Formal models of bank cards for free

Fides Aarts, Joeri de Ruiter, and Erik Poll
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{f.aarts,joeri,erikpoll}@cs.ru.nl

***Abstract*—Learning techniques allow the automatic inference of the behaviour of a system as a finite state machine. We demonstrate that learning techniques can be used to extract such formal models from software on banking smartcards which – as most bank cards do – implement variants of the EMV protocol suite. Such automated reverse-engineering, which only observes the smartcard 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 smartcard applications in general – as they can show unexpected additional functionality that is easily missed in conformance tests.**

I. Introduction

Software for banking 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 [1], 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 paper constructs models just by observing the smartcard's external behaviour.

A widely used technique for creating a model from observations is *regular inference*, also known as automata learning [2]. The regular inference algorithm [3], [4] provides sequences of inputs to a System Under Test (SUT) and observes the responses to infer a Mealy machine, a special form of finite state machine, as explained in Section III. In addition to standard learning methods, abstraction techniques for data parameters [5], [6] can be used to learn a more detailed model of the system.

We show that these techniques can provide useful formal models of bank cards quickly without much effort, which can be a useful addition to testing or security evaluations of these products. The technique is not just applicable to bank cards, but can be applied to any smartcard.

II. Background: smartcards and EMV

A. Smartcards

All smartcards follow the ISO/IEC 7816 standard [7]. Here communication is in master-slave mode: the terminal sends a command to the card, and the card returns a response, after which the terminal can send another command, etc. Commands and responses are simply sequences of bytes with a fixed format and meaning, called APDUs (Application Protocol Data Units).

The second byte in a command APDU is the *instruction byte*, and specifies the instruction that the smartcard is requested to perform. The last two bytes of a response APDU are the *status word*, which indicates if execution of the command went OK or if some error occurred. The ISO/IEC 7816 standard defines some standard instruction bytes and error codes.

Standard instructions we used to infer the behaviour of bank cards include:

- the `SELECT` instruction to select which of the possibly several applications on the smartcard the terminal wants to interact with;
- the `VERIFY` instruction to provide a PIN code to the card for authentication of the cardholder;
- the `READ RECORD` instruction to read some data from the simple file system that the card provides;
- the `GET DATA` instruction to retrieve a specific data element from the card (for example the PIN try counter, which records how often the PIN can still be guessed);

- the `INTERNAL AUTHENTICATE` instruction to authenticate the card; the terminal supplies a random number as argument to this command which the smartcard then encrypts or signs to prove knowledge of a secret key.

For the purposes of this paper the difference between the 'files' retrieved using `READ RECORD` and the 'data elements' retrieved using `GET DATA` is not important.

B. EMV

Most smartcards issued by banks or credit cards companies adhere to the EMV (Europay-MasterCard-Visa) standard [8]. This standard is defined on top of ISO/IEC 7816. It uses some of standard instruction bytes (incl. those listed above), but also defines additional ones specific to EMV, including:

- the `GENERATE AC` instruction to let the card generate a so-called Application Cryptogram (AC);
- the `GET PROCESSING OPTIONS` instruction to initialise the transaction, provide the necessary information to the card and retrieve the capabilities of the card.

A normal EMV session consists of the following steps:

1) *selection of the desired application on the smartcard using `SELECT`.* There may be several applications on a smartcard. Some bankcards will provide multiple EMV-applications for different uses, e.g. one to be used by ATMs and one to be used by a hand-held reader for internet banking.
2) *initialisation of the transaction using `GET PROCESSING OPTIONS`.* The terminal provides the card with data, specified in the response to the selection of the application. The card initialises the transaction and sends a response containing its capabilities.
3) *optionally: cardholder verification and/or card authentication.* Card authentication can, for example, be done using a challenge-response mechanism (called DDA in the EMV standard) by invoking the `INTERNAL AUTHENTICATE` instruction. Cardholder verification can be done offline by checking the PIN code using the `VERIFY` instruction; here the PIN can be sent to the smartcard either in plaintext or encrypted. Checking the PIN can also be done online, in which case the PIN is sent to the bank back-end to check it.
4) *the transaction.* For the actual transaction one or two cryptograms are requested using the `GENERATE AC` instruction, as discussed in more detail below.
5) *scripting.* After completing a transaction, the terminal may send additional Issuer-to-Card scripting commands, that allow the issuer to update cards in the field.

The cryptograms generated for a transaction can have one of the following three types:

- an Authorisation Request Cryptogram (ARQC), which is a request to perform a transaction online;
- a Transaction Certificate (TC), which indicates acceptance of a transaction;
- an Application Authentication Cryptogram (AAC), which indicates rejection of a transaction.

