Formal Analysis of the EMV Protocol Suite

Joeri de Ruiter and Erik Poll

Digital Security Group
Institute for Computing and Information Science (ICIS)
Radboud University Nijmegen

Abstract. This paper presents a formal model of the EMV (Europay-MasterCard-Visa) protocol suite in F# and its analysis using the protocol verification tool ProVerif [5] in combination with FS2PV [4].
The formalisation covers all the major options of the EMV protocol suite, including all card authentication mechanisms and both on- and offline transactions. Some configuration parameters have to be fixed to allow any security analysis; here we follow the configuration of Dutch EMV banking cards, but the model could easily be adapted to other configurations.
As far as we know this is the first comprehensive formal description of EMV. The convenience and expressivity of F# proved to be a crucial advantage to make the formalisation of something as complex as EMV feasible. Even though the EMV specs amount to over 700 pages, our formal model is only 370 lines of code.
Formal analysis of our model with ProVerif is still possible, though this requires some care. Our formal analysis does not reveal any new weaknesses of the EMV protocol suite, but it does reveal all the known weaknesses, as a formal analysis of course should.

1 Introduction

EMV is the world's leading standard for payments with Integrated Circuit Cards (ICCs), i.e. bank or credit cards that incorporate a smartcard chip. The initiative for EMV was taken by Europay, MasterCard and Visa in the 1990s. Currently the EMV standard is maintained by EMVCo, a company jointly owned by MasterCard, Visa, American Express, and JCB. According to EMVCo, the number of EMV cards in use reached 1 billion in 2010. In Europe, the UK has already completed migration to EMV, and the rest of Western Europe is following suit as part of the Single Euro Payment Area (SEPA).

The EMV specifications[1] consist of four books [8–11], amounting to over 700 pages. These books do not define a single protocol, but a highly configurable tool-kit for payment protocols. For instance, it allows a choice between:

- three *Card Authentication Methods*: SDA, DDA, and CDA, discussed in more detail in Section 2;

[1] Publicly available from `http://www.emvco.com`

– five *Cardholder Verification Methods*: none, signature, online PIN, offline unencrypted PIN, and offline encrypted PIN;
– on- and off-line transactions.

Moreover, many of these options are again parameterised, as explained in Section 2, possibly using proprietary data and formats. All these options and parameterisations make the EMV standard very difficult to comprehend, and a formal model to analyse the security implications all the more valuable.

Some weaknesses of certain EMV configurations are widely known and apparently accepted – at least by some issuers. For example, SDA cards can be cloned, and these clones can be used for offline transactions. The DDA mechanism that prevents such cloning authenticates the card but not the subsequent transaction, which led to a further improvement in the form of the CDA mechanism. More recently, a weakness in some configurations of EMV was discovered whereby a payment terminal can be fooled into thinking a payment was confirmed with PIN, when in reality it is not [12].

The rest of this paper is organised as follows. The EMV standard is explained in sections 2 and 3: Section 2 explains the fundamental concepts and terminology used in the standard, and then Section 3 describes the EMV protocols. Section 4 describes our formal F# model of EMV. Finally Section 5 discusses the security requirements for this formal model which have been verified using ProVerif.

2 EMV Fundamentals

This section explains the basic building blocks of EMV and some central concepts and terminology, and some of our notation.

In the EMV protocol several entities can be identified. First we have the customer, who is in possession of his EMV-compliant banking card, issued by his bank. The customer can use the card to perform a transaction at a merchant. The merchant has a terminal that can communicate with the customer's card. The terminal can also communicate over a secure channel with the bank, who has to authorise and perform the actual transaction. The terminal is 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 Infrastructure The essence of the key set-up in EMV is as follows:

– Every card has a unique symmetric key MK_{AC} that it shares with the issuer, derived from the issuer's master key MK_I. Using this key a session key SK_{AC} can be computed, based on the transaction counter.
– The issuer has a public-private key pair (P_I, S_I), and the terminal knows this public key P_I.
– Cards that support asymmetric crypto also have a public-private key pair (P_{IC}, S_{IC}).

This key setup is the basis of trust between the different parties.

