Security Testing

fuzzing

protocol fuzzing

model-based testing

automated reverse engineering

Erik Poll

Radboud University Nijmegen

Two ways to hunt for security vulnerabilities

Dynamic analysis

looking at the *behaviour* at *runtime*

- testing
- fuzzing
- (human) pen-testing
- ...

Static analysis

looking at the *code* at *compile time*

- source code scanners
- source code analysers
- (human) code review
- ...

for source code or byte code

Both can be (partly) automated, or done manually

Testing Ingredients

To test a **SUT (System Under Test)** we need two things

1. **test suite**, ie. collection of input data

2. **a test oracle**
   that decides if a test was passed ok or reveals an error
   i.e. some way to decide if the SUT behaves as we want

Both defining test suites and test oracles can be *a lot of work!*

In worst case, for test oracle: *for each individual test case, specify exactly what should happen*

A nice & simple test oracle: *just seeing if the SUT crashes*

Coverage Criteria

Measures of how good a test suite is

- statement coverage
- branch coverage

  Statement coverage does not imply branch coverage; eg for

  ```
  void f (int x, y) { if (x>0) {y++};
                               y--; }
  ```

  statement coverage needs 1 test case,
  branch coverage needs 2

- More complex coverage criteria exists, eg MCDC (Modified condition/decision coverage), commonly used in avionics

Possible perverse effect of coverage criteria

High coverage criteria may *dis*courage defensive programming

```
void m(File f){
  if <security_check_fails> {throw (SecurityException)}
   try { <the main part of the method> }
    catch (SomeException) { <take some measures>;
                            throw (SecurityException) }
  }
```

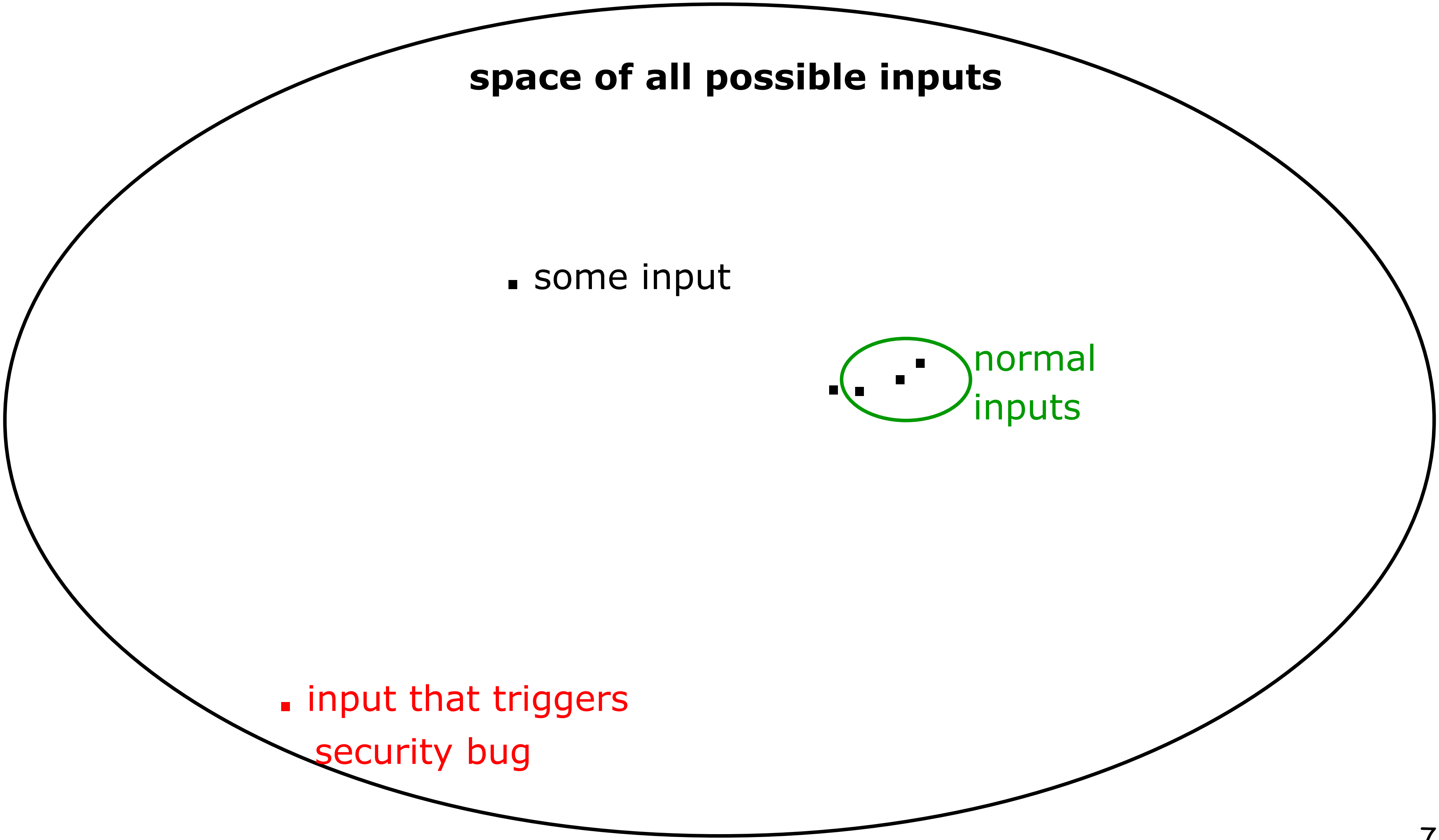
If the green defensive code is hard to trigger in tests programmers may be tempted (or forced) to remove it to improve coverage in testing...

Security testing is HARD

- Normal testing will look at right, wanted behaviour for sensible inputs ("the happy flow"), and some inputs on borderline conditions
- Security testing also involves looking for the wrong, unwanted behaviour for really silly inputs

- Similarly, normal use of a system will reveal
  functional problems (users will complain)
  but not
  security problems (hackers won't complain)

Security testing is HARD, in general



Fuzzing

Fuzzing

1. original form of “classic” fuzzing
   - trying put ridiculously long inputs
2. protocol/format fuzzing
   - trying out strange inputs, given some format/language
3. state-based fuzzing
   - trying out strange sequences of input

2 & 3 are essentially forms of model-based testing

Advanced forms of this become automated reverse engineering

Classic fuzzing

Fuzzing

Fuzzing

try really long inputs for string arguments to trigger segmentation faults and hence find buffer overflows

Benefit: can be automated, because test suite of long inputs can be automatically generated, and test oracle is trivial: just check if the program crashes

This original idea has been generalised to other settings.

The general idea of fuzzing: using *semi-random, automatically generated* test data that is likely to trigger security problems that can *automatically be detected*

Fuzzing in memory safe languages?

For memory safe languages, such as Java or C#,

classic fuzzing would not be able to spot buffer overflows?

Yes it can!

Fuzzing can still reveal bugs in a components of the Java or .NET platform that were written in C(++):

- Virtual Machine, the bytecode verifier, or libraries with native code

For example, fast graphics libraries often rely on native code

Fuzzing some format, language, ...

Fuzzing file formats

- Incorrectly parsing input formats or input languages is a common cause of security vulnerabilities

For example:

- email adresses
- X509 certificates
- audio, image & video formats: JPG, MPEG, MP3, MP4, ...
- HTML
- XML
- ....

Microsoft Security Bulletin MS04-028

Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution
Impact of Vulnerability: *Remote Code Execution*
Maximum Severity Rating: *Critical*
Recommendation : *Customers should apply the update immediately*

Root cause: a zero sized comment field, without content.

Even in "safe"programming languages…

CVE reference: CVE-2007-0243

Sun Java JRE GIF Image Processing Buffer Overflow Vulnerability

Critical: Highly critical

Impact: System access

Where: From remote

Description: A vulnerability has been reported in Sun Java Runtime Environment (JRE). The vulnerability is caused due to an error when processing GIF images and can be exploited to cause a heap-based buffer overflow via a specially crafted GIF image with an image width of 0.

Successful exploitation allows execution of arbitrary code.

Fuzzing web-applications?

- Could we fuzz a web application in the hope to find security flaws?
  - SQL injection
  - XSS
  - ...

