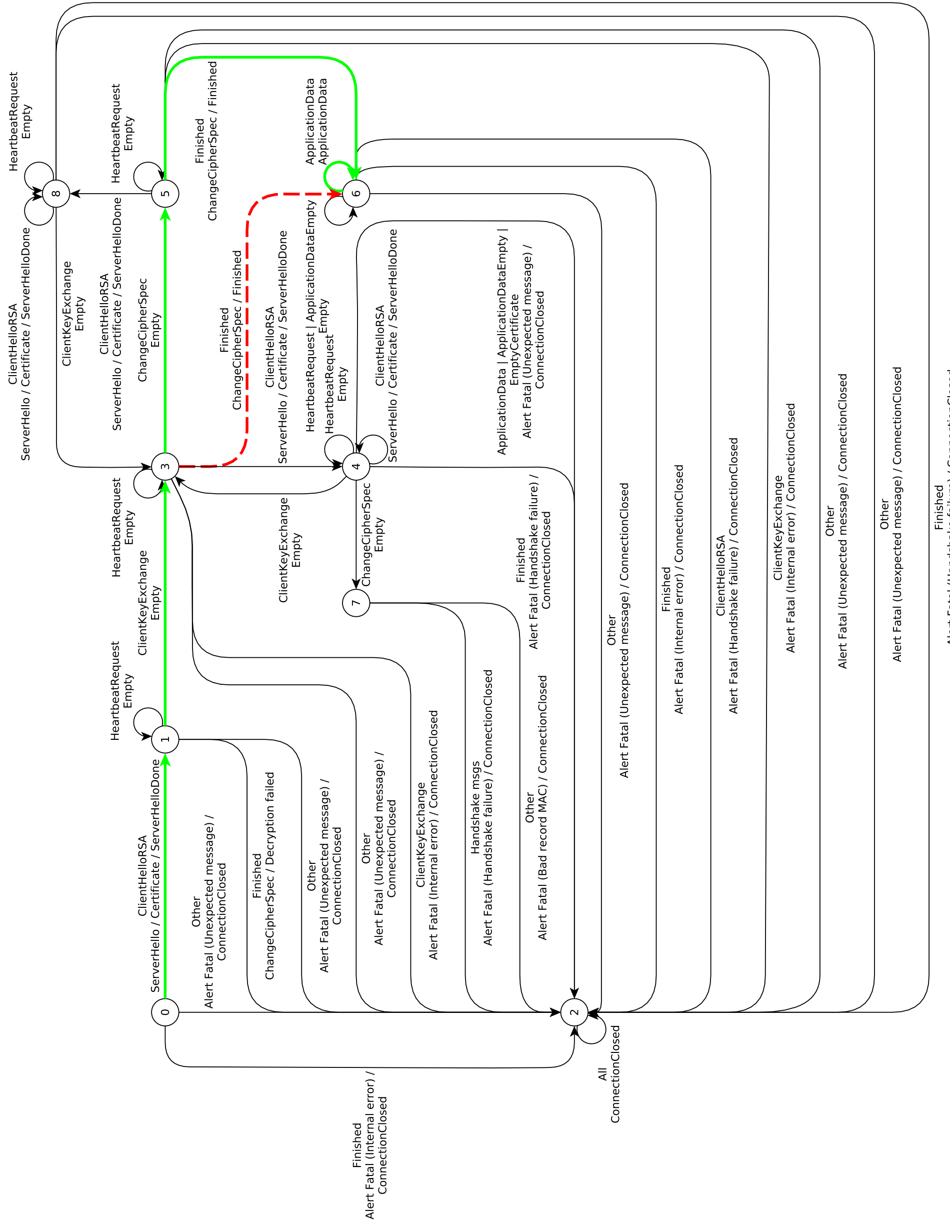


Figure 7.5: Learned state machine model for JSSE 1.8.0_25

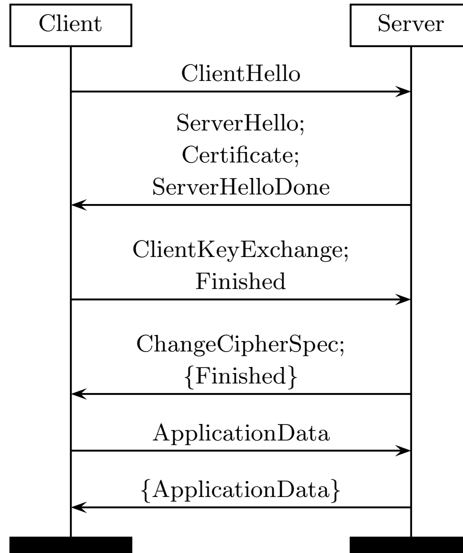


Figure 7.6: A protocol run triggering a bug in the JSSE, causing the server to accept plaintext application data.

CVE-2014-6593. A fix was released in their Critical Security Update in January 2015. By analysing JSSE version 1.8.0_31 we are able to confirm the issue was indeed fixed.

This issue was identified in parallel by Beurdouche et al. [BBFK+15], who also reported the same and a related issue for the client-side. By learning the client, we could confirm that the issue was also present there. Moreover, after receiving the ServerHello message, the client would accept the Finish message and start exchanging application data at any point during the handshake protocol. This makes it possible to completely circumvent both server authentication and the confidentiality and integrity of the data being exchanged.

### 7.5.4 miTLS

MiTLS is a formally verified TLS implementation written in F# [BFK+13]. For miTLS 0.1.3, initially our test harness had problems to successfully complete the handshake protocol and the responses seemed to be non-deterministic because sometimes a response was delayed and appeared to be received in return to the next message. To solve this, the timeout had to be increased considerably when waiting for incoming messages to not miss any message. This means that compared to the other implementations, miTLS was relatively slow in our setup. Additionally, miTLS requires the Secure Renegotiation extension to be enabled in the ClientHello message. The learned model looks very clean with only one path leading to an exchange of application data and does not contain more states than expected (see Figure 7.7).