All these cryptograms contain a MAC (Message Authentication Code), a hash over some data encrypted with a secret key.

An EMV transaction involves at most two of these cryptograms. The types of these cryptograms depend on the transaction. EMV transactions can be offline or online. For an offline transaction, the terminal sends data about the transaction to the card, and the card returns a TC to approve the transaction. For an online transaction, the card first provides an ARQC that is sent back to the issuing bank. The bank's response is sent to the card, which will then return a TC if the response is correct. Every transaction by the card is identified by a unique value of the Application Transaction Counter (ATC), that the card keeps track of.

The terminal requests these cryptograms using the `GENERATE AC` command. The terminal will indicate which type of cryptogram it wants, but the card may return a different type than requested. For example, when the terminal requests a TC, the card may return an ARQC (namely in case the card wants the terminal to go online to get approval for the transaction from the bank) or it may decline the transaction by responding with an AAC.

The EMV protocol as described above is also used for internet banking in the EMV-CAP protocol. Here bank customers use a handheld smartcard reader with a small display in which they insert their bank card. EMV-CAP is a proprietary standard of MasterCard. Unlike the EMV specs, the EMV-CAP specs are not public, but they have been largely reverse-engineered [9], [10]. In EMV-CAP the card is requested for an ARQC to authorise an transaction (e.g. an internet bank transfer). This is then followed by a request for an AAC, thus completing (or, more precisely, aborting) a regular EMV transaction so that the card is left in a 'clean' state.

III. Background: Inference of Mealy Machines

A. Mealy Machines

We use *Mealy machines* to model the behaviour of smartcards. 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 $\lambda(q, i) = o$.

We extend the transition and output functions from input symbols to input strings in the standard way, by defining:

$$\begin{array}{lcl} \delta(q,\varepsilon) & = & q \\ \delta(q,ui) & = & \delta(\delta(q,u),i) \\ \lambda(q,\varepsilon) & = & \varepsilon \\ \lambda(q,ui) & = & \lambda(q,u)\lambda(\delta(q,u),i) \end{array}$$

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.

Given a Mealy machine $\mathcal{M}$ with input alphabet I, output function λ, and initial state q^0, we define $\lambda_{\mathcal{M}}(u) = \lambda(q^0,u)$, for $u \in I^*$. Two Mealy machines $\mathcal{M}$ and $\mathcal{M}'$ with input alphabets I are *equivalent* if $\lambda_{\mathcal{M}}(u) = \lambda_{\mathcal{M}'}(u)$ for all input strings $u \in I^*$.

B. Inference of Mealy Machines

Angluin's well-known L* algorithm [2] is an active learning algorithm to infer deterministic finite automata. Inspired by work of Angluin, Niese [3] 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*, see Fig. 1:

- 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 hypothesized 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., $\lambda_{\mathcal{M}}(u) \neq \lambda_{\mathcal{H}}(u)$.

The equivalence query cannot be really supported if the actual machine $\mathcal{M}$ can only be observed as a black box, as is the case in our work, as we analyse smartcards without access to the program code. 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 explained below, there are different strategies to test this.

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 hypothesized automaton, which is supplied in an equivalence query, etc.

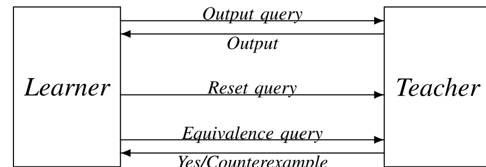

Fig. 1. Learning framework

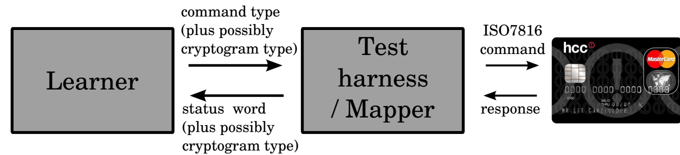

Fig. 2. Set-up

IV. Setup and procedure

We used authentic bank cards as *SUT*/*Teacher*. Access to the smartcards was realised via a standard smartcard reader and a testing harness discussed in Section IV-A. We connected the *SUT* to the LearnLib library [4], which served as *Learner*, see Fig. 2. The LearnLib tool provides a Java implementation of the adapted L* algorithm. Because LearnLib views the *SUT* as a black box, equivalence queries can only be approximated. The tool provides several different realisations of equivalence queries. In our experiments we used a random test suite with 1000 test traces of length 10 to 50 as equivalence oracle. We verified our results with the W-Method by Chow [11] by checking if it will find at least one more state than the random test suite.