  Effectively an automated pen tester

- What would be needed?
  - test inputs that trigger these security flaws
  - some way of detecting if a security flaw occurred
    - looking at website response, or log files

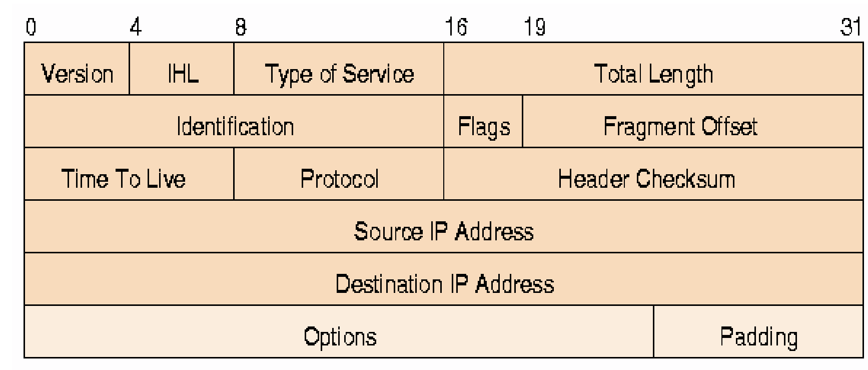
Fuzzing web-applications

- There are many tools to fuzz web-applications
  - Spike proxy, HP Webinspect, AppScan, acunetix, WebScarab, Wapiti, w3af, RFuzz, WSFuzzer, SPI Fuzzer Burp, Mutilidae, ...
- Some fuzzers crawl a website, generating traffic themselves, other fuzzers modify traffic generated by some other means.
- Can we expect false positives/negatives?
  - false negatives due to test cases not hitting the vulnerable cases
  - false positives & negatives due to incorrect test oracle, eg
  - for SQL injection: not recognizing some SQL database errors (false neg)
  - for XSS: signalling a correctly quoted echoed response as XSS (false pos)

Protocol Fuzzing

Protocol fuzzing based on known *protocol format*

ie format of packets or messages



Typical things to try in protocol fuzzing:

- trying out many/all possible values for specific fields

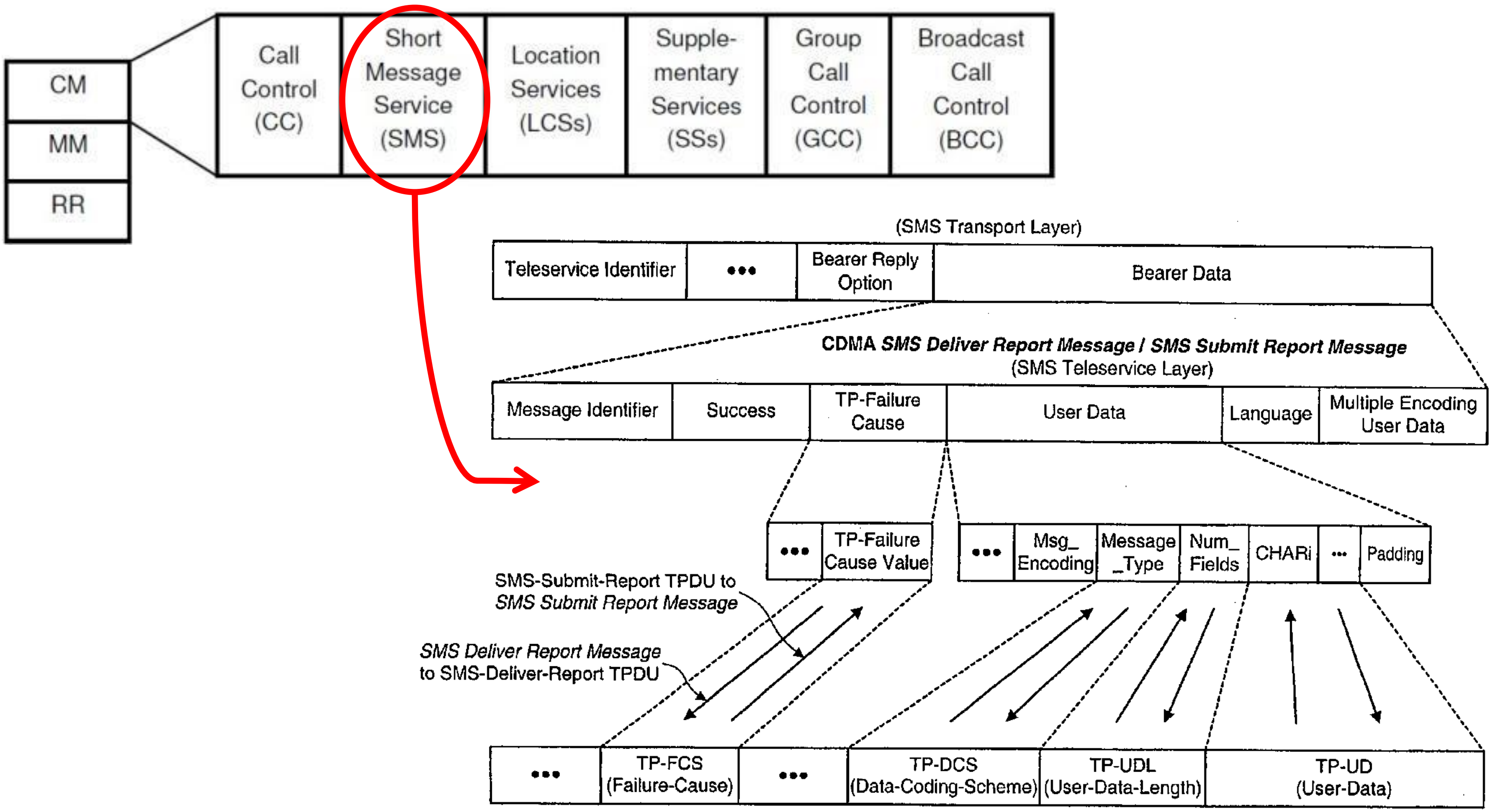
  esp undefined values, or values “Reserved for Future Use” (RFU)

- giving incorrect lengths, length that are zero, or payloads that are too short/long

Tools for protocol fuzzing exist, eg SNOOZE, Peach, Sulley

Example : GSM protocol fuzzing

GSM is a very rich & complicated protocol



SMS message fields

Field	size
Message Type Indicator	2 bit
Reject Duplicates	1 bit
Validity Period Format	2 bit
User Data Header Indicator	1 bit
Reply Path	1 bit
Message Reference	integer
Destination Address	2-12 byte
Protocol Identifier	1 byte
Data Coding Scheme (CDS)	1 byte
Validity Period	1 byte/7 bytes
User Data Length (UDL)	integer
User Data	depends on CDS and UDL

Example: GSM protocol fuzzing

Lots of stuff to fuzz!

We can use a USRP
(universal software radio peripheral)

with open source cell tower software

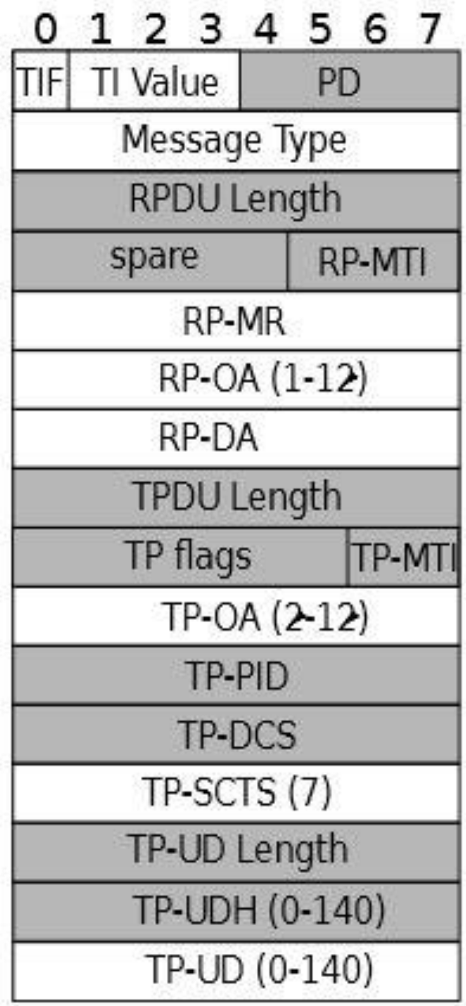
(OpenBTS)

to fuzz lots of mobile phones

[Mulliner et al, SMS of Death]

[F vb Broek, B. Hond, A. Cedillo Torres, Security Testing of GSM Implementations]

GSM fuzzing – fields fuzzed



(b) Overview of the fields we fuzzed in the SMS-DELIVER message.

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and phones

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and phones

eg possibility to send faxes (!?)

you have a fax!

Only way to get rid if this icon: reboot the phone

Example: GSM protocol fuzzing

Malformed SMS text messages showing raw memory contents, rather than content of the text message

(a) Showing garbage

(b) Showing the name of a wallpaper and two games

GSM fuzzing results : SMS

- all 16 phones accepted obscure SMS variants that could be read using the phone's UI, sometimes with unremovable icons
- 5 out of 16 phones would accept certain SMS messages without notification
- 7 out of 16 phones could be forced to reboot
- Nokia 2006 could be made to show random part of memory
- iPhone 4 and HTC Legend could be DoS-ed with an SMS message: they would not notify user that this message was received and stopped receiving further messages

GSM fuzzing results : CBS

CBS (Cell Broadcast Service) is meant for emergency warnings, which a mobile phone can subscribe to.

- No crashes
- Most phones have trouble to show even correct CBS messages
- Galaxy Note displayed message which should be ignored
- All phones except Blackberry would accept some CBS messages which were *not* subscribed to
- All phones would ignore some messages that they *were* subscribed to

GSM fuzzing of SMS and CBS: results

Legend: I: unremoveable icons, D: DoS message, M: memory bug, N: no notification, R: Reboot S: message handling in violation of specification.

Brand	Type	Firmware/OS	SMS fuzz	Result	CBS fuzz	Result
Apple	iPhone 4	iOS 4.3.3	yes	I,D	no	–
Blackberry	9700	BB OS 5.0.0.743	yes	I	yes	S
HTC	Legend	Android 2.2	yes	I,D	no	–
Nokia	1100	6.64	yes	I	no	–
Nokia	1600	RH-64 v6.90	no	–	yes	S
Nokia	2600	4.42	yes	I,M,R	no	–
Nokia	3310	5.57	yes	I	yes	S
Nokia	3410	5.06	yes	I	no	–
Nokia	6610	4.18	yes	I,N,R	no	–
Nokia	6610	4.74	yes	I,N,R	no	–
Nokia	7650	4.36	yes	I,R	no	–
Nokia	E70-1	3.0633.09.04	yes	I	no	–
Nokia	E71-1	110.07.127	yes	I	no	–
Samsung	SGH-A800	A80XAVK3	yes	I,N,R	no	–
Samsung	SGH-D500	D500CEED2	yes	I,M,R	no	–
Samsung	Galaxy S	Android 2.2.1	yes	I	no	–
Samsung	Galaxy Note	Android 4.1.2	no	–	yes	S
Sony Ericsson	T630	R7A011	yes	I,N	no	–

Example: GSM protocol fuzzing

- Lots of success to DoS phones: phones crash, disconnect from the network, or stop accepting calls
  - eg requiring reboot or battery removal to restart and accept calls
  - after reboot, a real network would re-deliver the SMS message, if no acknowledgement was sent before crashing, re-crashing phone!