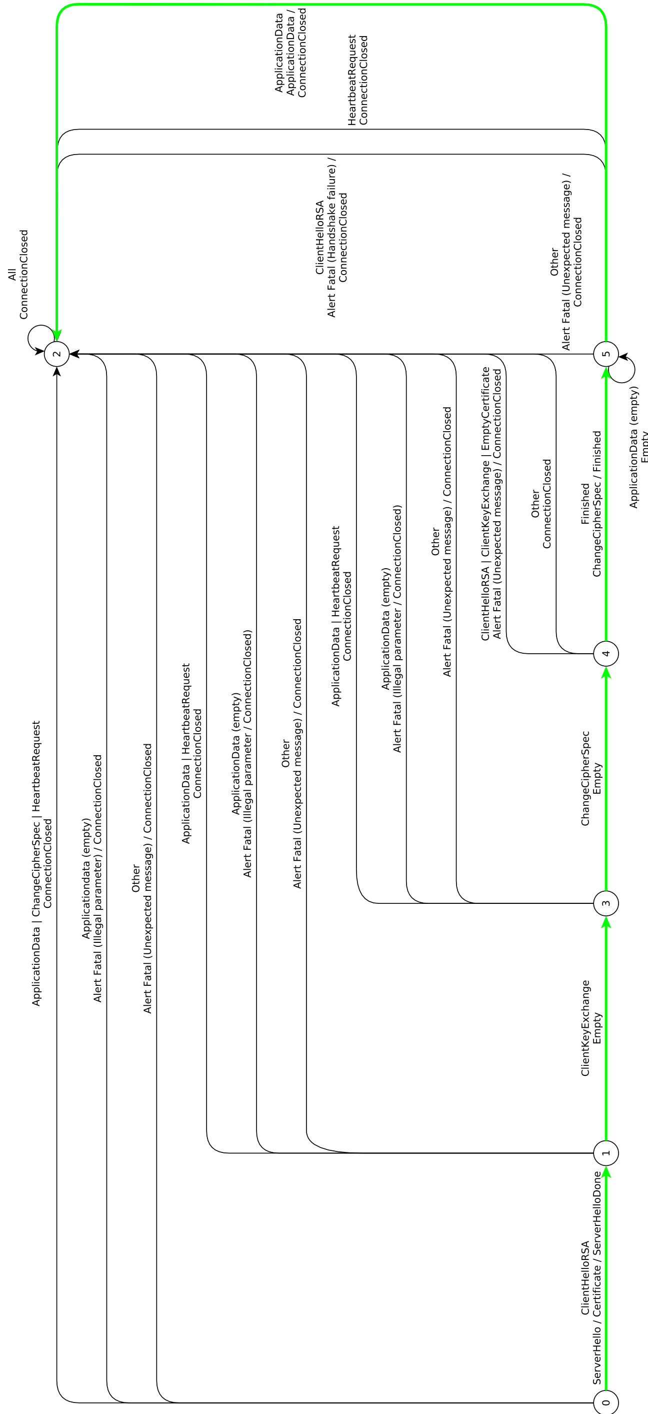


Figure 7.7: Learned state machine model for miTLS 0.1.3

### 7.5.5 RSA BSAFE for C

The RSA BSAFE for C 4.0.4 library resulted in a model containing two paths leading to the exchange application data (see Figure 7.8). The only difference between the paths is that an empty ApplicationData is sent in the second path. However, the alerts that are sent are not very consistent as they differ depending on the state and message. For example, sending a ChangeCipherSpec message after an initial ClientHello results in a fatal alert with reason 'Illegal parameter', whereas application data results in a fatal alert with 'Unexpected message' as reason. More curious however is a fatal alert 'Bad record MAC' that is returned to certain messages after the server received the ChangeCipherSpec in a regular handshake. As this alert is only returned in response to certain messages, while other messages are answered with an 'Unexpected message' alert, the server is apparently able to successfully decrypt and check the MAC on messages. Still, an error is returned that it is not able to do this. This seems to be a non-compliant usage of alert messages.

At the end of the protocol the implementation does not close the connection. This means we cannot take any advantage from a closed connection in our modified W-method and the analysis therefore takes much longer than for the other implementations.

### 7.5.6 RSA BSAFE for Java

The model for RSA BSAFE for Java 6.1.1 library looks very clean, as can be seen in Figure 7.9. The model again contains only one path leading to an exchange of application data and no more states than necessary. In general all received alerts are 'Unexpected message'. The only exception is when a ClientHello is sent after a successful handshake, in which case a 'Handshake failure' is given. This makes sense as the ClientHello message is not correctly formatted for secure renegotiation, which is required in this case. This model is the simplest that we learned during our research.

### 7.5.7 Network Security Services

The model for NSS that was learned for version 3.17.4 looks pretty clean (see Figure 7.10), although there is one more state than one would expect. There is only one path leading to a successful exchange of application data. In general all messages received in states where they are not expected are responded to with a fatal alert ('Unexpected message'). Exceptions to this are the Finished and Heartbeat messages: these are ignored and the connection is closed without any alert. Other exceptions are non-handshake messages sent before the first ClientHello: then the server goes into a state where the connection stays open but nothing happens anymore. Although the TLS specification does not explicitly specify what to do in this case, one would expect the connection to be closed, especially since it's not possible to recover from this. Because the connection is not actually closed in this case the

Figure 7.8: Learned state machine model for RSA BSAFE for C 4.0.4

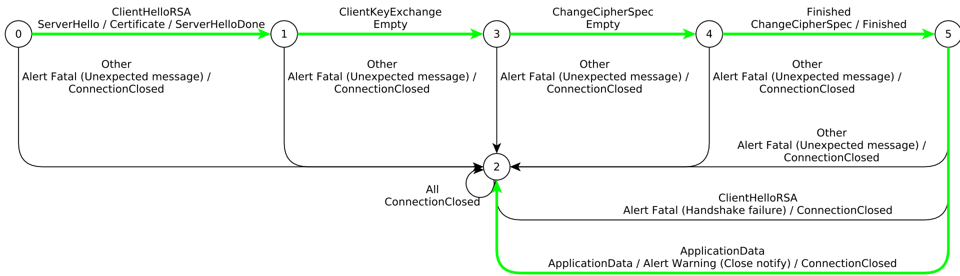


Figure 7.9: Learned state machine model for RSA BSAFE for Java 6.1.1

analysis takes longer, as we have less advantage of our modification of the W-method to decide equivalence.

### 7.5.8 OpenSSL

Figure 7.11 shows the model inferred for OpenSSL 1.01j. In the first run of the analysis it turned out that HeartbeatRequest message sent during the handshake phase were ‘saved up’ and only responded to after the handshake phase was finished. As this results in infinite models we had to remove the heartbeat messages from the input alphabet. This model obtained contains quite a few more states than expected, but does only contain one path to successfully exchange application data.

The model shows that it is possible to start by sending two ClientHello messages, but not more. After the second ClientHello message there is no path to a successful exchange of application data in the model. This is due to the fact that OpenSSL resets the buffer containing the handshake messages every time when sending a ClientHello, whereas our test harness does this only on initialisation of the connection. Therefore, the hash computed by our test harness at the end of the handshake is not accepted and the Finished message in state *9* is responded to with an alert. Which messages are included in the hash differs per implementation: for JSSE all handshake messages since the beginning of the connection are included.

**Re-using keys** In state *8* we see some unexpected behaviour. After successfully completing a handshake, it is possible to send an additional ChangeCipherSpec message after which all messages are responded to with a ‘Bad record MAC’ alert. This usually is an indication of wrong keys being used. Closer inspection revealed that at this point OpenSSL changes the keys that the client uses to encrypt and MAC messages to the server keys. This means that in both directions the same keys are used from this point.