This set-up provides cards with 2 mechanisms to prove authenticity of data:

1. All EMV cards can put MACs on messages, using the shared symmetric key with the issuing bank. The issuer can check these MACs, to verify authenticity of the messages, but the terminal cannot.
2. Cards that support asymmetric crypto can also digitally sign data to prove their authenticity to the terminal, as well as to the issuer.

We write H for hashing, $\mathsf{encrypt}$ for encrypting, and sign for signing.

Card Authentication Mechanisms (CAMs) The EMV standard defines three card authentication mechanisms: SDA, DDA, and CDA.

- In *SDA (Static Data Authentication)* the card provides some digitally signed data (including e.g. the card number and expiry date) to the terminal to authenticate itself. This allows the terminal to check the authenticity of this data, since it knows the issuer's public key, but does not rule out cloning.
- *DDA (Dynamic Data Authentication)* cards can do asymmetric crypto[2] and have a public/private key pair. DDA then involves a challenge-response mechanism, where the card proves its authenticity by signing a challenge chosen by the terminal using a private asymmetric key. Unlike SDA, this does rule out cloning.
  After the authentication of the card, DDA does not tie the subsequent transaction to that card. In other words, the terminal cannot verify that the transaction was actually carried out by the card.
- *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 the transaction.

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 the data is included in the certificate containing the public key of the card. In this sense DDA and CDA subsume SDA.

DOLs An important concept in the EMV specification is that of *Data Object List*, or *DOL*. A DOL specifies a list of data elements, and the format of such a list. Example data elements are the card's *Application Transaction Counter (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 some protocol step (and then also the format that this data has to be in) or which data the card will provide as (signed) output in some protocol step. This is explained in more detail in Section 3.

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

[2] This makes SDA cards cheaper than DDA cards, which seems to be SDA's raison d'être.

data gets signed or MACed, and hence no security analysis is possible without making some assumptions on the DOLs. Our model makes some assumptions on DOLs based on what we observed in Dutch bank and credit cards.

3 An EMV Protocol Session

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

1. *initialisation*: selection of the application on the smartcard and reading of some data;
2. (optionally) *data authentication*, by means of SDA, DDA, or CDA;
3. (optionally) *cardholder verification*, by means of PIN or signature;
4. the actual *transaction*.

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.

3.1 Initialisation

In the first phase of the 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 before providing this information, as specified in the first response.

T → C: `SELECT_APPLICATION`
C → T: [PDOL]
T → C: `GET_PROCESSING_OPTIONS` [(data specified by the PDOL)]
C → T: (AIP, AFL)
Repeat for all records in the AFL:
T → C: `READ_RECORD` (i)
C → T: (Contents of record i)

The protocol starts by selecting the payment application. In response to the selection, 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/Issuer authentication) and whether terminal risk management should be performed.

The AFL is a list identifying the files to be used in the transaction. For each file it is indicated whether it is included in the offline data authentication; if so, the contents of this file is authenticated as part of the card authentication later.

The following files are mandatory:

– *Application Expiry Date*,
– *Application Primary 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 at this stage. The process of card authentication, discussed below, will authenticate some of the data provided by the card.

3.2 Card Authentication

As already mentioned, there are three data authentication methods. The AIP tells the terminal which methods the cards supports, and the terminal should then choose the 'highest' method that both the card and itself support.

SDA On cards that support SDA, the *Signed Static Application Data (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 specified in a record. For SDA no additional communication is needed, since the data needed to verify the SSAD was already retrieved in the initialisation phase.

DDA For DDA some additional communication is needed. DDA consists of two steps.

First, the certificate containing the card's public key is checked. This certificate also contains the hash of the concatenation of the files indicated by the AFL, so this authenticates the same static data that is authenticated with SDA.

Second, a challenge-response protocol is performed. For the challenge an `INTERNAL_AUTHENTICATE` message is sent to the card. The argument of this message 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 *Signed Dynamic Application Data (SDAD)* is the signed *ICC Dynamic Data* and hash of the ICC Dynamic Data and data specified by the DDOL. The ICC Dynamic Data contains at least a time-variant parameter, e.g. a nonce or a transaction counter. On the bank and credit cards we studied, the DDOL only specified a terminal-generated nonce, and the ICC Dynamic Data only consisted of a card-generated nonce.

