Automated Reverse Engineering using Lego®

Georg Chalupar[1], Stefan Peherstorfer[1], Erik Poll[2], and Joeri de Ruiter[2]

[1]University of Applied Sciences Upper Austria
[2]Radboud University Nijmegen, The Netherlands

Abstract

State machine learning is a useful technique for automating reverse engineering. In essence, it involves fuzzing different sequences of inputs for a system. We show that this technique can be successfully used to reverse engineer hand-held smartcard readers for Internet banking, by using a Lego robot to operate these devices. In particular, the state machines that are automatically inferred by the robot reveal a security vulnerability in one such a device, the e.dentifier2, that was previously discovered by manual analysis, and confirm the absence of this flaw in an updated version of this device.

1 Introduction

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. This makes state machine learning 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.

This paper discusses the use of state machine learning to reverse engineer smartcard readers for Internet banking, more in particular different versions of the e.dentifier2, a USB-connected smartcard reader that a customer operates using his bank card and PIN code. The smartcard reader has a keyboard for the user to enter his PIN code and buttons to cancel or confirm Internet banking transactions. A security vulnerability was discovered in this device by a manual analysis of the USB communication between the PC and the device and the communication between the device and the smartcard. The vulnerability could be exploited by injecting malicious USB traffic [6]. Goal of this research was to see if we could automate such an analysis. To be able to learn the behaviour of the reader, we therefore constructed a Lego robot, controlled by a Raspberry Pi, that can operate this keyboard. Controlling all this from a laptop, we then could use LearnLib [15], a software library for state machine inference, to learn the behaviour of readers. We show that the state machine that can be learned using Lego robot reveals the presence of the security flaw, and shows that the flaw is no longer present in the new version of the device.

2 Background

This section describes the Internet banking devices we analysed, and the state machine learning technique used for this.

2.1 The e.dentifier2

The e.dentifier2 is a hand-held smartcard reader with a small display, a numeric keyboard, and OK and Cancel buttons (see Figure 1). It is used for Internet banking in combination with a bank card and a PIN code. The device can be connected to a PC by USB or used without USB connection. The analysis in this paper concerns the USB-connected mode.

Like most card readers used for online banking, both the connected and unconnected modes of the e.dentifier2 appear to be based on EMV-CAP, a proprietary standard by MasterCard, which in turn is based on the EMV standard [10] implemented in most bank cards. The behaviour of EMV-CAP readers for Internet banking has

Figure 1: The e.dentifier2

been largely reverse engineered [9, 19]. In more detail, during an EMV-CAP transaction, two MACs (Message Authentication Codes) are generated by the bank card using a secret key that is shared with the bank. Usually, a bit filter is then applied on the first MAC and the Application Transaction Counter of the card to get an 8 digit number used to confirm the transaction. However, the e.dentifier2 is slightly different from the devices discussed in [9, 19], as some additional transformation is performed in the device before the confirmation code is returned.

When the device is used without USB connection, the user has to copy numeric challenges shown on the bank web page onto the reader, after he inserted his bank card and typed in his PIN code. The display then shows a response, which is based on the response of the bank card, that has to be entered in the web page to confirm the transaction.

When the device is used with a USB connection, it is controlled by the browser using JavaScript via a plugin. The user then does not need to manually copy numeric challenges and responses, and more detailed (alphanumeric) information is given on the display of the device, in particular details of any bank transfer being approved. In principle, this is more secure, as it provides WYSIWYS (What-You-Sign-Is-What-You-See): the user can understand what transaction he is approving. When the user only sees meaningless random challenges, he can easily be misled by Man-in-the-Browser attacks. Unfortunately, a vulnerability in the device means that user approval of transactions – by pressing the OK button on the device – could actually be by-passed by malware sending USB commands at unexpected stages of the protocol [6].