  But: not all these SMS messages could be sent over real network
- Strangely, there is no correlation between problems and phone brands & firmware versions
  - ie. some similar phones have very different problems
- *The scary part: what would happen if we fuzz base stations?*

State-based fuzzing
of
sequences of inputs

State-based Protocol Fuzzing

Instead of fuzzing the content of individual messages,

we can also fuzz the *order* of messages

using *protocol state-machine* to

1. reach an interesting state in the protocol and then fuzz content of messages there;
2. fuzz the order of messages to discover effect of strange sequences

State-based Protocol Fuzzing

- Most protocols have different types of messages,
  which should come in a particular order,
  the so-called happy flow
- We can fuzz a protocol by trying out the different types of messages in all possible orders
- This can reveal loop-holes in the application logic

Essentially this is a from of model-based testing, where we automatically test if an implementation conforms to a model

[Tools for this: Peach, jTor]

Model based testing

General framework for automating testing

1. make a *formal model M* of (some aspect of) the SUT
2. fire *random inputs* to M and the SUT
3. look for differences in the response

Such a difference means an error in the SUT, *or* the model...

Once we have the model, the testing can be largely automated

Example: analysis of SSH implementations

Essence of SSH transport layer

1. C -> S: NC
2. S -> C: NS
3. C -> S: exp(g,X)
4. S -> C: k_S.exp(g,Y).{H}_inv(k_S)
   with K=exp(exp(g,X),Y),
   H=hash(NC.NS.k_S.exp(g,X).exp(g,Y).K)
5. C -> S: {XXX}_KCS
   with SID=H, KCS=hash(K.H.c.SID)
6. S -> C: {YYY}_KSC
   with SID=H, KSC=hash(K.H.d.SID)

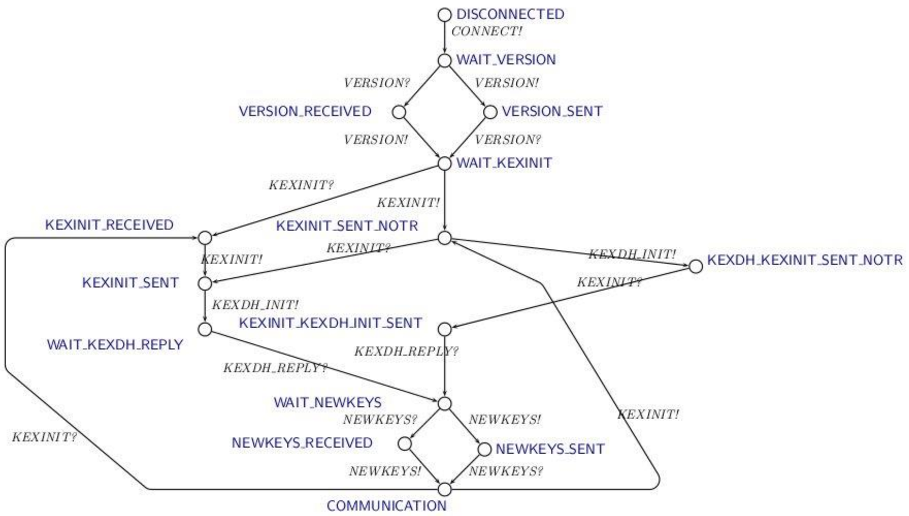
Goal: establish a session key to encrypt traffic between client and server, as in SSL/TLS and https

Example: analysis of SSH implementations

Essence of SSH transport layer

1. C -> S: NC
2. S -> C: NS
3. C -> S: exp(g,X)
4. S -> C: k_S.exp(g,Y).{H}_inv(k_S)
   with K=exp(exp(g,X),Y),
   H=hash(NC.NS.k_S.exp(g,X).exp(g,Y).K)
5. C -> S: {XXX}_KCS
   with SID=H, KCS=hash(K.H.c.SID)
6. S -> C: {YYY}_KSC
   with SID=H, KSC=hash(K.H.d.SID)

Real SSH transport layer

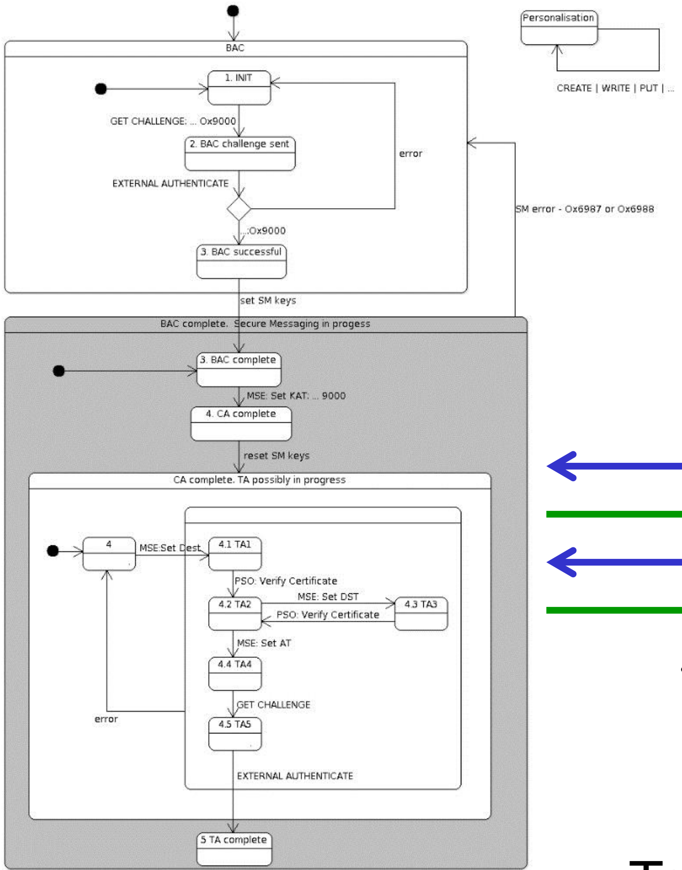


excluding all the error transitions back to the initial state

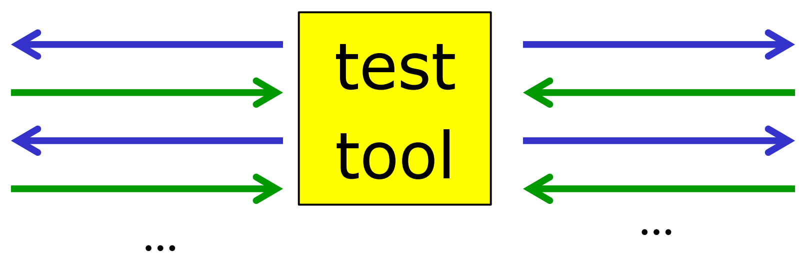
Example: analysis of SSH implementations

- One open source implementation of MIDPSSH we analysed forgot to implement the state machine
  - a Man-in-the-Middle attacker could request a username & password *before* a session key was established, so this would go unencrypted over the network

Example: model based testing of e-passport



model



SUT

Test tool sends the same *random* sequence of commands to the model and the SUT, and checks if the responses match

(Automated)
Reverse Engineering

In the other direction:

Instead of using protocol knowledge when testing

in protocol fuzzing or model-based fuzzing

we can also use testing to gain knowledge about a protocol

or a particular *implementation* of a protocol

This is useful

1. to analyse your own code and hunt for bugs, or
2. to reverse-engineer someone else's unknown protocol, eg a botnet to fingerprint or to analyse (and attack) it

What to reverse engineer?

Different aspects that can be learned:

- timing/traffic analysis
- protocol formats
    - ie format of protocol packets

      [eg using Discoverer, Dispatcher, Tupni,....]
- protocol state-machine

  [eg using LearnLib]
- both protocol format & state-machine

  [eg using Prospex]

How to reverse engineer?