We observed the following behaviour after the additional ChangeCipherSpec message. First, OpenSSL expects a ClientHello message (instead of a Finished message

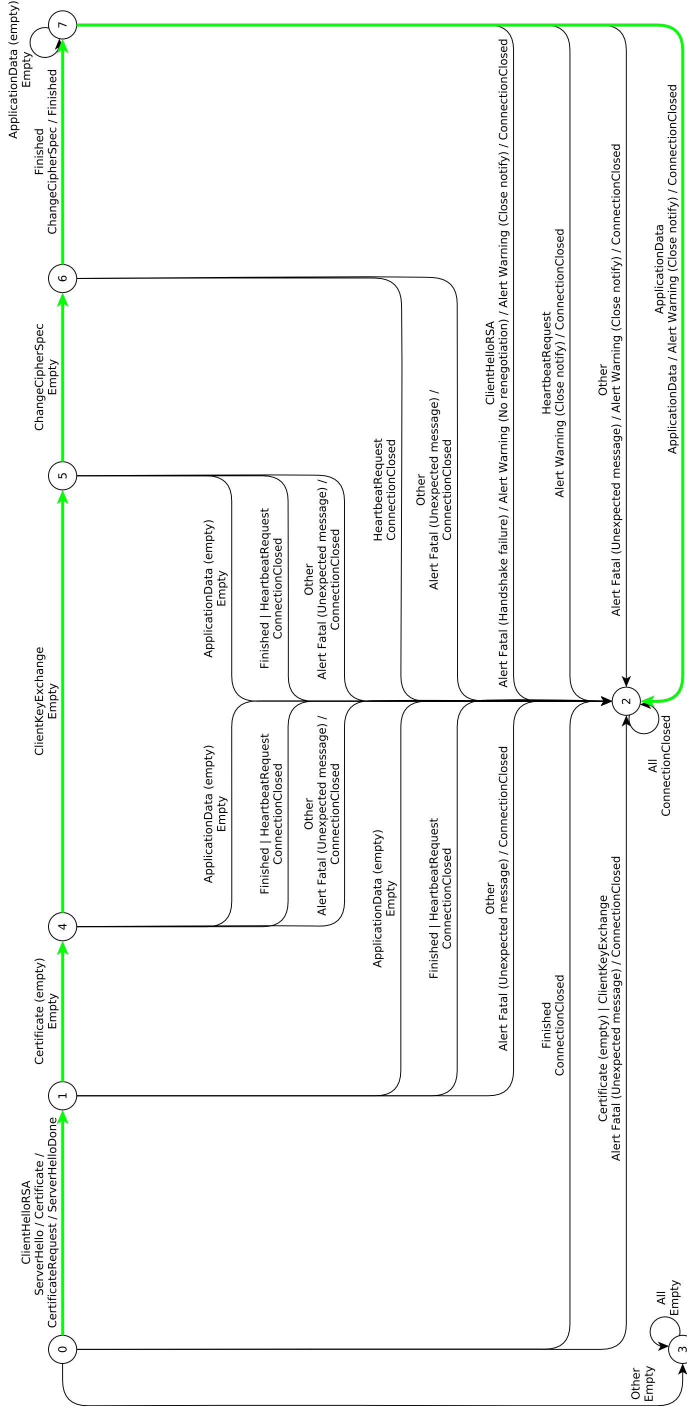


Figure 7.10: Learned state machine model for NSS 3.17.1

Figure 7.11: Learned state machine model for OpenSSL 1.0.1j

as one would expect). This ClientHello is responded to with the ServerHello, ChangeCipherSpec and Finished messages. OpenSSL does change the server keys then, but does not use the new randoms from the ClientHello and ServerHello to compute new keys. Instead the old keys are used and the cipher is thus basically reset (i.e. the original IVs are set and the MAC counter reset to 0). After receiving the ClientHello message, the server does expect the Finished message, which contains the keyed hash over the messages since the second ClientHello and does make use of the new client and server randoms. After this, application data can be send over the connection, where the same keys are used in both directions. The issue was reported to the OpenSSL team and was fixed in version 1.0.1k.

**Early ChangeCipherSpec** The state machine model of the older version OpenSSL 1.0.1g (Figure 7.12) reveals a known vulnerability that was recently discovered [Kik], which makes it possible for an attacker to easily compute the session keys that are used in the versions up to 1.0.0l and 1.0.1g, as described below.

As soon as a ChangeCipherSpec message is received, the keys are computed. However, this also happened when no ClientKeyExchange was sent yet, in which case an empty master secret is used. This results in keys that are computed based on only public data. In version 1.0.1 it is possible to completely hijack a session by sending an early ChangeCipherSpec message to both the server and client, as in this version the empty master secret is also used in the computation of the hash in the Finished message. In the model of OpenSSL version 1.0.1g in Figure 7.12 it is clear that if a ChangeCipherSpec message is received too early, the Finished message is still accepted as a ChangeCipherSpec is returned (see path *0, 1, 6, 9, 12* in the model). This is an indication of the bug and would be reason for closer inspection. The incoming messages after this path cannot be decrypted anymore however, because the corresponding keys are only computed by our test harness as soon as the ChangeCipherSpec message is received, which means that these keys are actually based on the ClientKeyExchange message. A simple modification of the test harness to change the point at which the keys are computed will even provide a successful exploitation of the bug.

An interesting observation regarding the evolution of the OpenSSL code is that for the four different versions that we analysed (1.0.1g, 1.0.1j, 1.0.1l and 1.0.2) the number of states reduces with every version. For version 1.0.2 there is still one state more than required, but this is an error state from which all messages result in a closed connection.

### 7.5.9 nqsb-TLS

A recent TLS implementation, nqsb-TLS, is intended to be both a specification and usable implementation written in OCaml [MMMS15]. For nsqb-TLS we analysed version 0.4.0 (see Figure 7.13). Our analysis revealed a bug in this implementation:

Figure 7.12: Learned state machine model for OpenSSL 1.0.1g, an older version of OpenSSL which had a known security flaw [Kik].

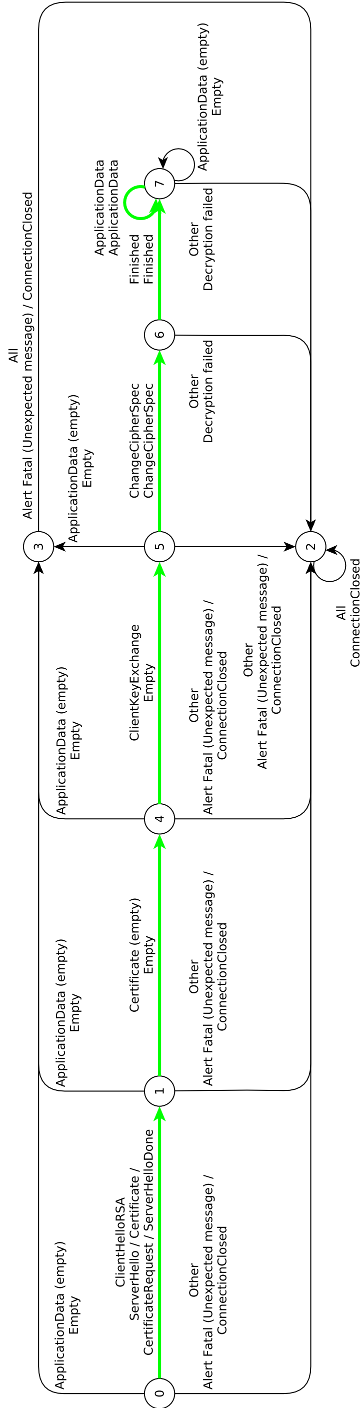


Figure 7.13: Learned state machine model for nqsb-TLS 0.4.0

alert messages are not encrypted even after a ChangeCipherSpec is received. What is more interesting is a design decision with regard to the state machine: after the client sends a ChangeCipherSpec, the server immediately responds with a ChangeCipherSpec. This is different compared to all other implementations, that first wait for the client to also send a Finished message before sending a response. This is a clear example where the TLS specifications are not completely unambiguous and adding a state machine would remove room for interpretation.