After discovery of the vulnerability, a new version of the device was released. We used both the old and the new version in order to see if such flaws can be automatically discovered. To be precise, we used e.dentifiers with version numbers `F/W 01.02 H/W C Dec 19 2007 18:39:42` and `F/W 01.05 H/W C Feb 7 2012A 14:54:39`; this firmware version is shown on the display if the button 5 is pressed while a smartcard is inserted.

2.2 LearnLib

The behaviour of our target – the e.dentifier2 device – can be modelled as a Mealy machine. Mealy machines are deterministic finite automata (DFAs), which produce an output on every state transition.

A Mealy machine $\mathscr{M}$ is specified by the tuple $\langle I, O, S, s_0, \delta, \lambda \rangle$, consisting of the *input alphabet* I, the *output alphabet* O, the states S, the *initial state* $s_0 \in S$, the *transition function* $\delta : S \times I \to S$, and the *output function* $\lambda : S \times I \to O$. We define *input* and *output symbols* as elements of I and O and *input* and *output strings* as elements of I^* and O^*.

We use LearnLib [15] to learn the Mealy machine implemented by the e.dentifier2. LearnLib uses a version of Angluin's L^* algorithm [4] optimised for Mealy machines. In the L^* algorithm, a *learner* infers a Mealy machine with the help of a *teacher*, who is assumed to know the Mealy machine we want to learn.

In essence, this is done by randomly trying different sequences of inputs, and observing the output; if including an input makes a difference in subsequent input/output responses, it is inferred that this input must have caused an internal state change.

In more detail, this is done by creating a hypothesis and iteratively refining it in a two-step process:

- First, the learner chooses input strings and obtains the output strings produced by the target from the teacher using so called *output queries*. This way, it tries to discover different states using *distinguishing strings*, i.e. suffixes that produce different outputs for different states. For each discovered state, the algorithm stores an *access sequence*, i.e. an input string that reaches this state from the initial state. When every input symbol in every known state results in a transition to another known state, the model is *consistent* and a hypothesis automaton is generated.

- Next, the learner sends an *equivalence query* to the teacher to check whether the hypothesis is equal to the target automaton. If this is not the case, the teacher replies with a counter example, i.e. an input string which results in different outputs for the target and the hypothesis automaton.

The learner takes a suffix of the counter example as additional distinguishing sequence, which will lead to a new state in the next round and updates the model to produce the correct result for the counter example. The learning process continues as long as the teacher is able to find a difference between the hypothesis and the target.

We have to provide LearnLib with an input alphabet consisting of abstract input symbols. These input symbols are translated by our software to real USB and robot commands, which we will discuss in more detail in Section 4.1. The responses returned via the USB connection are translated back into abstract output symbols before being returned to LearnLib.

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 the target, 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 what is essentially a form of model-based testing.

In our case, where we do not know the implementation of the e.dentifier2, we have to resort to the latter approach. Note that in such a black-box setting, equivalence queries cannot be answered with certainty: there may be differences between our hypothesis and the real implementation that do not show up in the finite number of tests that we 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 our hypothesis automaton on the millionth time. The automaton that is ultimately inferred might only show a subset of the actual behaviour. To approximate equivalence checking of the teacher, we use either the random walk method or Chow's W-method [8] 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, just like a minimum and maximum trace length. The W-method can guarantee 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.

From a security perspective, it means that we cannot hope to discover carefully hidden backdoors in an implementation. Still, as our experiments with the e.dentifier2 confirm, we can hope to find accidental mistakes in the program logic.

3 Setup

To learn the state machine of the e.dentifier2, as input alphabet we use the USB commands that are sent to the e.dentifier2 from the PC, plus the keyboard actions of pressing OK, pressing Cancel, or entering the PIN code. The output alphabet is the set of USB responses that are sent back to the PC.

We could also have included the output to the display and the output to the smartcard as observable outputs of the device, but it turns out that the USB responses are informative enough to successfully infer the state machine of the device.