- ***passive*** vs ***active*** learning

  ie passive observing or active testing

  - active learning involves a form of fuzzing
  - these approaches learns different things:
    passive learning produces statistics on normal use,
    active learning will more aggresively try our strange things

- **black box** vs **white box**

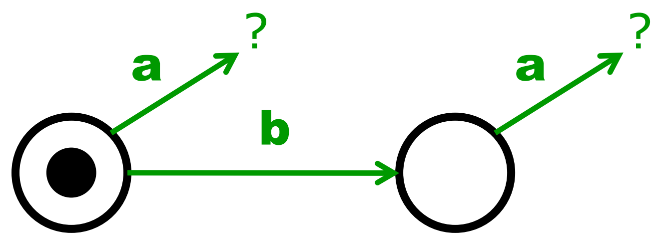
  ie only observing in/output or also looking inside running code

Active learning with Angluin's L* algorithm

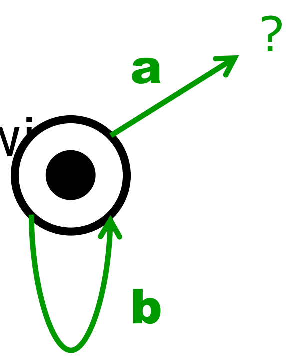
Basic idea: compare a deterministic system's response to

- **a**
- **b ; a**

If response is different, then



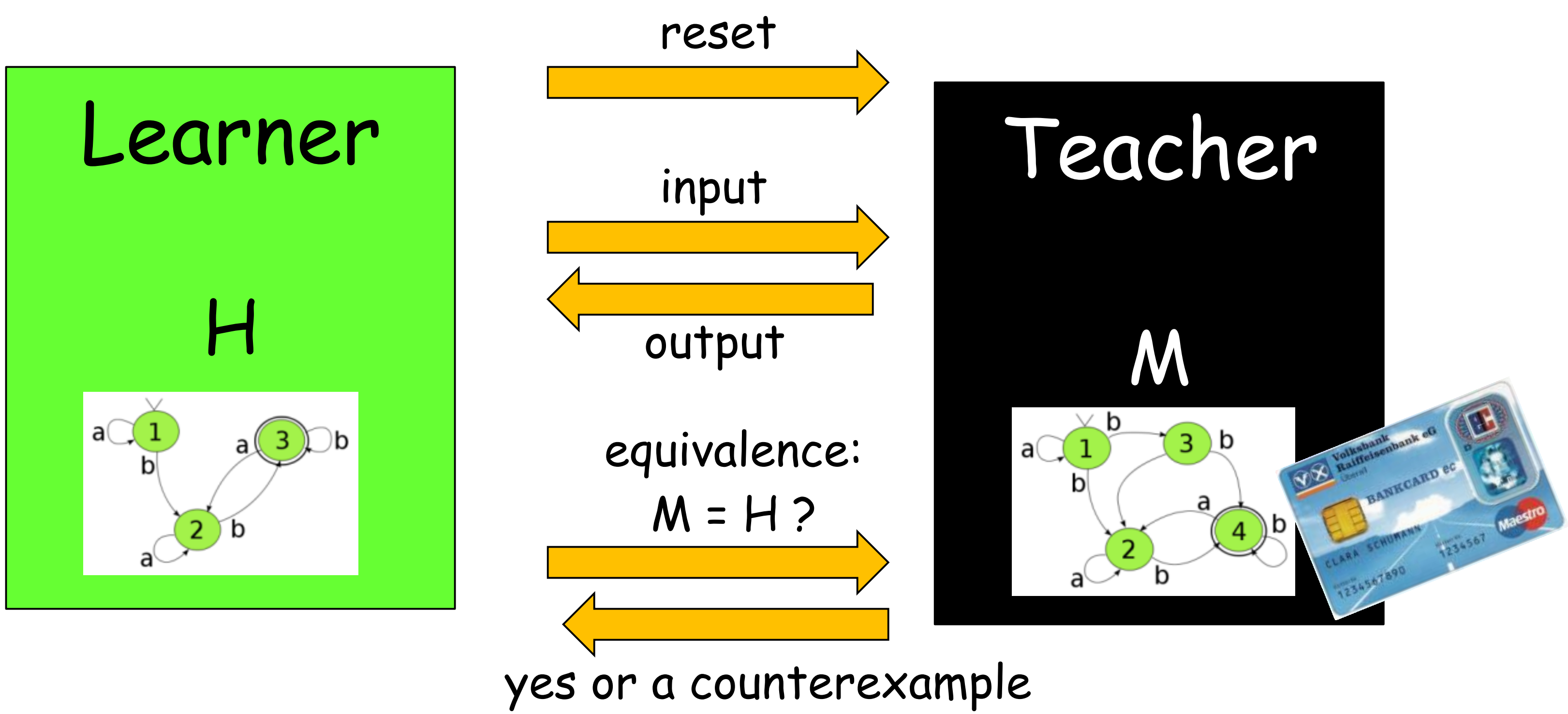
otherwi



?

Active learning with L*

Implemented in LearnLib library;
The learner builds hypothesis H of what the real system M is



Equivalence can only be approximated in a black box setting;
by doing model-based testing to see if a difference can be detected

Test harness for EMV

Our test harness implements standard EMV instructions, eg

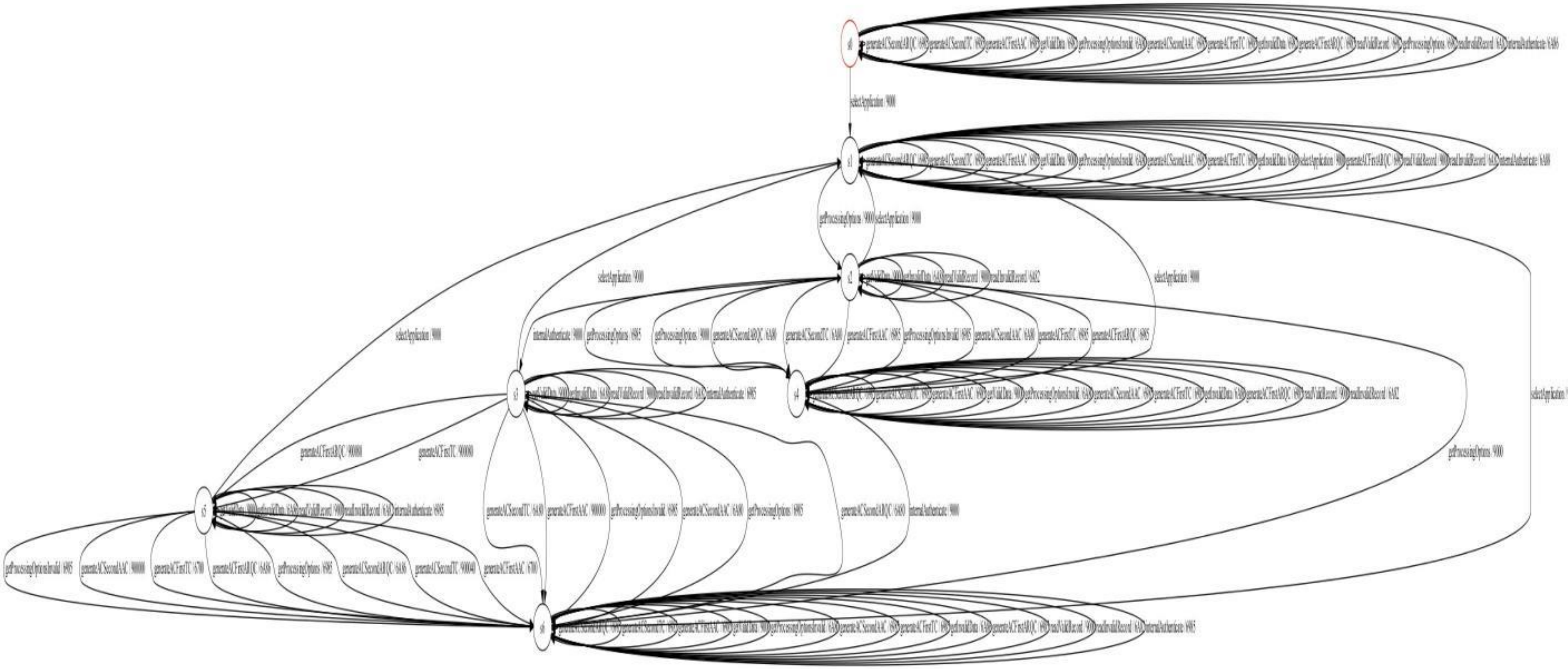
- SELECT (to select application)
- INTERNAL AUTHENTICATE (for a challenge-response)
- VERIFY (to check the PIN code)
- READ RECORD
- GENERATE AC (to generate application cryptogram)