## 7.6 Conclusion

We presented a thorough analysis of commonly used TLS implementations using the systematic approach we call protocol state fuzzing: we use state machine learning, which relies only on black box testing, to infer a state machine and then we perform a manual analysis of the state machines obtained. We demonstrated that this is a powerful and fast technique to reveal security flaws: in 3 out of 9 tested implementations we discovered new flaws. We applied the method on both server- and client-side implementations. By using our modified version of the W-method we are able to drastically reduce the number of equivalence queries used, which in turn results in a much lower running time of the analysis.

Our approach is able to find mistakes in the logic in the state machine of implementations. Deliberate backdoors, that are for example triggered by sending a particular message 100 times, would not be detected. Also mistakes in, for example, the parsing of messages or certificates would not be detected.

An overview of different approaches to prevent security bugs and more generally improve the security of software is given in [Whe14] (using the Heartbleed bug as a basis). The method presented in this paper would not have detected the Heartbleed bug, but we believe it makes a useful addition to the approaches discussed in [Whe14]. It is related to some of the approaches listed there; in particular, state machine learning involves a form of negative testing: the tests carried out during the state machine learning include many negative tests, namely those where messages are sent in unexpected orders, which one would expect to result in the closing of the connection (and which probably *should* result in closing of the connection, to be on the safe side). By sending messages in an unexpected order we get a high coverage of the code, which is different from for example full branch code coverage, as we trigger many different paths through the code.

In parallel with our research Beurdouche et al. [BBFK+15] independently performed closely related research. They also analyse protocol state machines of TLS implementations and successfully find numerous security flaws. Both approaches have independently come up with the same fundamental idea, namely that protocol state machines are a great formalism to systematically analyse implementations of security protocols. Both approaches require the construction of a framework to send arbitrary

TLS messages, and both approaches reveal that OpenSSL and JSSE have the most (over)complicated state machines.

The approach of Beurdouche et al. is different though: whereas we infer the state machines from the code without prior knowledge, they start with a manually constructed reference protocol state machine, and subsequently use this as a basis to test TLS implementations. Moreover, the testing they do here is not truly random, as the 'blind' learning by LearnLib is, but uses a set of test traces that is automatically generated using some heuristics.

The difference in the issues identified by Beurdouche et al. and us can partly be explained by the difference in functionality that is supported by the test frameworks used. For example, our framework supports the Heartbeat extension, whereas theirs supports Diffie-Hellman certificates and export cipher suites. Another reason is the fact that our approach has a higher coverage due to its 'blind' nature.

One advantage of our approach is that we don't have to construct a correct reference model by hand beforehand. But in the end, we do have to decide which behaviour is unwanted. Having a visual model helps here, as it is easy to see if there are states or transitions that seem redundant and don't occur in other models. Note that both approaches ultimately rely on a manual analysis to assess the security impact of any protocol behaviour that is deemed to be deviant or superfluous.

When it comes to implementing TLS, the specifications leave the developer quite some freedom as how to implement the protocol, especially in handling errors or exceptions. Indeed, many of the differences between models we infer are variations in error messages. These are not fixed in the specifications and can be freely chosen when implementing the protocol. Though this might be useful for debugging, the different error messages are probably not useful in production (especially since they differ per implementation).

This means that there is not a single 'correct' state machine for the TLS protocol and indeed every implementation we analysed resulted in a different model. However, there are some clearly wrong state machines. One would expect to see a state machine where there is clearly one correct path (or possibly more depending on the configuration) and all other paths going to one error state – preferably all with the same error code. We have seen one model that conforms to this, namely the one for RSA BSAFE for Java, shown in Figure 7.9.

Of course, it would be interesting to apply the same technique we have used on TLS implementations here on implementations of other security protocols. The main effort in protocol state fuzzing is developing a test harness. But as only one test harness is needed to test all implementations for a given protocol, we believe that this is a worthwhile investment. In fact, one can argue that for any security protocol such a test harness should be provided to allow analysis of implementations.

The first manual analysis of the state machines we obtain is fairly straightforward: any superfluous strange behaviour is easy to spot visually. This step could even be automated as well by providing a correct reference state machine. A state machine

that we consider to be correct would be the one that we learned for RSA BSAFE for Java.

Deciding whether any superfluous behaviour is exploitable is the hardest part of the manual analysis, but for security protocols it makes sense to simply require that there should not be any superfluous behaviour whatsoever.

The difference behaviour between the various implementations might be traced back to Postel's Law:

> 'Be conservative in what you send,
> be liberal in what you accept.'

As has been noted many times before, e.g. in [SPB12], this is an unwanted and risky approach in security protocols: if there is any suspicion about inputs they should be discarded, connections should be closed, and no response should be given that could possibly aid an attacker. To quote [Gee10]: 'It's time to deprecate Jon Postel's dictum and to be conservative in what you accept'.

Of course, ideally state machines would be included in the official specifications of protocols to begin with. This would provide a more fundamental solution to remove – or at least reduce – some of the implementation freedom. It would avoid each implementer having to come up with his or her own interpretation of English prose specifications, avoiding not only lots of work, but also the large variety of state machines in implementations that we observed, and the bugs that some of these introduce.

# Chapter 8

# Conclusions

In this thesis we have shown, using the EMV protocol as a case study, that even though real world security protocols can be very complex, it is still possible to analyse them using formal methods. Though using pi-calculus directly was no longer feasible, a functional language such as F# is powerful enough to specify the protocol and existing tools can still be used to perform the analysis. The main advantage of F# is that it provides sequential if-statements and functions. This provides the flexibility required to formalise a protocol like EMV that offers many options. This does raise the question whether these protocols cannot be simpler. Due to backwards compatibility it might be necessary to support old functionality, but with contactless EMV – the new EMV variant for contactless smart cards – for example, support for magnetic stripe is still included which makes it two technologies backwards compatible: magnetic stripe data was included in the original EMV specifications for backwards compatibility, and now for contactless EMV – based on the original contact-based EMV specifications – it is still included and even a special 'mag-stripe mode' is introduced in the specifications.

However, as having only correct specifications is not enough we also looked at actual implementations of security protocols: namely bank cards, a handheld smart card reader for online banking and TLS implementations. State machine learning for security protocols – what we call protocol state fuzzing – is a very useful technique to analyse implementations. As opposed to the formal analysis that can find mistakes in protocol specifications as discussed before, this technique can detect mistakes in actual implementations of the protocol. Previously, state machines have been extracted from implementations of security protocols by hand [PS11], but using protocol state fuzzing this can be automated. A general observation that can be made is that most of the state machines learned from implementations are more complicated than one would expect and is necessary. Especially when implementing security protocols it is crucial to not introduce superfluous behaviour as this might eventually lead to security issues. For TLS it was clear to see that developers might not always have a clear picture of the state machine in mind of the state machine for the protocol they are implementing: in 3 out of 9 implementations we discovered new flaws using this method (GnuTLS, Java Secure Socket Extension and OpenSSL). Developers will always look at the 'correct' path as this is vital for correct functioning of the implementation, but it is just as important to consider what to do if unexpected things happen. One hopes that every developer makes a conscious decision about the state machine when implementing a protocol, though everybody still has to struggle to extract them from the specifications

from scratch.