T → C: `INTERNAL_AUTHENTICATE`(data specified by DDOL)
C → T: $\mathsf{sign}_{S_{IC}}$(ICC Dynamic Data,
H(ICC Dynamic Data, data specified by DDOL))

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

With CDA, the Signed Dynamic Application Data (SDAD) is a signature on a card-generated nonce, the CID, the cryptogram, the *Transaction Data Hash Code (TDHC)* and a terminal-generated nonce (specified by the CDOL1 and CDOL2):

$$\text{SDAD} = \mathsf{sign}_{S_{IC}}(\ \text{nonce}_{IC},\ \text{CID},\ \text{AC},\ \text{TDHC},\ \mathsf{H}(\text{nonce}_{IC},\ \text{CID},\ \text{AC},\ \text{TDHC},\ \text{nonce}_{Terminal})).$$

Here the TDHC is a hash of the elements specified by the PDOL, the elements specified by the CDOL1, the elements specified by the CDOL2 (in case of the second `GENERATE_AC` command) and the elements returned by the card in the response.

3.3 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 *Cardholder Verification Method (CVM)* is used – if any – is quite involved. The card provides the terminal with its list of CVM Rules, that specify under which conditions which CVMs are acceptable, in order of decreasing preference. The terminal then chooses the CVM to be used. In all Dutch banking 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 chip checks the PIN, and the PIN is transmitted to the chip in the clear;
- offline encrypted PIN, in which case the chip 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, which in its turn will return an unauthenticated response indicating whether the PIN was correct or how many failed PIN attempts there are left before the card blocks.

T $\rightarrow$ C: `VERIFY`(pin)
C $\rightarrow$ T: `Success`/ (`PIN_failed`, tries_left) / `Failed`

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.

T → C: `GET_CHALLENGE`
C → T: nonce_{IC}
T → C: `VERIFY`(`encrypt`$_{P_{IC}}$(pin, nonce_{IC}, random_padding))
C → T: `Success`/ (`PIN_failed`, tries_left) / `Failed`

3.4 The transaction

In the final step of an EMV session, after the optional card authentication and card holder verification, the actual transaction is performed. Transactions can be 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 Transaction Certificate (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 TC or ARQC.

Below we first discuss the different types of cryptograms, before we describe the protocol steps for off- and online transactions.

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),
- 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 message 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. Since we do not know how the cryptogram is computed on the Dutch banking cards, we follow the recommendations from the EMV specification. Here a MAC is computed using the symmetric key SK_{AC}, 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 Book 2, 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.

Additionally, if CDA is used, the card also returns the SDAD over the response using its private key P_{IC} so that the terminal can check the authenticity of the complete message.

Both CDOL1 and CDOL2 always include a terminal-generated nonce. A 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 by the terminal can be used.

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.

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.

T $\rightarrow$ C: `GENERATE_AC` (TC, cda_requested, data specified by the CDOL1)
C $\rightarrow$ T: TC = (CID, ATC, MAC, [IAD])
where MAC = $\mathsf{MAC}_{SK_{IC}}$(amount, terminal country, terminal verification results, currency, date, transaction type, terminal nonce, AIP, ATC).

The card may refuse to do the transaction offline, and generate an ARQC cryptogram instead of the requested TC, forcing the terminal to go online.

In an online transaction the `EXTERNAL_AUTHENTICATE` command is used to pass the issuer's response to the ARQC back to the card. Alternatively, this response can be passed as a parameter of the next `GENERATE_AC` command.

4 Formalisation in F#

The complete formal model for card and terminal can be found online.[3] The issuer is not yet considered in the model. For the security properties considered in Section 5 we do not need the issuer.

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.

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;[4]
- `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: (either just a TC or AAC, or an ARQC followed by a TC, or an ARQC followed by an AAC).

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

[3] Available from `http://www.cs.ru.nl/~joeri/`

[4] The EMV specs apparently allow cards to support more than one card authentication mechanism, though this seems strange. Actual cards we observed always provide just one, namely SDA or DDA.