LearnLib then tries to learn all possible combinations

- Most commands with fixed parameters, but some with different options

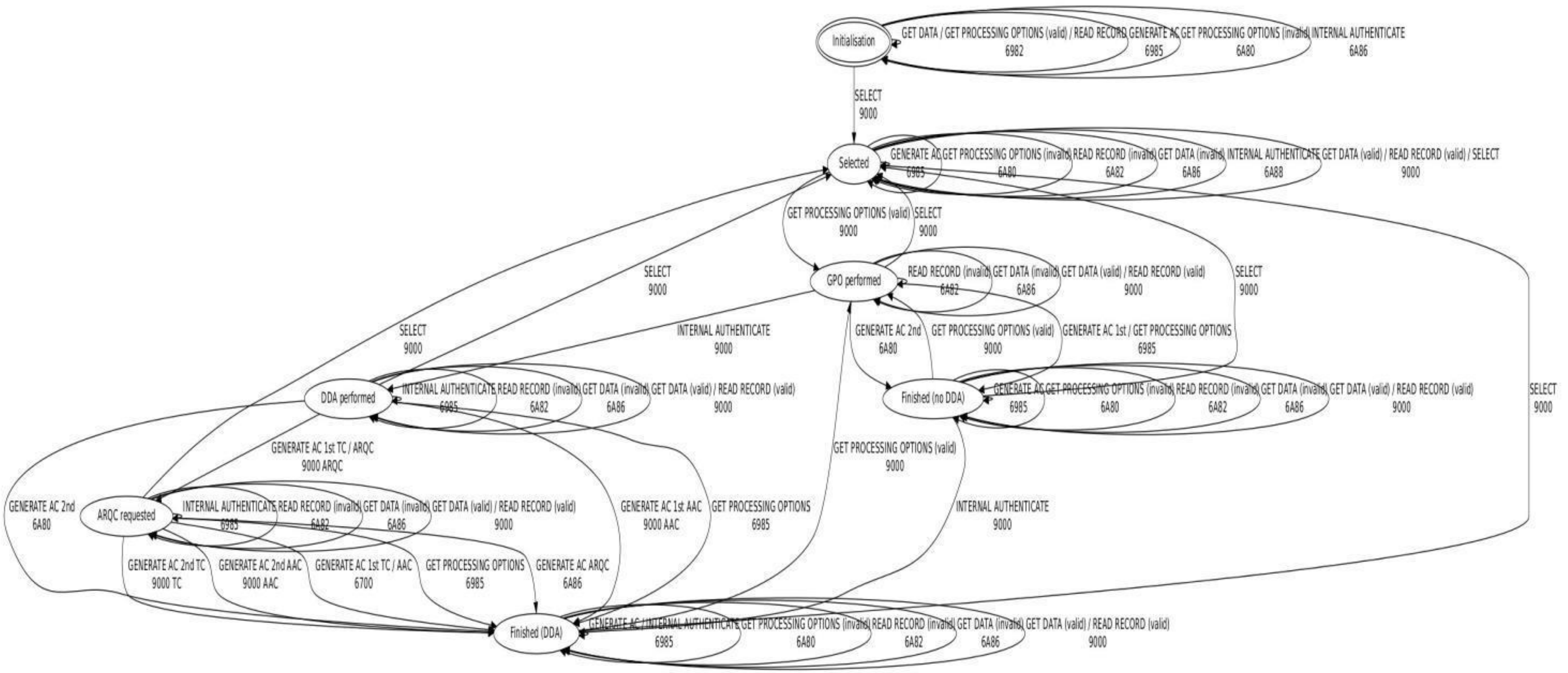

woj@impreza:~/javacard - Shell No. 4 - Konsole
Session Edit View Bookmarks Settings Help
woj@impreza ~/javacard $ javac -classpath ~/jpcsc/jpcsc.jar:
woj@impreza ~/javacard $ chipknip.sh
Using reader OMNIKEY CardMan 5121 00 00
Command APDU: 5: E1B4000105
Response APDU: 7: 0000E009789000
Chipknip balance: 2.24 Euro
woj@impreza ~/javacard $
AUTO
Wereldpas
Maestro

Maestro application on Volksbank bank card
raw result



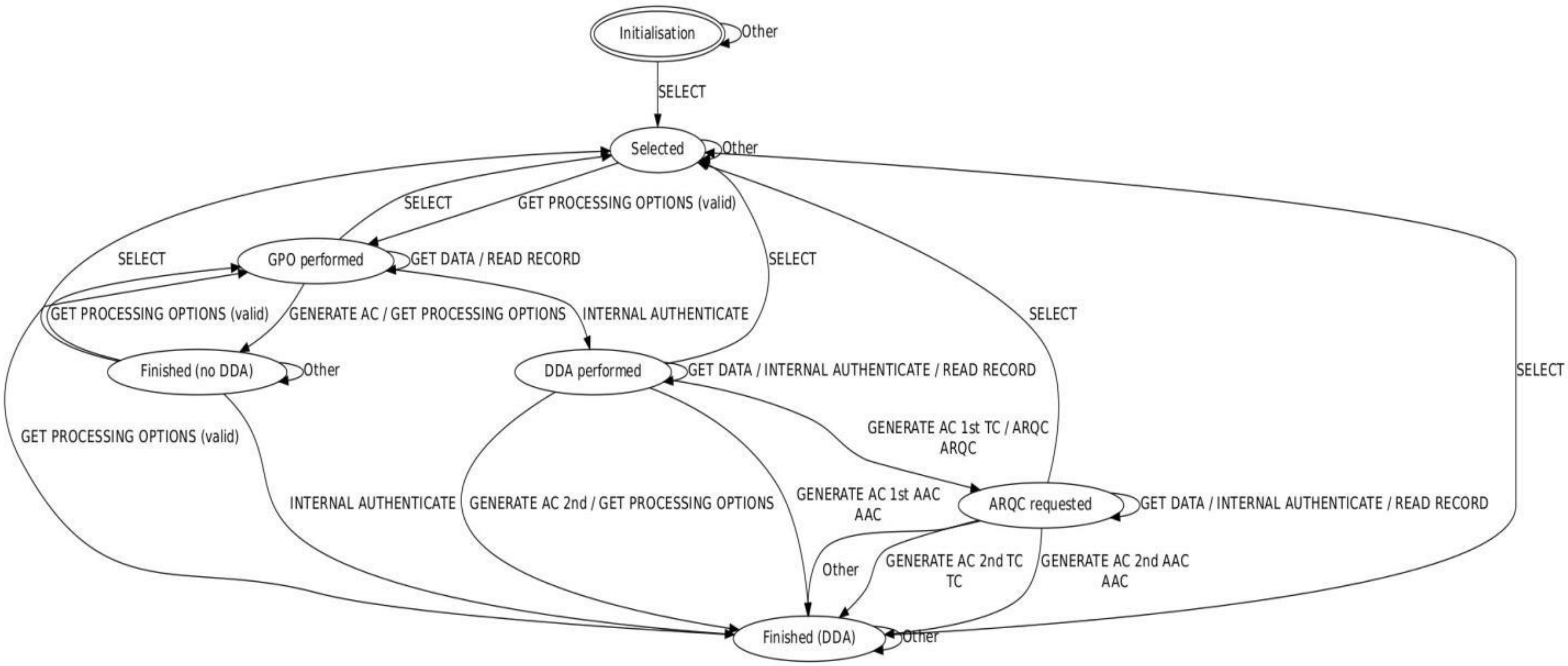
Maestro application on Volksbank bank card

merging arrows with identical outputs



Maestro application on Volksbank card

merging all arrows with same start & end state



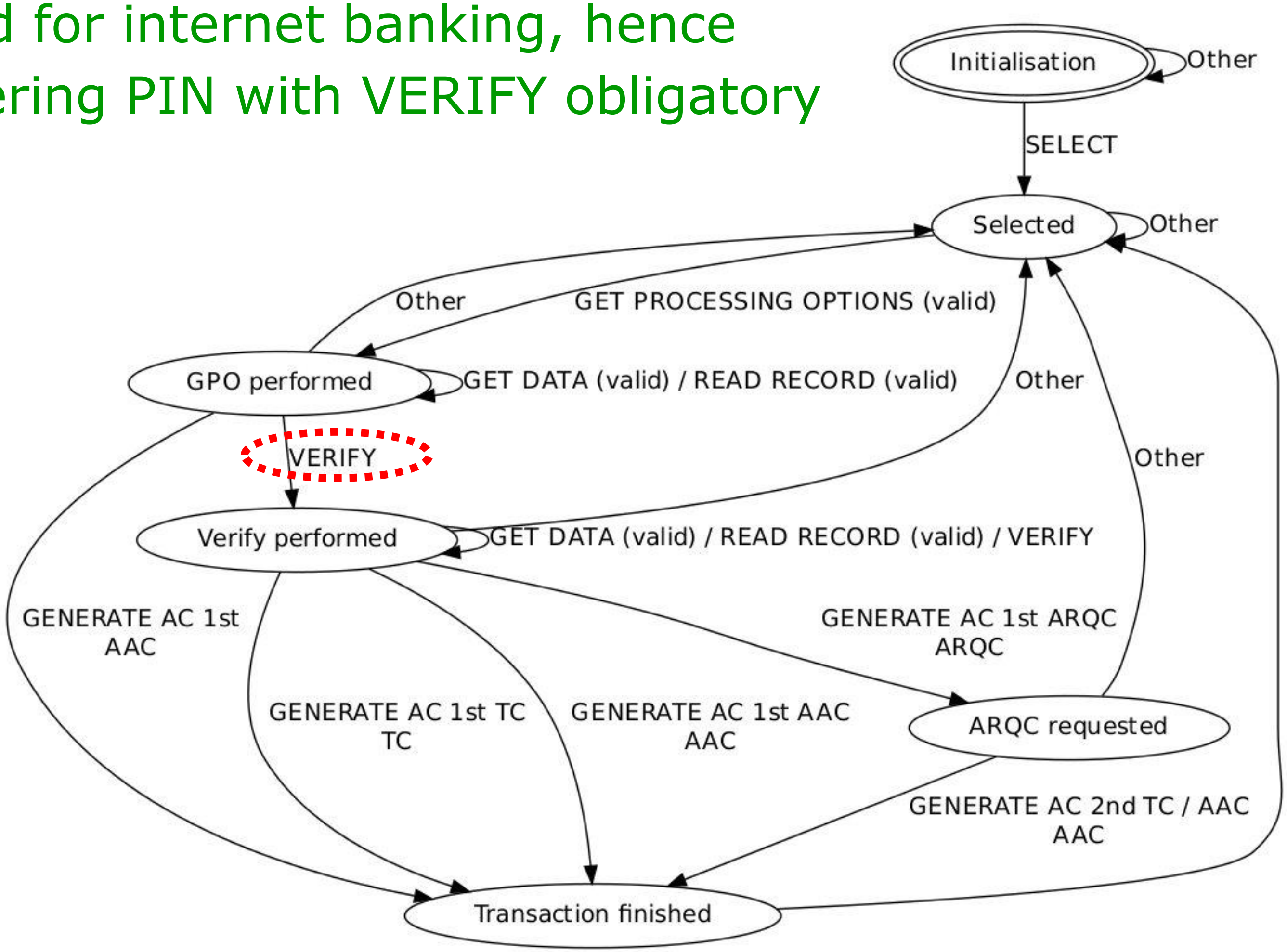
Formal models of banking cards for free!