3.1 Lego Robot

We did not want to physically open the e.dentifier2 but decides to interact with it through the normal keyboard. This meant we needed a robot that actually presses the buttons on the device. By using Lego motors, we found a cheap and flexible solution (see Figure 2). We use a small gear on the motors to move a Lego bar via a toothed rack.

The robot needs to be able to enter PIN codes and press OK and Cancel. We used a PIN with 4 times the same digit, so that only one 'finger' is needed to enter the PIN. In total, we then only needed three fingers and three Lego motors to move them up and down.

We choose a Raspberry Pi to actuate the motors as it offers some GPIO (General Purpose Input/Output) 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 we use 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.

In order to control the motors, some electronic circuitry between the Raspberry Pi and the motors is needed. To keep the setup as simple as possible, a so called H bridge is used for this, rather than building our own circuitry. An H bridge controls the direction of the current and thus the direction in which the motor turns. The Texas Instruments SN754410 ICs used each provide two H bridges. Each motor is controlled using one H bridge, which is connected to two GPIO pins on the Raspberry Pi.

The e.dentifier2 has to be reset after each LearnLib query. For this, we spliced the USB cable to take out the power wire, and then used another H bridge to be able to

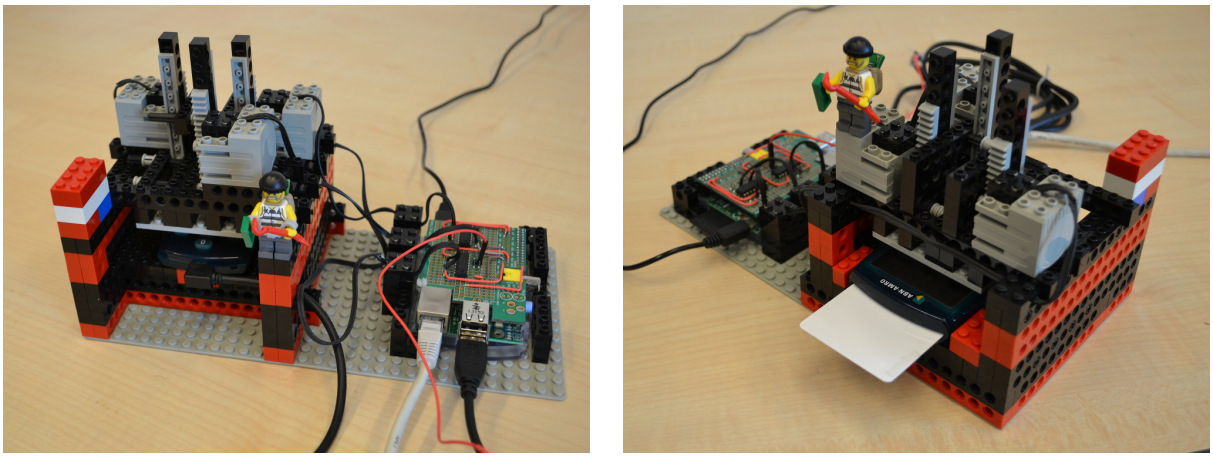

Figure 2: Our Lego robot is capable of pressing three buttons on the e.dentifier2. The learning setup includes the Lego robot, circuitry to power the engines, and a Raspberry Pi that interacts with the e.dentifier2 and controls the engines.

turn the USB power supply on and off.

In our experiments, we used a smartcard 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 identical responses for each EMV request. A real bank card has an Application Transaction Counter (ATC), which is included in the MAC and increased after each request, and the resulting change in the output would have to be filtered out to learn the state machine with LearnLib.

3.2 Software

We created a small program in Java on top of LearnLib (based on the one used in [2]) to create output queries and build a model from the responses. For the equivalence checking, we used either the random walk method or the W-method. To create reproducible results we used a static seed for the random number generator. The input alphabet consists of command names that represent either USB messages or commands for the Lego robot. A class implementing the `Oracle` interface answers output queries by sending the commands to a script on the Raspberry Pi over a TCP socket. This script controls the motors and resets the device. Additionally, it sends the USB commands to the e.dentifier2 using the PyUSB library.