The success of protocol state fuzzing demonstrated in this thesis highlights the usefulness of state machines as a specification formalism. We discovered a surprising variety of state machines found in smart card implementations of EMV and TLS implementations. This suggests it would be good to have these state machines in the actual specifications. Ideally every protocol specification would come with a state machine. And even when no state machine is provided, at least a test harness can be provided to test an implementation.

Considering our current and previous findings, the question is who takes responsibility of the security of protocols like EMV. Many parties are involved in the EMV ecosystem such as EMVCo, payments systems, banks and manufacturers. These parties seem to place a lot of trust into each other when it comes to security of the protocols and devices that are in use. A clear example of this is the e.dentifier2: how was it possible that the bug in the device was not picked up by any of the parties involved before?

Based on observations we made during the analysis of the e.dentifier2 we proposed a solution for a handheld card reader for online transactions that provides stronger security guarantees than existing devices: the Radboud Reader. The trusted computing base of this device is minimised and thus it is easier to check and less trust has to be placed in the manufacturer of the device.

## 8.1 Future work

A useful next step would be to add support for protocol state fuzzing of other security protocols, e.g. PPTP, IPSec, WEP and WPA. This can be used to analyse existing implementations and aid developers implementing these protocols.

Also it would be good to come up with reference state machines for existing protocols, as these are usually not yet included in protocol specifications. This could provide important guidance for any developer implementing a security protocol.

To make protocol state fuzzing even more powerful, it might prove useful to add additional information to the analysis. This could, for example, be the timing of responses. This information might be used to find timing attacks against implementation, e.g. when the response time is dependent on the number of correct numbers in a PIN code. An orthogonal direction would be to take into account code coverage, branch coverage or actual source code in a grey-box or white-box setting respectively. This would take advantage of additional access developers or users might have to the implementations that are tested. This method could be extended by using the learned model as guidance in traditional fuzzing.

Analysis of the EMV ecosystem could be extended by considering payment terminals and other systems used in the back-end. Analysis of these systems can again be done using formal analysis of the specifications and protocol state fuzzing on ac-

tual implementations. Previous research has shown that these devices are not without flaws [DMA08,Bre14], so there might still be room for improvement for these systems.

# Bibliography

[AB05a] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, January 2005.

[AB05b] Xavier Allamigeon and Bruno Blanchet. Reconstruction of attacks against cryptographic protocols. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 140–154. IEEE, 2005.

[ABP+13] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 305–320. USENIX Association, 2013.

[AJU10] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In Alexandre Petrenko, Adenilso Simão, and JoséCarlos Maldonado, editors, *Testing Software and Systems – 22nd IFIP WG 6.1 International Conference, ICTSS 2010*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer-Verlag, 2010.

[AMR+12] Myrto Arapinis, Loretta Mancini, Eike Ritter, Mark Ryan, Nico Golde, Kevin Redon, and Ravishankar Borgaonkar. New privacy issues in mobile telephony: Fix and verification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 205–216. ACM, 2012.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540. IEEE, 2013.

[ARP13] Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal models of bank cards for free. In *4th International Workshop on Security Testing (SECTEST 2013), in association with the 6th International Conference on Software Testing, Verification and Validation (ICST)*, pages 461–468. IEEE, 2013.

[ASV10] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation – 4th International Symposium on Leveraging Applications, ISoLA 2010*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer-Verlag, 2010.

[BBFK+15] Antoine Delignat-Lavaud Benjamin Beurdouche, Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, , and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 535–552. IEEE, 2015.

[BBLF11] Andrea Barisani, Daniele Bianco, Adam Laurie, and Zac Franken. Chip & PIN is definitely broken. Presentation at CanSecWest Applied Security Conference, Vancouver. Slides available at http://dev.inversepath.com/download/emv/emv_2011.pdf, 2011.

[BCM+14] Mike Bond, Omar Choudary, Steven J. Murdoch, Sergei Skorobogatov, and Ross Anderson. Chip and skim: cloning EMV cards with the preplay attack. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 49–64. IEEE, 2014.

[BECN+04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. Cryptology ePrint Archive, Report 2004/100, 2004. http://eprint.iacr.org/.

[BFCZ08] 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*, CCS '08, pages 459–468. ACM, 2008.

[BFGT08] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1):5:1–5:61, December 2008.

[BFK+13] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459, 2013.

[BJR+14] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 114–129, 2014.

[BKGP+12] Arjan Blom, Gerhard de Koning Gans, Erik Poll, Joeri de Ruiter, and Roel Verdult. Designed to fail: A USB-connected reader for online banking. In Audun Jøsang and Bengt Carlsson, editors, *Secure IT Systems – 17th Nordic Conference, NordSec 2012*, volume 7617 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2012.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*, pages 82–96. IEEE, 2001.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1998.

[Blo11] Arjan Blom. ABN-AMRO e.dentifier2 reverse engineering, 2011. MSc. thesis, Radboud University Nijmegen.

[Bre14] Jordi van den Breekel. A security evaluation and proof-of-concept relay attack on Dutch EMV contactless transactions, 2014. MSc. thesis, Eindhoven University of Technology.

[CGR+15] Tom Chothia, Flavio Garcia, Joeri de Ruiter, Jordi van den Breekel, and Matthew Thompson. A lightweight distance-bounding contactless EMV payment protocol. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security – 19th International Conference, FC 2015*, volume 8975 of *Lecture Notes in Computer Science*. Springer-Verlag, 2015.

[Che] Check-In-Phone – Technology and Security. Available from http://upload.rb.ru/upload/users/files/3374/check-in-phone-technologie_security-english-_2010-08-12_20.05.11.pdf.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*, 4(3):178–187, 1978.

[Cod] Codenomicon. Heartbleed bug. http://heartbleed.com/. Accessed on August 26th 2014.

[Cou09] European Payments Council. SEPA Cards Framework, version 2.1, 2009.

[CPPR14] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri de Ruiter. Automated reverse engineering using Lego®. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*. USENIX Association, August 2014.

[CRP14] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on Android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 11–20. ACM, 2014.

[DA99] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, 1999.

[DCVP04] Gregorio Díaz, Fernando Cuartero, Valentín Valero, and Fernando Pelayo. Automatic verification of the TLS handshake protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 789–794. ACM, 2004.

[DM07] Saar Drimer and Steven J. Murdoch. Keep your enemies close: distance bounding against smartcard relay attacks. In *16th USENIX Security Symposium*, pages 87–102. USENIX Association, 2007.

[DMA08] Saar Drimer, Steven J. Murdoch, and Ross Anderson. Thinking inside the box: System-level failures of tamper proofing. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 281–295. IEEE, 2008.