- Experiments with Dutch, German and Swedish bank and credit cards
- Learning takes between 9 and 26 minutes
- Editing by hand to merge arrows and name states
- Limitations
  - We do not try to learn response to incorrect PIN as cards would block...
  - We cannot learn about one protocol step which requires knowledge of card's secret 3DES key
  - We would also like to learns some integer parameter used in protocol
- No security problems found, but interesting insight in implementations

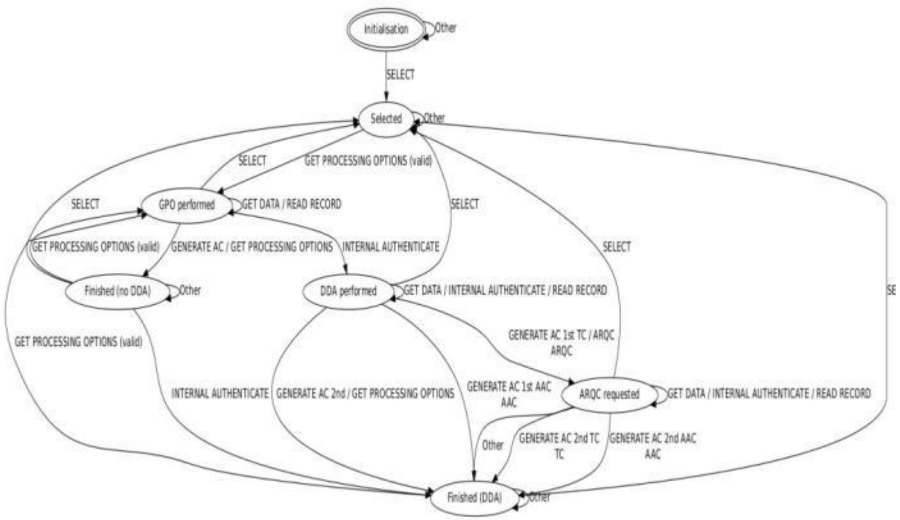
[F. Aarts et al, Formal models of bank cards for free, SECTEST 2013]

SecureCode application on Rabobank card

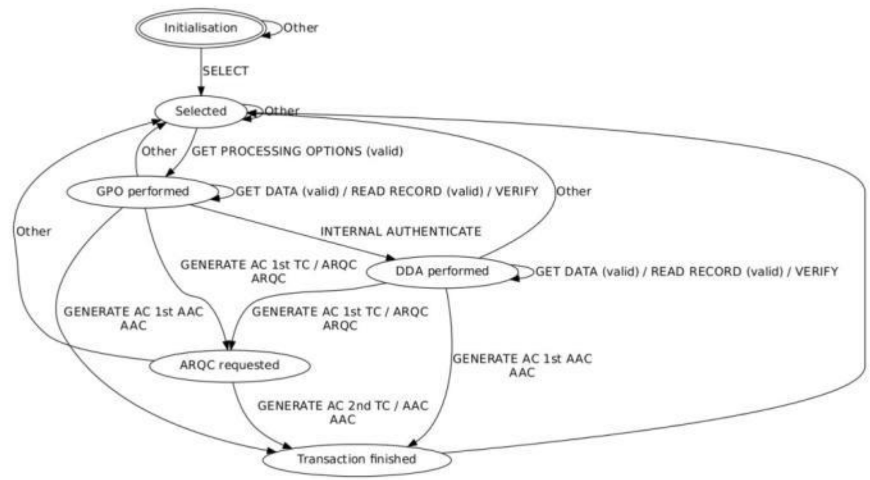
used for internet banking, hence
entering PIN with VERIFY obligatory



understanding & comparing implementations



Volksbank Maestro
implementation



Rabobank Maestro
implementation

Are both implementations correct & secure? And compatible?

Presumably they both passed a Maestro-approved compliance test suite...

Using such protocol state diagrams

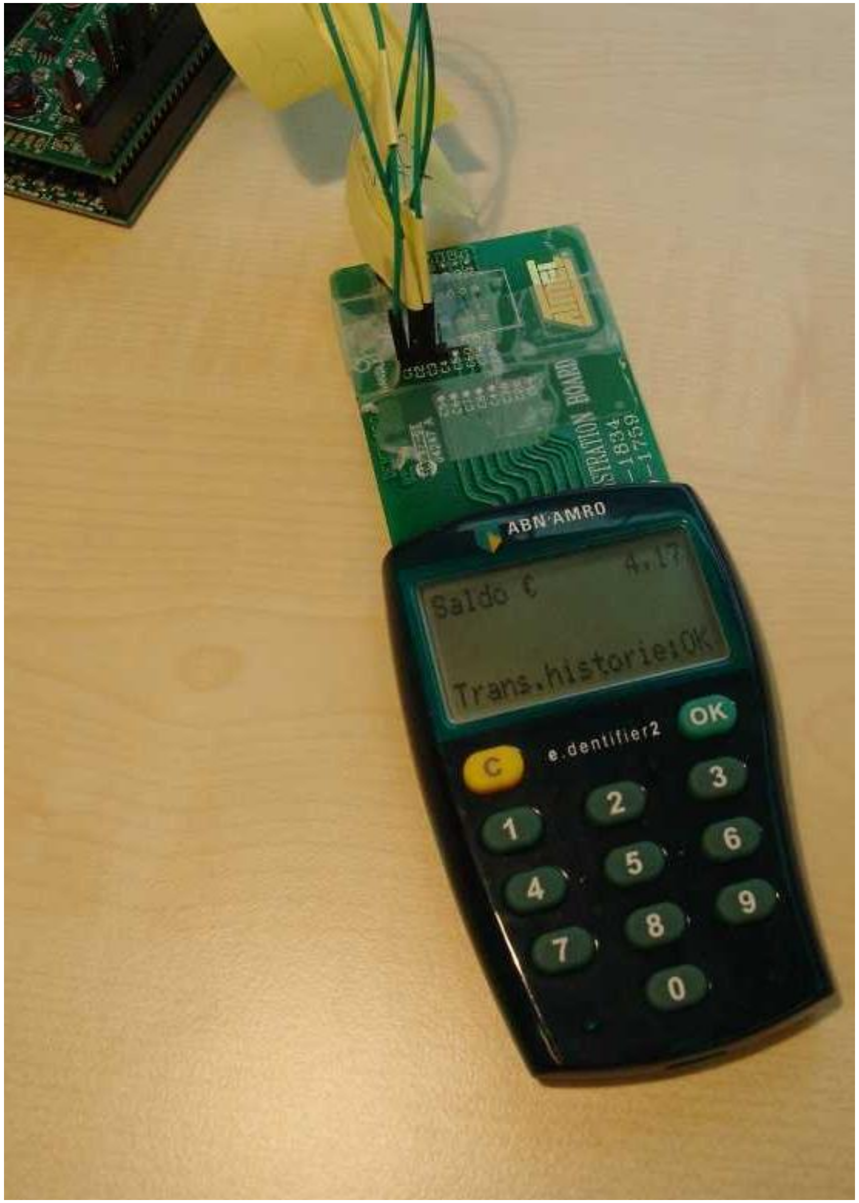
- Analysing the models by hand, or with model checker, for flaws
  - to see if *all paths* are correct & secure
- Fuzzing or model-based testing
  - using the diagram as basis for “deeper” fuzz testing
    - eg fuzzing also parameters of commands
- Program verification
  - *proving* that there is no functionality beyond that in the diagram, which using testing you can never establish
- Using it when doing a manual code review

Reverse engineering the USB-connected e.dentifier

Can we fuzz

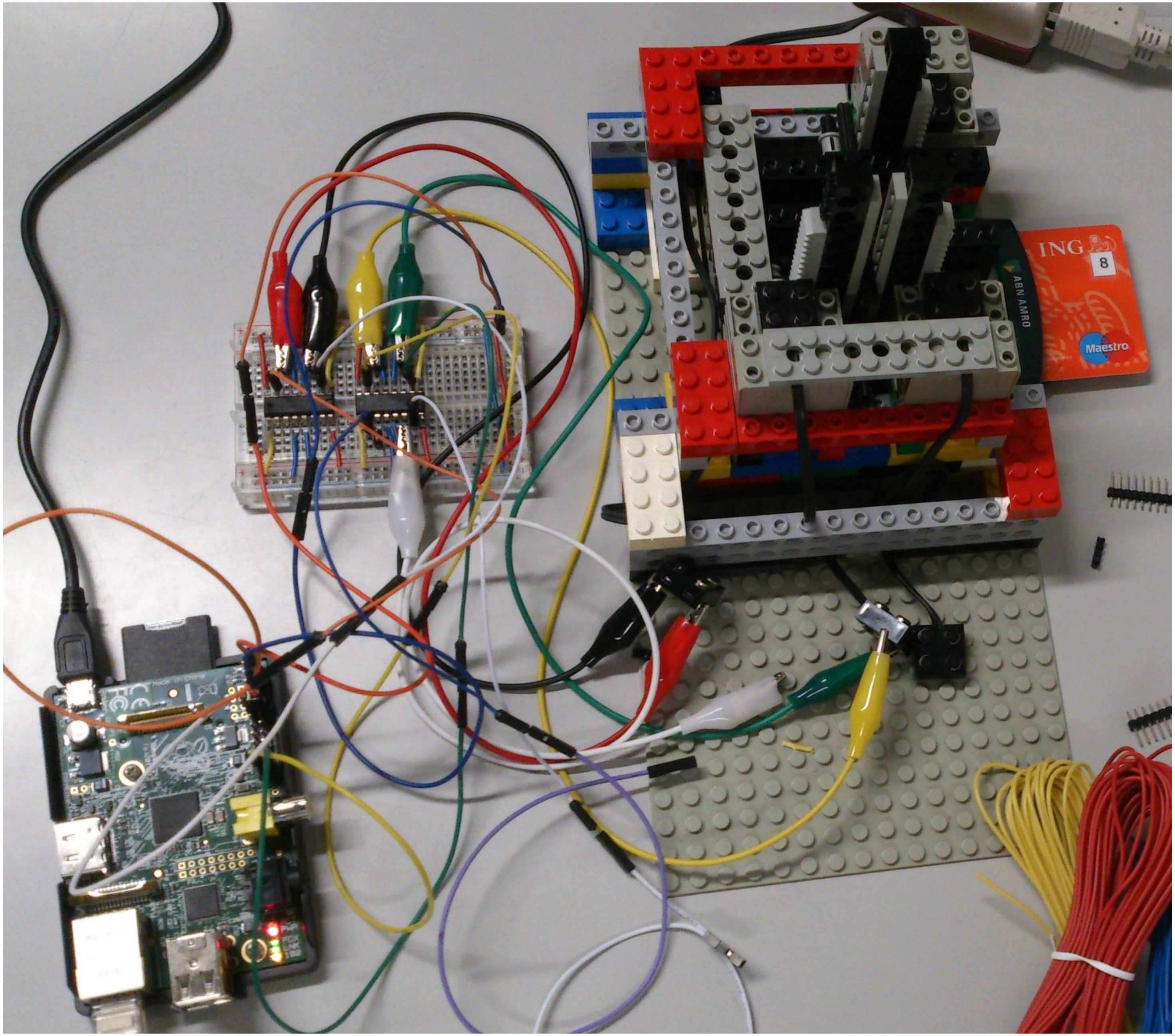
- USB commands
- user actions via keyboard