For our tests 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. The Visa branded debit card contains the Visa Debit application.

A. Test harness

As illustrated in Fig. 2, our test harness[1] 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). It supports the commands listed in Section II-B.

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 smartcard reader. A regular smartcard 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 some command APDU to be sent to the card. The input alphabet corresponds to these 15 methods.

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

For many parameters of these commands the test harness uses some fixed value, for instance for the random number sent as argument of `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 2 sequences of arguments (one for the first and one for the second cryptogram).

The test harness does not output the entire response of the smartcard 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 are 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 for this, we were unfortunately not able to use `VERIFY` with this card.

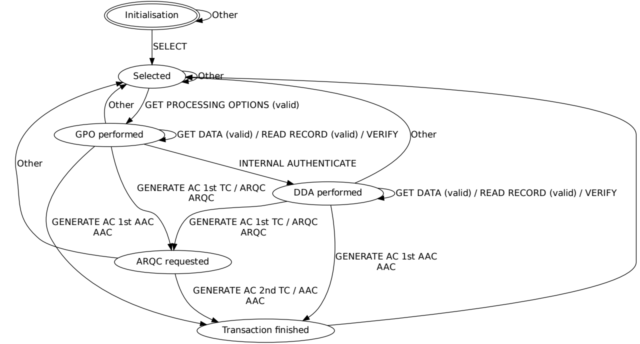

Fig. 3. Automaton of Dutch Maestro application. Just to highlight one observation that can be made from this diagram: the `VERIFY` operation, i.e. the verification of the PIN code by the smartcard, is optional; this makes sense because the terminal may check the PIN code with the bank (so-called online PIN verification), or choose not to verify the PIN at all.

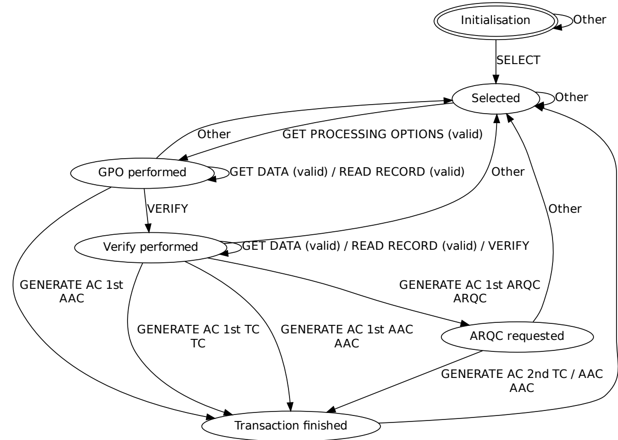

Fig. 4. Automaton of Dutch SecureCode Aut application. Note that here the `VERIFY` operation – i.e. verification of the PIN code – must be passed successfully before cryptograms can be generated, except for the AAC cryptogram to abort the session.

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 [5], [6] 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.

B. 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 3, 4, 5 and 6. 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, 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.

V. RESULTS

We learned models of EMV applications on bank cards issued by several Dutch banks (ABN-AMRO, ING, Rabobank) and one German bank (Volksbank), and on MasterCard credit cards issued by Dutch and Swedish banks (SEB, ABN-AMRO, ING) and of one UK Visa debit card (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 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.

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 (Fig. 3) and the SecureCode Aut application (Fig. 4) 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 (as explained at the end of Section II). 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 [9], [10].
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 Fig. 5):

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` 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 it's 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.