[DMA09] Saar Drimer, Steven J. Murdoch, and Ross Anderson. Optimised to fail: Card readers for online banking. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security – 13th International Conference, FC 2009*, volume 5628 of *Lecture Notes in Computer Science*, pages 184–200. Springer-Verlag, 2009.

[DR06] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.1. RFC 4346, Internet Engineering Task Force, 2006.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246, Internet Engineering Task Force, 2008.

[DY83] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[EMV05] EMVCo. EMV – Integrated Circuit Card Specifications for Payment Systems – Common Payment Application Specification, version 1.0, 2005.

[EMV08] EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 1-4, 2008. Available at emvco.com.

[EMV11a] EMVCo. EMV – Integrated Circuit Card Specifications for Payment Systems, Book 1: Application Independent ICC to Terminal Interface Requirements, version 4.3, 2011.

[EMV11b] EMVCo. EMV – Integrated Circuit Card Specifications for Payment Systems, Book 2: Security and Key Management, version 4.3, 2011.

[EMV11c] EMVCo. EMV – Integrated Circuit Card Specifications for Payment Systems, Book 3: Application Specification, version 4.3, 2011.

[EMV11d] EMVCo. EMV – Integrated Circuit Card Specifications for Payment Systems, Book 4: Cardholder, Attendant, and Acquirer Interface Requirements, version 4.3, 2011.

[EMV12] EMVCo. Specification bulletin no. 103: Unpredictable Number generation, April 2012.

[FAZB11] Michael Felderer, Berthold Agreiter, Philipp Zech, and Ruth Breu. A classification for model-based security testing. In *VALID 2011, The Third International Conference on Advances in System Testing and Validation Lifecycle*, pages 109–114. IARIA, 2011.

[FIN04] CEN Workshop Agreement (CWA) 14174: Financial transactional IC card reader (FINREAD), 2004.

[Gee10] Dan Geer. Vulnerable compliance. *;login: The USENIX Magazine*, 35(6):10–12, 2010.

[GMP+08] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. Universally composable security analysis of TLS. In Joonsang Baek, Feng Bao, Kefei Chen, and Xuejia Lai, editors, *Provable Security – Second International Conference, ProvSec 2008*, volume 5324 of *Lecture Notes in Computer Science*, pages 313–327. Springer-Verlag, 2008.

[GP06] Global Platform Organization. *Card Specification, Version 2.2*, March 2006. http://www.globalplaform.org.

[Gul10] Peter Gullberg. Method and device for creating a digital signature, 2010. European Patent Application EP 2 166 483 A1, filed September 17, 2008, published March 24, 2010.

[HSD+05] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 2–15. ACM, 2005.

[ICA08] Machine readable travel documents – part 3-2, third edition, 2008.

[ISOa] ISO/IEC. ISO/IEC 14443: Identification cards – contactless integrated circuit(s) cards – proximity cards.

[ISOb] ISO/IEC. ISO/IEC 7816: Identification cards – integrated circuit cards.

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 273–293. Springer-Verlag, 2012.

[KGR12] Gerhard de Koning Gans and Joeri de Ruiter. The SmartLogic tool: Analysing and testing smart card protocols. In *3rd International Workshop on Security Testing (SECTEST 2012), in association with the 5th International Conference on Software Testing, Verification and Validation (ICST)*, pages 864–871. IEEE, 2012.

[Kik] Masashi Kikuchi. OpenSSL #ccsinjection vulnerability. http://ccsinjection.lepidum.co.jp/. Access on August 26th 2014.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology – CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[KL11] Allaa Kamil and Gavin Lowe. Analysing TLS in the strand spaces model. *Journal of Computer Security*, 19(5):975–1025, 2011.

[KPW13] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer-Verlag, 2013.

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems – Second International Workshop, TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[Mas05] MasterCard International Inc. Paypass - M/Chip technical specifications, 2005. Version 1.3.

[Mas11] MasterCard. PayPass – M/Chip application note #19, 2011.

[MDAB10] Steven J. Murdoch, Saar Drimer, Ross Anderson, and Mike Bond. Chip and PIN is broken. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 433–446. IEEE, 2010.

[MMMS15] David Kaloper Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, August 2015.

[MS14] Christopher Meyer and Jörg Schwenk. SoK: Lessons learned from SSL/TLS attacks. In Yongdae Kim, Heejo Lee, and Adrian Perrig, editors, *Information Security Applications – 14th International Workshop, WISA 2013*, Lecture Notes in Computer Science, pages 189–209. Springer-Verlag, 2014.

[MSW10] P. Morrissey, N.P. Smart, and B. Warinschi. The tls handshake protocol: A modular analysis. *Journal of Cryptology*, 23(2):187–223, 2010.

[MSW+14] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 733–748. USENIX Association, 2014.

[Nie03] Oliver Niese. *An integrated approach to testing complex systems.* PhD thesis, Dortmund University, 2003.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[OF05] Kazuhiro Ogata and Kokichi Futatsugi. Equational approach to formal analysis of TLS. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 795–804. IEEE, 2005.

[Pau99] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):332–351, 1999.

[PR13] Erik Poll and Joeri de Ruiter. The Radboud Reader: A minimal trusted smartcard reader for securing online transactions. In Simone Fischer-Hübner, Elisabeth Leeuw, and Chris Mitchell, editors, *Policies and Research in Identity Management – Third IFIP WG 11.6 Working Conference, IDMAN 2013*, volume 396 of *IFIP Advances in Information and Communication Technology*, pages 107–120. Springer-Verlag, 2013.

[PS11] Erik Poll and Aleksy Schubert. Rigorous specifications of the SSH Transport Layer. Technical Report ICIS-R11004, Radboud University Nijmegen, 2011.

[RMP08] Henning Richter, Wojciech Mostowski, and Erik Poll. Fingerprinting passports. In *NLUUG Spring Conference on Security*, pages 21–30, 2008.

[RP12] Joeri de Ruiter and Erik Poll. Formal analysis of the EMV protocol suite. In S. Mödersheim and C. Palamidessi, editors, *Theory of Security and Applications – Joint Workshop, TOSCA 2011*, volume 6993 of *Lecture Notes in Computer Science*, pages 113–129. Springer-Verlag, 2012.

[RP15] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, August 2015.

[RSBM09] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer*, 11:393–407, 2009.

[Rui14] Joeri de Ruiter. Automated algebraic analysis of structure-preserving signature schemes. Cryptology ePrint Archive, Report 2014/590, 2014. http://eprint.iacr.org/.

[SPB12] Len Sassaman, Meredith L. Patterson, and Sergey Bratus. A patch for postel's robustness principle. *IEEE Security & Privacy*, 10(2):87–91, 2012.

[ST11] Jean-Pierre Szikora and Philippe Teuwen. Banques en ligne: à la découverte d'EMV-CAP. *MISC (Multi-System & Internet Security Cookbook)*, 56:50–62, 2011.

[STW12] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, Internet Engineering Task Force, 2012.

[TP11] S. Turner and T. Polk. Prohibiting Secure Sockets Layer (SSL) version 2.0. RFC 6176, Internet Engineering Task Force, 2011.