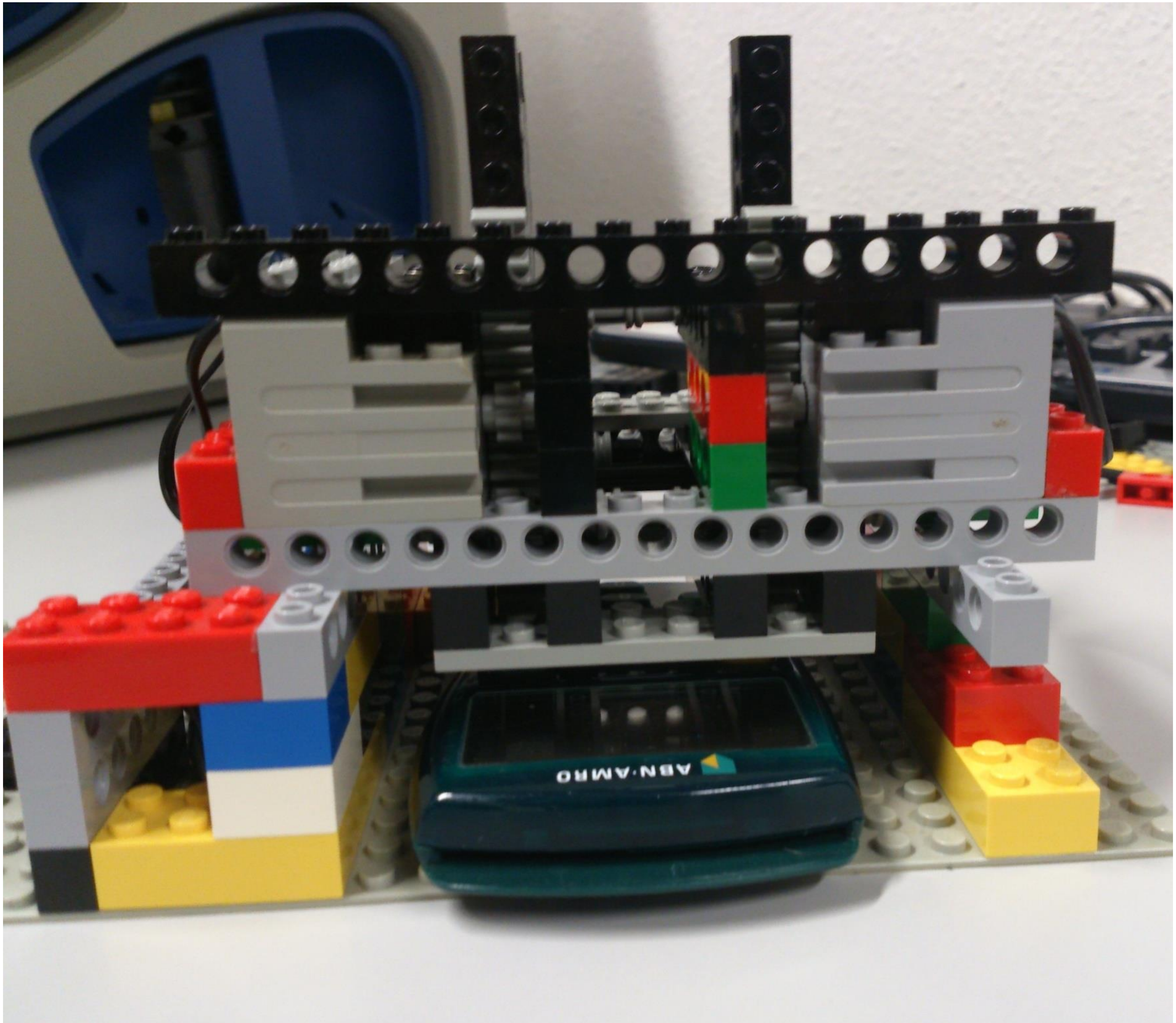
to find bug in ABN-AMRO e.dentifier2 using automated learning?



[Arjan Blom et al, Designed to Fail: a USB-connected reader for online banking, NORDSEC 2012]