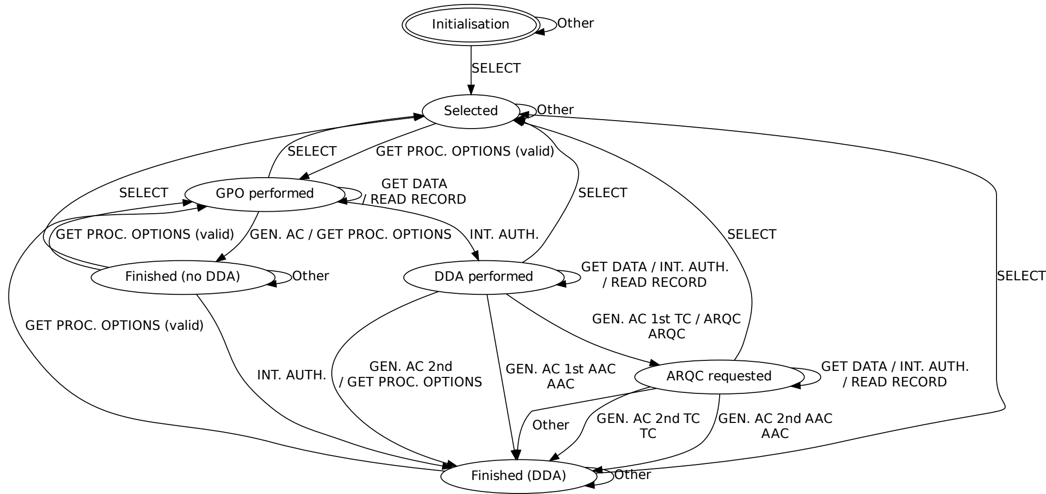

Fig. 5. Automaton of Maestro application on Volksbank bank card

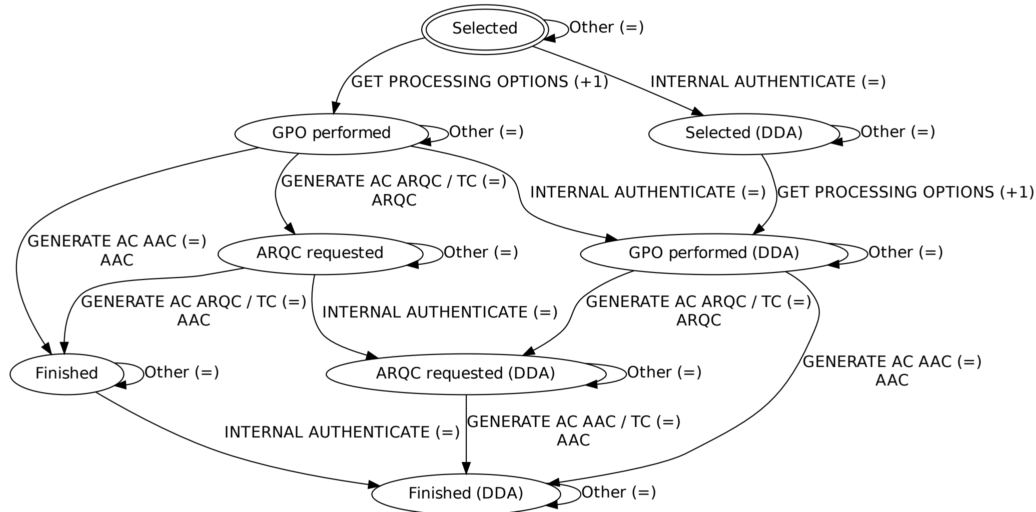

Fig. 6. Automaton of Visa Debit application on Barclays card. Note that the `INTERNAL AUTHENTICATE` can be performed at any stage of the protocol.

an AAC to be returned), as we did not provide a valid response from the bank.

A. Difference with MasterCard's specifications

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

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 does suggest that this process does not include checking

[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 [12, p. 98].

for implementation of the exact state machine.

B. *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 smartcard. 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).

VI. Related work

Protocol fuzzing is an increasingly popular technique to test for security vulnerabilities. Simpler forms of protocol fuzzing consider only the format of messages, and then fuzz the different fields, often simply to try and crash an implementation. More advanced fuzzers, such as Snooze [13] and Peach[3], take a state-based approach and also use a state machine describing the protocol as basis for fuzzing. Models such as we obtain could be the basis for more thorough state-based fuzzing by such tools. Model-based testing has already been applied to security, including for smartcards, for instance using UMLSec models [14]. For EMV smartcards, there have been successful experiments with protocol fuzzing based on state models at a commercial test lab [15]; here models were constructed by hand.

There is a growing interest in model inference, or more generally automated protocol reverse engineering, for security testing and analysis; see [16] for a survey and a proposed classification of approaches, and [17] for a discussion of future directions in combining model inference and model-based testing for security. In automated techniques for protocol reverse engineering one can distinguish approaches that try to infer either message formats (e.g. [18]) or protocol state machines (as we do, and [19]), or both (e.g. [20]). Another classification is that some approaches use 'passive learning', where the learner just observes traffic between other parties (e.g. [18], [19], [20]), and 'active learning', i.e. where the learner actively takes part in the traffic in order to learn (as we do). A fundamental limitation of passive learning is that the quality of the model depends on the traffic that is observed. It will typically not provide good insight into the possibility of unwanted behaviour. It is therefore natural to follow such passive learning by protocol fuzzing to actively look for any such behaviour. Indeed, fuzzing based on the inferred model is considered as final stage in [19], [20].