Especially in longer experiments, we experienced problems with non-deterministic behaviour due to, for example, buttons not always correctly being pressed. To counter this, we made use of an alternative version of the software using majority voting were 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.

4 Experiments

4.1 Input and Output Symbols

We based the input symbols on the USB commands discovered by previous reverse engineering [6] (for their byte values, see Table 1 in Appendix A). 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 smartcard.

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.

We used the USB responses (see Table 2 in Appendix A) of the e.dentifier2 as output symbols for the learning process. The smartcard 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 indicated in [6]. 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 smartcard 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 `0103031900000` (in case of the old e.dentifier2) or `0103031b00000` (in case of the new version) 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.

4.2 Learning Parameter and Timings

At the beginning of every input sequence, the e.dentifier2 has to be reset to an initial state. Since resetting includes waiting for 0.5 seconds before powering down to ensure that all previous operations have finished, 3 seconds before powering it again to ensure that it is completely reset, and 2 seconds to wait for the device to start, this takes on average 5.6 seconds. ROBOT_OK and ROBOT_CANCEL take on average 2.5 seconds, because the robot waits for 1 second before pressing the button (to ensure the e.dentifier is ready to accept a button after the last command), then it takes the robot 0.4 seconds to press the button and the Raspberry waits for 1 second to receive a response via USB. USB commands take on average 1.1 seconds which is mostly due to the timeout when waiting for a response. Since COMBINED_INIT and COMBINED_DATA consist of two USB commands, they take twice as long as a single USB command. COMBINED_PIN includes a USB command and 5 button presses (4 PIN digits followed by OK) which results in an average duration of 4.4 seconds.

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. The learner generated 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 with 50 random 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 3 and 4, already have enough detail to show the flaw in the old version and reveal differences between the old and new version.

- Next, we learned a more detailed 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. 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.

5 Results

Figure 3 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 2.1 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 (see figure 4) of the e.dentifier2 does not have this additional state *unconfirmed cryptogram*. The device thus misses this (superfluous) state which causes the security vulnerability. This shows that the vulnerability present in the old version is 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 4. This leads to more states and paths as shown in figure 5. 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[1], 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.

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

6 Model checking

We easily converted the models, that result from the automated learning process, to the Aldebaran file format for labelled transition systems. These files were then used as input for the CADP model checker to automatically search for possible vulnerabilities [11]. As the models get increasingly complex when learning with a large input alphabet, it becomes difficult to review them manually. Model checkers can then be very helpful to verify security properties. However, coming up with all the security properties that need to be checked and specifying them correctly is not a trivial task.

7 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 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.

8 Future Work

Additionally, it would be interesting to investigate whether more messages are implemented by the e.dentifier2, how message parameters are handled, and