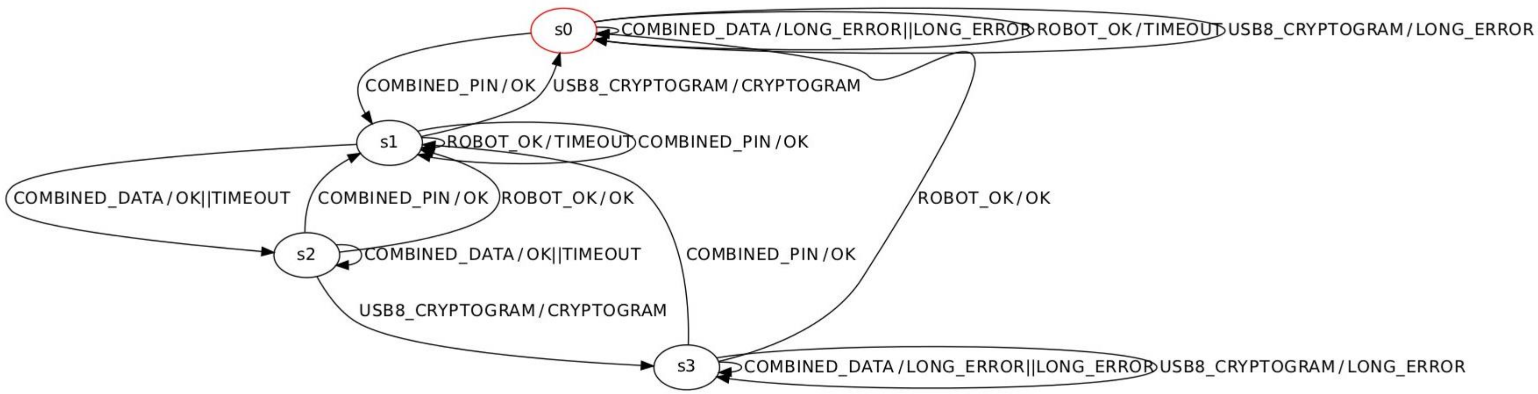
Operating the keyboard using of LEGO


ING
8
ABN AMRO
Maestro

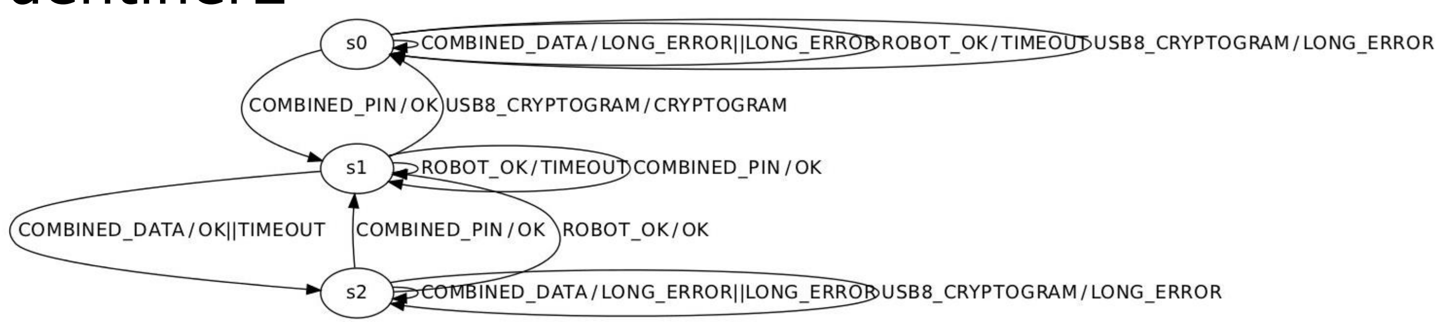

ABN·AMRO

The LEGO hacker let loose on

old e.dentifier2



new e.dentifier2



Case study: analysing SSL/TLS

Work in Progress

Early 2014: TSL bug in iOS and OSX

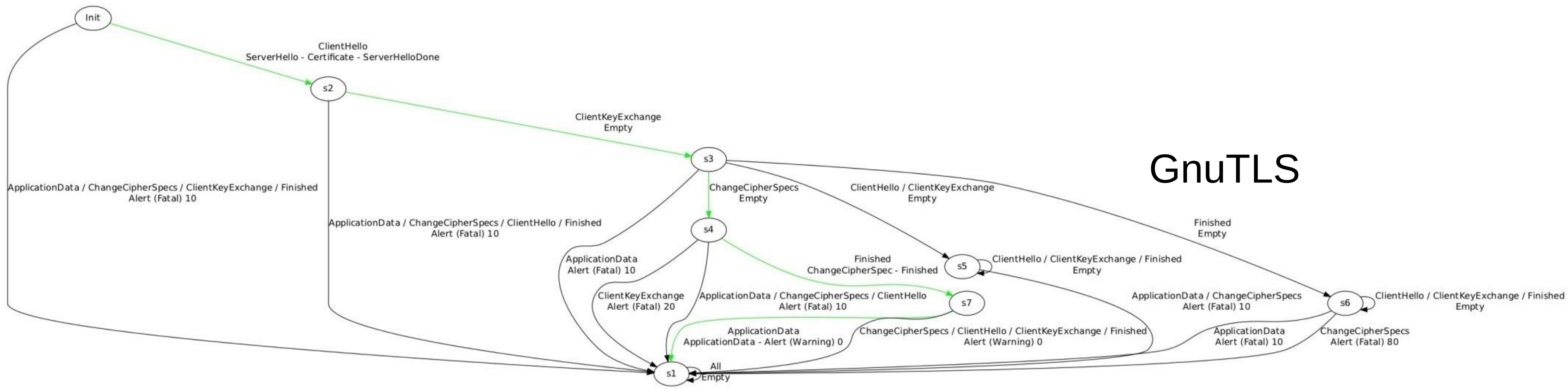
```c
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer
                                 uint8_t *signature, UInt16 signatureLen
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

Analysing TLS implementations

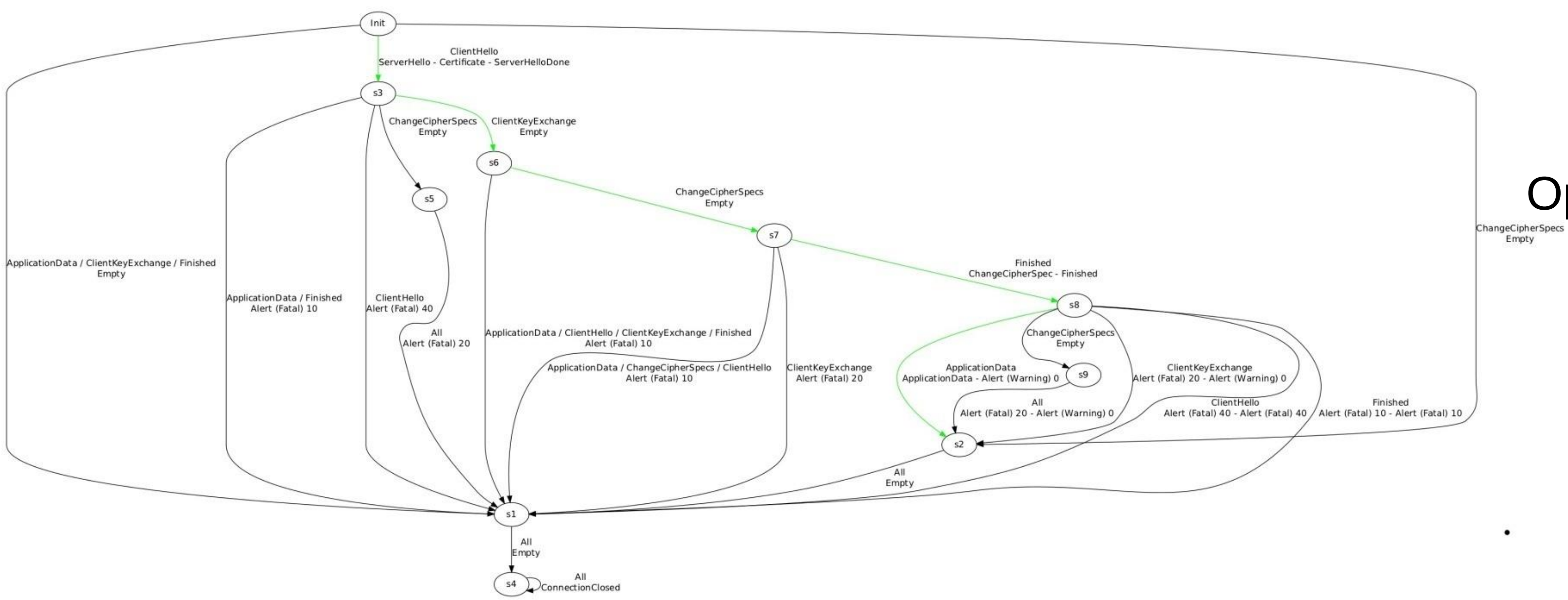
- Can we use state machine learning to extract the state machine from TLS/SSL implementations?
- Could be find bugs that way?

- Work in progress: Joeri de Ruiter analysed 9 TLS implementations, and found
  - state machines of all implementations are different!
  - new security flaws in 3 of them, plus another bug that had already been reported.

Open SSL vs GnuTLS



GnuTLS



OpenSSL

All TLS implementations are different!

Conclusions

- Various forms of **fuzzing** are great techniques to spot some security flaws
- More advanced forms of (protocol) fuzzing and automated reverse engineering (or learning) are closely related
- **State machines** are a great specification formalism
  - easy to draw on white boards, typically omitted in official specs

  and you can extract them for free from implementations
  - using standard, off-the-shelf, tools like LearnLib

  Useful for security analysis of protocol implementations
  - for reverse engineering, fuzz testing, code reviews, or formal program verification

Different forms of fuzzing for security testing

1. original form of fuzzing
   - trying put ridiculously long inputs to find buffer overflows
2. protocol/format fuzzing
   - trying out strange inputs, given some format/language to find flaws in program logic
3. state-based fuzzing
   - trying out strange sequences of input to find flaws in program logic

2 & 3 are essentially forms of model-based testing

Advanced forms of this become automated reverse engineering

specifications

implementing

```
                              Untitled
import java.util.*;
import java.text.*;

//Rod Bernardson
//Date: 02/22/2008
//Chapter 18 Programming Challenge 6
//DealerCards Class Demo

public class DealerCardsDemo
{
        /**
         * @param args
         */
        public static void main(String[] args)
        {
                // Determine who's turn to play it is
                // Create the
                Dealer deal = new Dealer();

                CardPlayer player = new CardPlayer(deal);
                ComputerPlayer cplayer = new ComputerPlayer(deal);

                deal.shuffleCards();
                deal.StartPlayingGame(cplayer);
                deal.StartPlayingGame(player);

                player.showCard();
                System.out.println("Player Points..1: " +
player.getTotalCardPoints());

                player.makeDecision();
                player.showCard();
                System.out.println(player.getTotalCardPoints());
                System.out.println("Player Points..1: " +
player.getTotalCardPoints());
                System.out.println("Computer Points.: " +
cplayer.getTotalCardPoints());

                if (cplayer.getTotalCardPoints() > player.getTotalCardPoints() &&
(cplayer.getTotalCardPoints() <= 21))
                        {
                                System.out.println("Computer wins the
game! \n\n");
                                System.out.println("\n");
                        }
                else
                        if (player.getTotalCardPoints() >
cplayer.getTotalCardPoints() && (player.getTotalCardPoints() <= 21))
                {
                        System.out.println("Player wins the game! \n\n");
                        System.out.println("\n");
                }
                        else
                                if (player.getTotalCardPoints() ==
cplayer.getTotalCardPoints() && (player.getTotalCardPoints() <= 21))
                                {
                                        System.out.println("Game is a tie! \n\n");
                                }

                        else
                                if (player.getTotalCardPoints() > 21)
                                {
                                System.out.println("Game Over - Computer wins and
                                    Page 1
```

code

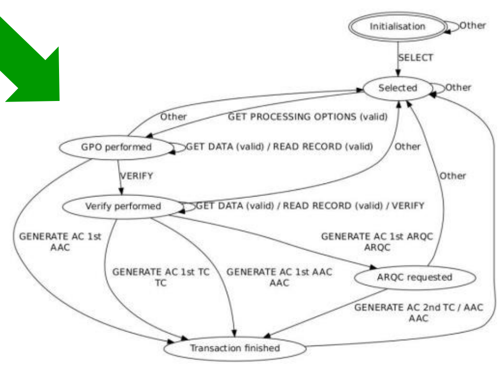
specifications

implementing

code

model-based testing

making model by hand



model

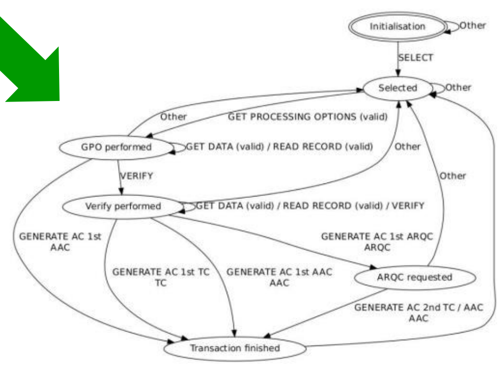
specifications

implementing

code

model-based testing

making model by hand



model

or automated learning