[Vis11] Visa. Key mandates summary, 2011.

[Whe14] David A. Wheeler. Preventing Heartbleed. *Computer*, 47(8):80–83, 2014.

[WKH+08] Thomas Weigold, Thorsten Kramp, Reto Hermann, Frank Höring, Peter Buhler, and Michael Baentsch. The Zurich Trusted Information Channel – an efficient defence against man-in-the-middle and malicious software attacks. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch,

editors, *Trusted Computing - Challenges and Applications – First International Conference on Trusted Computing and Trust in Information Technologies, TRUST 2008*, volume 4968 of *Lecture Notes in Computer Science*, pages 75–91. Springer-Verlag, 2008.

[WRF12] David Williams, Joeri de Ruiter, and Wan Fokkink. Model checking under fairness in ProB and its application to fair exchange protocols. In Abhik Roychoudhury and Meenakshi D'Souza, editors, *Theoretical Aspects of Computing – ICTAC 2012*, volume 7521 of *Lecture Notes in Computer Science*, pages 168–182. Springer-Verlag, 2012.

# Index


Active attack, 6, 7
AID, 5, 73
APDU, 6, 23, 74
  Command APDU, 6
  Response APDU, 6
Application Identifier, *see* AID
Application Protocol Data Unit, *see* APDU
Attack trace, 8
Authentication, 9, 32–34, 68
Authorisation Request Cryptogram, *see* ARQC
Automated learning, 10–12, 29, 35–47, 59–66, 81–107

Chip Authentication Program, *see* EMV-CAP
Cloning cards, 25
Command APDU, 37, 75
Common Core Definition, 13
Common Payment Application, 13
Confidentiality, *see* Secrecy

Dolev-Yao attacker, 8, 69

e.dentifier2, 2–4, 49, 52–66, 78, 79
EMV, 1–3, 13–47, 49, 51, 60, 70, 78
  AAC, 17, 21–23, 39, 43, 44, 49, 56
  Action Codes, 22
  AIP, 18, 30, 32
  Application Authentication Cryptogram, *see* AAC
  Application Interchange Profile, *see* AIP
  Application Transaction Counter, *see* ATC
  ARQC, 17, 21–23, 25, 26, 38, 39, 43, 44, 49, 50, 53, 56
  ATC, 22, 27, 30, 38, 45, 50, 60
  CAM, 13, 15–19
  Card Authentication Method, *see* CAM
  Cardholder Verification Method, *see* CVM
  CDA, 13, 15, 16, 18–19, 22, 27, 32, 34, 35
  CDOL, 50
  CDOL1, 18, 19, 22, 30, 31
  CDOL2, 18, 19, 22, 30, 31
  Combined Data Authentication, *see* CDA, *see* CDA
  CVM, 13, 16, 17, 20–21, 25–26
  Data Object List, *see* DOL
  DDA, 13, 15, 16, 18–19, 25, 32, 33, 35, 44, 45
  DDOL, 19, 31
  DOL, 16, 30–31, 35
  Dynamic Data Authentication, *see* DDA, *see* DDA
  ICC Dynamic Number, 19, 21
  Issuer Action Codes, 22, 26
  PDOL, 17–19, 31
  Script processing, 16, 17, 23–24
  SDA, 13, 15, 16, 18–19, 25, 32, 33, 35
  Static Data Authentication, *see* SDA, *see* SDA
  TC, 17, 21–23, 26, 34, 38, 39, 43, 46

Terminal Action Codes, 22, 26
Transaction Certificate, *see* TC
UN, 19, 26, 27, 31, 50, 55, 56
Unpredictable Number, *see* UN
EMV-CAP, 2, 3, 36, 43, 49–51, 53, 56, 67, 78, 79
IPB, 49
Issuer Proprietary Bitmap, *see* IPB
Mode 1, 50, 51, 53, 56
Mode 2, 50, 56
Mode 2 + TDS, 50, 51
Mode 3, 50, 53
EMVCo, 13

F#, 9, 29, 34, 35
FS2PV, 3, 9, 29, 35

HTTPS, 1, 2, 81

ISO/IEC 14443, 5
ISO/IEC 7816, 5–6, 69, 70, 73, 74

L*, 11, 12
equivalence query, 11
Learner, 11
output query, 11
reset query, 11
Teacher, 11, 36
LearnLib, 12, 36, 39, 43, 46, 59, 60, 65, 81–85

Man-in-the-Browser, 51, 52, 67, 80
Man-in-the-Middle, 7, 25, 29, 33, 35, 58
Mapper, 38, 45
Mealy machine, 10–11
Model checker, 8

Needham-Schroeder protocol, 8
Non-repudiation, 68

Passive attack, 6
Pi-calculus, 8, 9, 35
ProVerif, 1, 3, 8–9, 29–35

Random walk method, 12, 36, 43, 60, 62
Relay attack, 24
Response APDU, 37, 76

Secrecy, 9, 32, 68
Security guarantee, 68
Security property, 9
Security requirements, 31–35
Side-channel attack, 6, 7
SmartLogic, 7, 26
State machine, 10, 35, 65
Status Word, 6
SUT, 83–85
SWYS, 52, 54–59
System Under Test, 10, 36, 39

TLS, 2, 4, 54, 70, 77, 81–107

W-method, 12, 36, 43, 60, 62, 82–83, 85
Wedge attack, 25
What-You-See-Is-What-You-Sign, *see* WYSIWYS
WYSIWYS, 49, 51–52, 56, 58, 67, 69, 77

# Acronyms

**AAC** Application Authentication Cryptogram . . . 22
**AC** Application Cryptogram . . . 23
**AFL** Application File Locator . . . 18
**AID** Application Identifier . . . 5
**AIP** Application Interchange Profile . . . 18
**APDU** Application Protocol Data Unit . . . 6
**ARQC** Authorisation Request Cryptogram . . . 22
**ATC** Application Transaction Counter . . . 14
**ATM** Automatic Teller Machine . . . 27
**CA** Certificate Authority . . . 14
**CAM** Card Authentication Method . . . 13
**CCD** Common Core Definition . . . 13
**CDA** Combined Data Authentication . . . 13
**CDOL** Card Risk Management Data Object List . . . 23
**CDOL1** Card Risk Management Data Object List 1 . . . 18
**CDOL2** Card Risk Management Data Object List 2 . . . 18
**CID** Cryptogram Information Data . . . 20
**CPA** Common Payment Application . . . 13
**CVM** Cardholder Verification Method . . . 13
**DDA** Dynamic Data Authentication . . . 13
**DDOL** Dynamic Data Authentication Data Object List . . . 19
**DES** Data Encryption Standard . . . 56