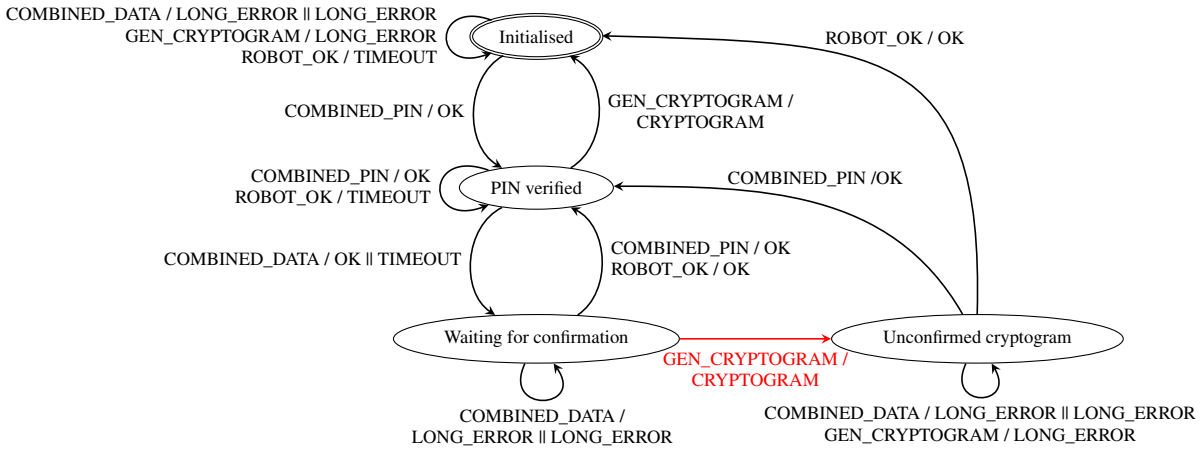

Figure 3: The learned result of the old version of the e.dentifier2. The initial state is marked with a double ellipse. For readability, multiple transitions between the same pair of states with different labels have been merged to one transition with these different labels separated by "||" (e.g. OK || TIMEOUT).

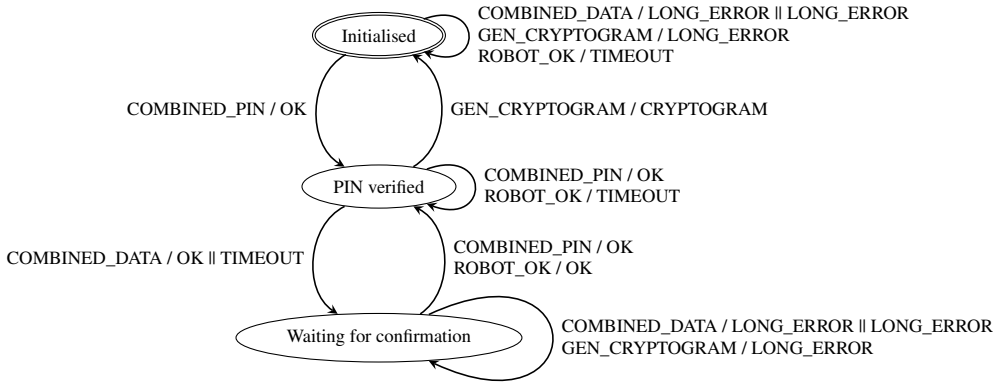

Figure 4: The learned result of the new version of the e.dentifier2.