Active learning was also successfully used to infer models for the electronic passport [6], confirming that the right 'files' are accessible at different stages of the protocol. The models inferred in this paper are more complex than the one of the passport, and the models reveal interesting differences between various cards. Additionally, we could learn which command increased the ATC using a mapper component.

VII. Conclusions

We have demonstrated that after defining a simple test harness/mapper component, we can easily obtain useful state machine models for banking smartcards using learning and simple abstraction techniques [4], [5]. 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 specs, but can really affect the security conditions imposed (for example, the difference between figures 3 and 4 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.

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 make 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 specs [8] (of over 700 pages), state diagrams describing the smartcard are conspicuously absent. A state diagram is specified in MasterCard's specification [12], but most of the cards we analysed actually did not conform to it. The differences between e.g. figures 3 and 5 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 smartcards 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 smartcards 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

[3]http://peachfuzzer.com

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 smartcard code would.

For future work, we want to try out our technique on more standard networking protocols such as SSH or TLS/SSL. This might be more fruitful in the sense that we can expect implementation bugs to be more common here, as these protocols are more complex and the code is less rigorously developed and tested than smartcard code. In the field of EMV, we plan to see if learning techniques can be used to assess EMV test suites provided by commercial testing companies; models learned from such test suites, using passive rather than active learning, could provide coverage criteria to assess their quality.

Acknowledgements

This work is supported by the Netherlands Organisation for Scientific Research (NWO) and by STW project 11763 Integrating Testing And Learning of Interface Automata (ITALIA).

References

[1] M. Felderer, B. Agreiter, P. Zech, and R. Breu, "A classification for model-based security testing," in *Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011, pp. 109–114.

[2] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, 1987.

[3] O. Niese, "An integrated approach to testing complex systems," Dortmund University, Tech. Rep., 2003, doctoral thesis.

[4] H. Raffelt, B. Steffen, T. Berg, and T. Margaria, "LearnLib: a framework for extrapolating behavioral models," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, pp. 393–407, 2009.

[5] F. Aarts, B. Jonsson, and J. Uijen, "Generating models of infinite-state communication protocols using regular inference with abstraction," in *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ser. ICTSS'10, 2010, pp. 188–204.

[6] F. Aarts, J. Schmaltz, and F. Vaandrager, "Inference and abstraction of the biometric passport," in *International Symposium on Leveraging applications of formal methods, verification, and validation (ISoLa'10)*, 2010, pp. 673–686.

[7] ISO/IEC, "ISO/IEC 7816: Identification cards – Integrated circuit cards."

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

[9] S. Drimer, S. Murdoch, and R. Anderson, "Optimised to fail: Card readers for online banking," in *Financial Cryptography and Data Security*, ser. LNCS, vol. 5628. Springer, 2009, pp. 184–200.

[10] J.-P. Szikora and P. Teuwen, "Banques en ligne: à la découverte d'EMV-CAP," *MISC (Multi-System & Internet Security Cookbook)*, vol. 56, pp. 50–62, 2011.

[11] T. Chow, "Testing software design modeled by finite-state machines," *Software Engineering, IEEE Transactions on*, vol. 4, no. 3, pp. 178–187, 1978, special issue on COMPSAC.

[12] M. I. Inc., "Paypass - m/chip technical specifications," Tech. Rep., September 2005, version 1.3.

[13] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: toward a stateful network protocol fuzzer," *Information Security*, pp. 343–358, 2006.

[14] J. Jürjens, "Model-based security testing using UMLsec:: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 93–104, 2008.

[15] J. Lancia, "Un framework de fuzzing pour cartes à puce: application aux protocoles EMV," in *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, 2011.

[16] X. Li and L. Chen, "A survey on methods of automatic protocol reverse engineering," in *Computational Intelligence and Security (CIS 2011)*. IEEE, 2011, pp. 685–689.

[17] K. Hossen, R. Groz, and J. L. Richier, "Security vulnerabilities detection using model inference for applications and security protocols," in *Software Testing, Verification and Validation Workshops (ICSTW'11)*. IEEE, 2011, pp. 534–536.

[18] W. Cui, J. Kannan, and H. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *USENIX Security Symposium (Security07)*, 2007.

[19] Y. Hsu, G. Shu, and D. Lee, "A model-based approach to security flaw detection of network protocol implementations," *2008 IEEE International Conference on Network Protocols*, vol. 40, no. 2, pp. 114–123, 2008.

[20] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Security and Privacy*. IEEE, 2009, pp. 110–125.