**DOL** Data Object List ........ 16
**DPA** Differential Power Analysis ........ 7
**GPIO** General-purpose Input/Output ........ 60
**IAC** Issuer Action Code ........ 22
**IAD** Issuer Application Data ........ 23
**ICC** Integrated Circuit Card ........ 5
**IPB** Issuer Proprietary Bitmap ........ 49
**MAC** Message Authentication Code ........ 15
**MitB** Man-in-the-Browser ........ 52
**MitM** Man-in-the-Middle ........ 7
**PAN** Primay Account Number ........ 18
**PDOL** Processing Options Data Object List ........ 18
**PIN** Personal Identification Number ........ 7
**PKI** Public Key Infrastructure ........ 14
**POS** Point of Sale ........ 5
**PSN** PAN Sequence Number ........ 49
**SDA** Static Data Authentication ........ 13
**SDAD** Signed Dynamic Application Data ........ 19
**SEPA** Single Euro Payment Area ........ 13
**SPA** Simple Power Analysis ........ 7
**SSAD** Signed Static Application Data ........ 19
**SSC** Session Sequence Counter ........ 77
**SSH** Secure Shell ........ 77
**SSL** Secure Socket Layer ........ 82
**SUT** System Under Test ........ 10
**SWYS** Sign-What-You-See ........ 52
**TAC** Terminal Action Code ........ 22
**TC** Transaction Certificate ........ 22
**TDHC** Transaction Data Hash Code ........ 20
**TDOL** Transaction Certificate Data Object List ........ 23
**TDS** Transaction Data Signing ........ 50
**TLS** Transport Layer Security ........ 4
**TLV** Tag-Length-Value ........ 35
**TVR** Terminal Verification Results ........ 22

**UN** Unpredictable Number . . . . . . . . 19

**WYSIWYS** What-You-See-Is-What-You-Sign . . . . . . . . 49

# Curriculum vitae

## Joeri Evert Jan de Ruiter

1986:
Born in Sittard, the Netherlands

2004 - 2007:
Bachelor of Science (Cum Laude)
Computer Science and Engineering
Eindhoven University of Technology

2007 - 2010:
Master of Science (Cum Laude)
Computer Science and Engineering, Information Security Technology track
Eindhoven University of Technology, Kerckhoffs Institute

2010 - 2014:
PhD
Digital Security
Radboud University Nijmegen

## Titles in the IPA Dissertation Series since 2009

**M.H.G. Verhoef**. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

**M. de Mol**. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

**M. Lormans**. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

**M.P.W.J. van Osch**. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

**H. Sozer**. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

**M.J. van Weerdenburg**. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

**H.H. Hansen**. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

**A. Mesbah**. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

**A.L. Rodriguez Yakushev**. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

**K.R. Olmos Joffré**. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

**J.A.G.M. van den Berg**. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

**M.G. Khatib**. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

**S.G.M. Cornelissen**. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

**D. Bolzoni**. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

**H.L. Jonker**. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

**M.R. Czenko**. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

**T. Chen**. *Clocks, Dice and Processes.* Faculty of Sciences, Division

of Mathematics and Computer Science, VUA. 2009-17

**C. Kaliszyk**. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

**R.S.S. O'Connor**. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

**B. Ploeger**. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

**T. Han**. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

**R. Li**. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

**J.H.P. Kwisthout**. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

**T.K. Cocx**. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

**A.I. Baars**. *Embedded Compilers.* Faculty of Science, UU. 2009-25

**M.A.C. Dekker**. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

**J.F.J. Laros**. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

**C.J. Boogerd**. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

**M.R. Neuhäußer**. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

**J. Endrullis**. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

**T. Staijen**. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

**Y. Wang**. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

**J.K. Berendsen**. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

**A. Nugroho**. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

**A. Silva**. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

**J.S. de Bruin**. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

**D. Costa**. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

**M.M. Jaghoori**. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

**R. Bakhshi**. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

**B.J. Arnoldus**. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

**E. Zambon**. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

**L. Astefanoaei**. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

**J. Proença**. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

**A. Moralı**. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

**M. van der Bijl**. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

**C. Krause**. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

**M.E. Andrés**. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

**M. Atif**. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

**P.J.A. van Tilburg**. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

**Z. Protic**. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

**S. Georgievska**. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

**S. Malakuti**. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

**M. Raffelsieper**. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

**C.P. Tsirogiannis**. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

**Y.-J. Moon**. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

**R. Middelkoop**. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

**M.F. van Amstel**. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

**A.N. Tamalet**. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

**H.J.S. Basten**. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

**M. Izadi**. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

**L.C.L. Kats**. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

**S. Kemper**. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

**J. Wang**. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

**A. Khosravi**. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

**A. Middelkoop**. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

**Z. Hemel**. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

**T. Dimkov**. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

**S. Sedghi**. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

**F. Heidarian Dehkordi**. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

**K. Verbeek**. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

**D.E. Nadales Agut**. *A Compositional Interchange Format for Hybrid Systems:*

*Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

**H. Rahmani**. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

**S.D. Vermolen**. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

**L.J.P. Engelen**. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

**F.P.M. Stappers**. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

**W. Heijstek**. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

**C. Kop**. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

**A. Osaiweran**. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

**W. Kuijper**. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

**H. Beohar**. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

**G. Igna**. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

**E. Zambon**. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

**B. Lijnse**. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

**G.T. de Koning Gans**. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

**M.S. Greiler**. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

**L.E. Mamane**. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

**M.M.H.P. van den Heuvel**. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

**J. Businge**. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

**S. van der Burg**. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

**J.J.A. Keiren**. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

**D.H.P. Gerrits**. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

**M. Timmer**. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

**M.J.M. Roeloffzen**. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

**L. Lensink**. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

**C. Tankink**. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

**C. de Gouw**. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

**J. van den Bos**. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

**D. Hadziosmanovic**. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

**A.J.P. Jeckmans**. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

**C.-P. Bezemer**. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

**T.M. Ngo**. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

**A.W. Laarman**. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

**J. Winter**. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

**W. Meulemans**. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

**A.F.E. Belinfante**. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

**A.P. van der Meer**. *Domain Specific Languages and their Type Systems.* Faculty of Mathematics and Computer Science, TU/e. 2014-10

**B.N. Vasilescu**. *Social Aspects of Collaboration in Online Software Communities.* Faculty of Mathematics and Computer Science, TU/e. 2014-11

**F.D. Aarts**. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2014-12

**N. Noroozi**. *Improving Input-Output Conformance Testing Theories.* Faculty of Mathematics and Computer Science, TU/e. 2014-13

**M. Helvensteijn**. *Abstract Delta Modeling: Software Product Lines and Beyond.* Faculty of Mathematics and Natural Sciences, UL. 2014-14

**P. Vullers**. *Efficient Implementations of Attribute-based Credentials on Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2014-15

**F.W. Takes**. *Algorithms for Analyzing and Mining Real-World Graphs.* Faculty of Mathematics and Natural Sciences, UL. 2014-16

**M.P. Schraagen**. *Aspects of Record Linkage.* Faculty of Mathematics and Natural Sciences, UL. 2014-17

**G. Alpár**. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World.* Faculty of Science, Mathematics and Computer Science, RU. 2015-01

**A.J. van der Ploeg**. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

**R.J.M. Theunissen**. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

**T.V. Bui**. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

**A. Guzzi**. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

**T. Espinha**. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

**S. Dietzel**. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

**E. Costante**. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

**S. Cranen**. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

**R. Verdult**. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

**J.E.J. de Ruiter**. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11