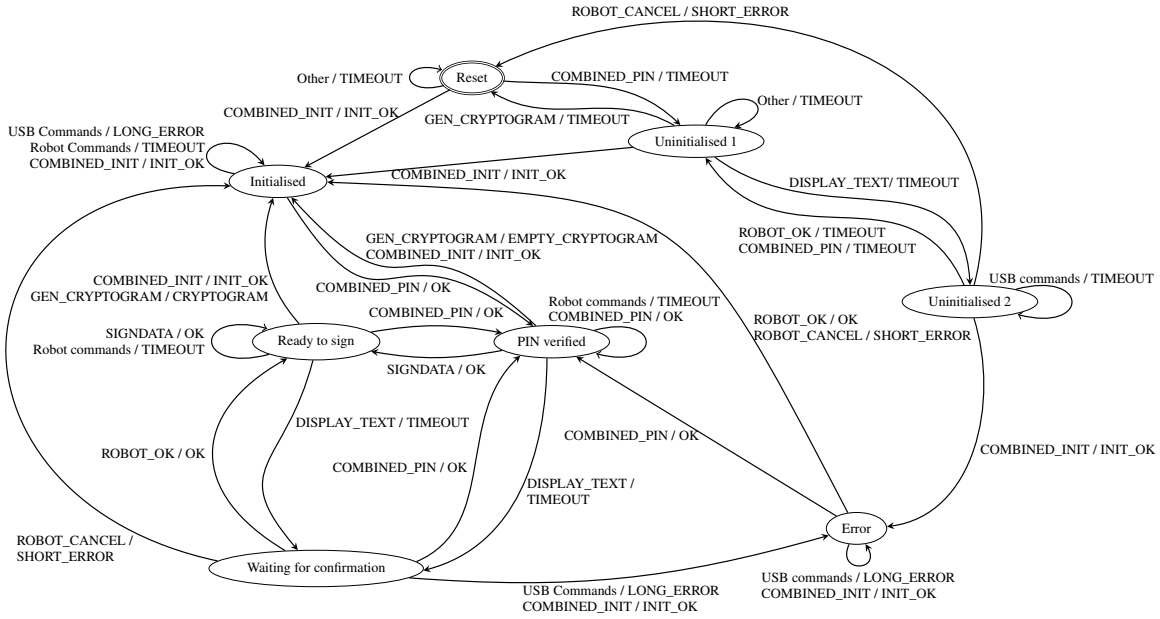

Figure 5: The learned result of the new version of the e.dentifier2 with a larger input alphabet using the W-method for 8 states.

determine whether other vulnerabilities exist as well. This could be done by fuzzing USB messages, for which our setup could be used as a basis by providing a state machine that can be used to perform targeted fuzzing.

The same approach could be applied to analyse other tamper resistant devices that need input via buttons, not only for automated learning but also, for example, when fuzzing or looking for weak random number generators. It would, for example, be interesting to apply automated learning techniques to ATMs or Point-of-Sale terminals given the complexity and potential for trouble with these devices [14, 5].

9 Related work

State machine learning can be seen as an automatic reverse engineering approach or as an advanced fuzzing technique. As a fuzzing technique, it provides a next step beyond protocol fuzzing [13, 1]: in protocol fuzzing, the *contents* of (fields in) messages are fuzzed, in state machine learning the *order* of messages is fuzzed. State machine learning has been used for network protocol analysis, to find flaws in implementations of known protocols [17] but also to reverse engineer unknown protocols (e.g. of botnets [7]). Reverse engineering of network protocols has also been used to simulate services for use in honeypots [12]. It has also proved capable of reverse engineering smartcard implementations of bank cards [2] and electronic passports [3]. This work on applying state machine learning to authentication tokens that use a smartcard is in a sense a logical next step. Note that state machine learning has been used on far more complex software systems, e.g. software in photocopiers [18], resulting in automata with thousands of states.

A Lego robot was used as an offensive technology before to test the security of touch-based authentication [16].

10 Conclusions

Using the LearnLib standard software library for state machine learning and our Lego robot, we can automatically reverse engineer the state machines of smartcard readers for Internet 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. 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 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) 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 now longer generates intermittent errors in the USB traffic.

References

[1] Code archive of the Sulley Fuzzing Framework. `https://github.com/OpenRCE/sulley`.

[2] AARTS, F., DE RUITER, J., AND POLL, E. Formal models of bank cards for free. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (2013), pp. 461–468.

[3] AARTS, F., SCHMALTZ, J., AND VAANDRAGER, F. 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.

[4] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Inf. Comput. 75*, 2 (1987), 87–106.

[5] BARISANI, A., BIANCO, D., LAURIE, A., AND FRANKEN, Z. Chip & PIN is definitely broken. Presentation at CanSecWest Applied Security Conference, Vancouver 2011. More info available at `http://dev.inversepath.com/download/emv`, 2011.

[6] BLOM, A., DE KONING GANS, G., POLL, E., DE RUITER, J., AND VERDULT, R. Designed to fail: A USB-connected reader for online banking. In *17th Nordic Conference on Secure IT Systems (NordSec 2012)* (2012), vol. 7617 of *LNCS*, Springer.

[7] CHO, C., SHIN, E., SONG, D., ET AL. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on Computer and Communications Security* (2010), ACM, pp. 426–439.