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: ($\text{nonce}_{Terminal}$)
- CDOL1: (amount, CVM Results, $\text{nonce}_{Terminal}$), where the *CVM Results (Cardholder Verification Method Results)* contains the result of the cardholder verification.
- CDOL2: ($\text{nonce}_{Terminal}$)
- 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.

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.
- `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 Primary Account Number.

Translation to ProVerif Using the FS2PV tool [4], our F# model can be translated to a model in the applied pi calculus, which can then be analysed using the ProVerif tool [5], as discussed in the next section. 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 duplications of large fragments of code in the applied pi calculus – and the (convenient) use of functions in F#.

In fact, initially we tried to formalise the EMV protocol in the applied pi calculus, but we gave up as the model became too complex to oversee.

5 Security Requirements

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

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 repetition in the right place.

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 To check for integrity and authenticity we make use of two types of ProVerif queries. First we use the type

```
ev:Event1() ==> ev:Event2()
```

This query checks if event `Event1` was executed, then event `Event2` was executed.

The second type of query to verify is

```
evinj:Event1() ==> evinj:Event2()
```

In this case, for every execution of event `Event1` there exists a distinct executed event `Event2`.

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 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 banking 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.[5]

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. 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 does not allow replays.

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

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 tells 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 by Murdoch et al. [12].

Transaction authenticity If a transaction is successfully completed by the terminal, the corresponding card should also agree on having the transaction completed 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.

When using CDA, the card and terminal should agree on the result of the transaction, i.e. if CDA is successful 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.

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 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.

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 [9, Page 88]. 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 man-in-the-middle could change the type of the message to a TC. This would result in the terminal and issuer accepting the transaction, tough 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 Dconstruction of the cryptograms.

7 Conclusions

We have presented a first formal model of the EMV protocol suite and demonstrated that it can be analysed using ProVerif in combination with FS2PV.

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 applied 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 specs. 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.

Our formal analysis reveals all the known weaknesses and limitations in the security of the EMV protocol suite, e.g. the possibilities

- to use cloned SDA cards for offline transactions,
- to fake transactions with DDA cards used offline, and
- to fool a terminal into thinking a PIN was correctly entered [12].

Moreover, these problems *inevitably* come to light when trying to establish the very generic security requirement that whenever a transaction is completed, all parties agree on the transaction's parameters (incl. whether it was with/without PIN, off- or online, on the amount, and on the card authentication mechanism used).

The most recent attack published on EMV is a man-in-the-middle attack that forces a rollback to unencrypted offline PIN, which then allows the plaincode PIN code to be observed [2]. (In this attack the CVM list provided by the card is modified; suprisingly, this attack may be possible even when the CVM list is signed, which it typically is, i.e. when the terminal can tell the CVM list has been modified.) This attack does not come up in our analysis, because our model excludes the complex process by which the terminal decides which Cardholder Verification Method to use, and simply always does offline unencrypted PIN.

Given the complexity of the EMV specifications, and the bewildering range of options and parameters, we hope that our formal model can be a useful tool in evaluating the security of different EMV configurations – or of other protocols, such as EMV-CAP [7], that have been defined on top of EMV.

Related Work Even though the EMV specifications are public, and obviously very important, there has been surprisingly little public scrutiny of them. Possibly the large size of the specifications, or the fact that they cannot be analysed without fixing some of the configurations and parameters, have discouraged people. A brief overview of EMV is given in [6], but this mainly discusses the cryptographic primitives and the PKI, and does not go into details about the actual protocol. EMV security is analysed in [13], but with the view to using it for internet payments.

Researchers at the university of Cambridge have been studying EMV for quite some time [1], e.g. investigating the possibility of relay attacks, the security of the EMV API in HSMs (Hardware Security Modules), and discovering a weakness in EMV configurations of some UK banks [12]. They also reverse-engineered and investigated EMV-CAP, a proprietary MasterCard protocol for internet banking and online payments that builds on EMV [7].

Future Work We still want to extend our model to include the issuer. Instead of using ProVerif in combination with FS2PV, we also want to experiment with verification using F7 [3]. This avoids the translation from F# to ProVerif, and might produce better and more predictable response times in verification.