[8] CHOW, T. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering 4*, 3 (1978), 178–187.

[9] DRIMER, S., MURDOCH, S., AND ANDERSON, R. Optimised to fail: Card readers for online banking. In *Financial Cryptography and Data Security* (2009), vol. 5628 of *LNCS*, Springer, pp. 184–200.

[10] EMVCO. EMV– Integrated Circuit Card Specifications for Payment Systems, Book 1-4, 2008. Available at `http://emvco.com`.

[11] GARAVEL, H., LANG, F., MATEESCU, R., AND SERWE, W. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *Proc. TACAS'11* (2011), vol. 6605 of *LNCS*, Springer, pp. 372–387.

[12] KRUEGER, T., GASCON, H., KRÄMER, N., AND RIECK, K. Learning stateful models for network honeypots. In *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence* (New York, NY, USA, 2012), AISec '12, ACM, pp. 37–48.

[13] MULLINER, C., GOLDE, N., AND SEIFERT, J.-P. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *USENIX Security Symposium* (2011).

[14] MURDOCH, S., DRIMER, S., ANDERSON, R., AND BOND, M. Chip and PIN is Broken. In *Symposium on Security and Privacy* (2010), IEEE, pp. 433–446.

[15] RAFFELT, H., STEFFEN, B., BERG, T., AND MARGARIA, T. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf. 11* (2009), 393–407.

[16] SERWADDA, A., AND PHOHA, V. V. When kids' toys breach mobile phone security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (2013), CCS '13, ACM, pp. 599–610.

[17] SHU, G., HSU, Y., AND LEE, D. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *Formal Techniques for Networked and Distributed Systems - FORTE 2008* (2008), vol. 5048 of *LNCS*, Springer, pp. 299–304.

[18] SMEENK, W., JANSEN, D., AND VAANDRAGER, F. Applying automata learning to embedded control software. In submission., 2013.

[19] SZIKORA, J.-P., AND TEUWEN, P. Banques en ligne: à la découverte d'EMV-CAP. *MISC (Multi-System & Internet Security Cookbook) 56* (2011), 50–62.

A USB Commands and Responses

SHIELD	02 09 00 00 00 00 00 00
INSERT_CARD	01 03 01 02 00 00 00 00 00 02 65 6e 00 00 00 00
PIN	01 03 04 00 00 00 00 00
SIGNDATA	01 03 05 16 00 00 00 00 00 06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 04 00 00 00 00 00 00
DISPLAY_TEXT	01 03 05 46 00 00 00 00 00 06 01 44 42 65 74 61 00 06 6c 65 6e 20 20 20 00 06 20 20 20 20 20 20 00 06 20 31 20 74 72 61 00 06 6e 73 61 63 74 69 00 06 65 28 73 29 20 20 00 06 45 55 52 20 31 2e 00 06 30 30 30 2c 30 30 00 06 20 20 20 20 20 42 00 06 65 76 65 73 74 69 00 06 67 20 6d 65 74 20 00 04 4f 4b 20 20 74 20
GEN_CRYPTOGRAM	01 03 06 00 00 00 00 00

Table 1: USB commands sent to the e.dentifier2.

CRYPTOGRAM	01 03 03 1b 00 00 00 00 00 06 80 01 1d cd 29 7f 00 06 6f 1e 10 a5 00 03 00 06 04 00 00 00 00 00 00 06 00 00 00 00 00 00 00 03 ff 01 05 00 00 00
CARD_INSERTED	02 81 01 00 00 00 00 00
INSERT_OK	01 03 01 01 00 00 00 00 00 01 01 01 00 00 00 00
LONG_ERROR	01 03 08 01 00 00 00 00 00 01 25 01 00 00 00 00
SHORT_ERROR	01 03 07 00 00 00 00 00
OK	01 03 02 00 00 00 00 00

Table 2: USB responses of the e.dentifier2.