Our F# models are executable. By supplying the right helper functions to deal with the low level details of smartcard communication, it should be possible to interact with real cards and terminals (i.e. have the F# terminal model interact with real EMV smartcards, and – using special but readily available hardware to provide a smartcard interface to a laptop – have the F# card model card interact with a real EMV terminal). This would provide the strongest possible guarantee that our formalisation is indeed correct, as well as a highly trustworthy terminal implementation.

Finally, it would also be interesting to see if the F# models would be useful for (model-based) testing of terminals and cards, or for security code reviews.

Acknowledgements Thanks to Karthikeyan Bhargavan and Andy Gordon for producing a patch for FS2PV. This work is supported by the European Commission, through the ICT programme under contract ICT-2007-216676 ECRYPT II, and the Netherlands Organisation for Scientific Research (NWO).

References

1. B. Adida, M. Bond, J. Clulow, A. Lin, S. Murdoch, R. Anderson, and R. Rivest. Phish and chips: Traditional and new recipes for attacking EMV. In *Security Protocols*, volume 5087 of *LNCS*, pages 40–48. Springer, 2009.
2. A. Barisani, D. Bianco, A. Laurie, and Z. Franken. Chip & PIN is definitely broken. Presentation at CanSecWest Applied Security Conference, Vancouver 2011. Slides available at `http://dev.inversepath.com/download/emv/emv\_2011.pdf`., 2011.
3. K. Bhargavan, C. Fournet, and A. Gordon. Modular verification of security protocol code by typing. *ACM SIGPLAN Notices*, 45(1):445–456, 2010.
4. K. Bhargavan, C. Fournet, and S. Tse. Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop (CSFW)*, pages 139–152. IEEE, 2006.
5. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE, 2001.
6. M. de Soete and M. Ward. EMV. In H. van Tilborg, editor, *Encyclopedia of cryptography and security*, pages 197–202. Springer, 2005.
7. S. Drimer, S. Murdoch, and R. Anderson. Optimised to fail: Card readers for online banking. *Financial Cryptography and Data Security*, 5628:184–200, 2009.
8. EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 1: Application Independent ICC to Terminal Interface Requirements, 2008.
9. EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 2: Security and Key Management, 2008.
10. EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 3: Application Specification, 2008.
11. EMVCo. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 4: Cardholder, Attendant, and Acquirer Interface Requirements, 2008.

12. S. Murdoch, S. Drimer, R. Anderson, and M. Bond. Chip and PIN is Broken. In *Symposium on Security and Privacy*, pages 433–446. IEEE, 2010.
13. E. Van Herreweghen and U. Wille. Risks and potentials of using EMV for internet payments. In *Proceedings of the 1st USENIX Workshop on Smartcard Technology*. USENIX Association, 1999.

Glossary

- AAC - Application Authentication Cryptogram
- AC - Application Cryptogram, which can be an AAC, ARCQ, or TC
- AFL - Application File Locator; identifies files on the card, and indicates whether their content is included in the SSAD
- AIP - Application Interchange Profile; indicates which authentication options the card supports
- ARQC - Authorisation Request Cryptogram
- ATC - Application Transaction Counter
- CAM - Card Authentication Method
- CVM - Cardholder Verification Method
- CDA - Combined Data Authentication
- CDOL - Card Risk Management DOL
- CID - Cryptogram Information Data; indicates the type of the cryptogram and the actions to be performed by the terminal
- DDA - Dynamic Data Authentication
- DDOL - Dynamic Data Authentication DOL
- DOL - Data Object List
- IAD - Issuer Application Data; proprietary data to be sent to the issuer
- MAC - Message Authentication Code
- PAN - Primay Account Number
- PDOL - Processing Options DOL
- SDA - Static Data Authentication
- SSAD - Signed Static Application Data; used in SDA
- SDAD - Signed Dynamic Application Data; used in DDA and CDA
- TC - Transaction Certificate
- TDHC - Transaction Data Hash Code; used in CDA
- TDOL - Transaction Certificate DOL