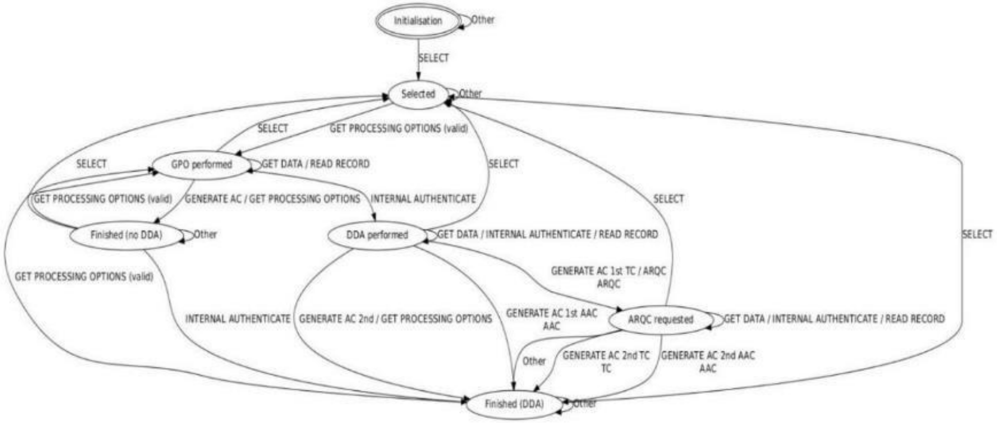


Security
of
Protocols & other Stateful Systems

using
automata learning
aka active state machine learning

To read: [Protocol state machines and session languages, LangSec'15]

Stateless vs stateful systems

- State*less* system: giving the same input (again) always results in the *same* response
  - Eg. opening a.pdf, b.pdf, c.pdf in a PDF viewer
  - In other words, the system has no memory/no history
- State*ful* system: giving the same input again may result in a *different* response
  - Eg. withdrawing 100 euros from an ATM
  - Processing the input results in a state change of the system

Do the fuzzers you tried work best for stateless or stateful systems?

Stateless

Which systems are harder to test (or fuzz): stateless or stateful systems?

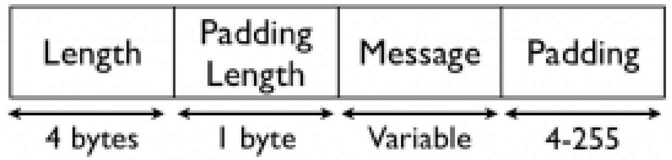
Stateful, because we can not just try different inputs,
but also different sequences of inputs

Protocols

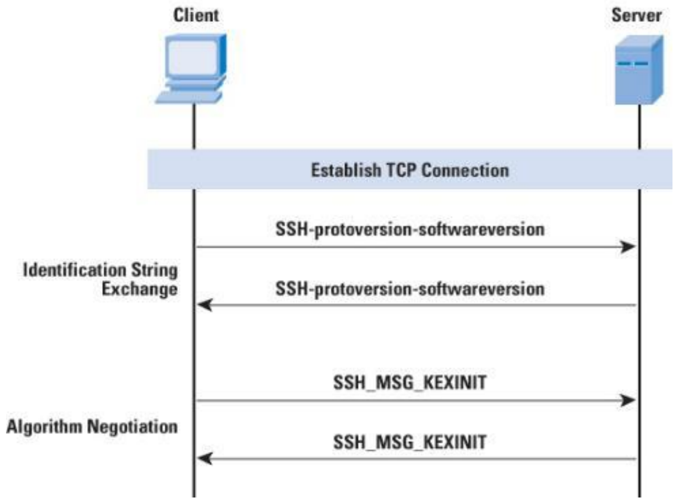
Many procotols are stateful and involve two levels of languages

1) a language of input messages (or packets)

2) a notion of session, or *sequence* of messages (aka traces)



Length	Padding Length	Message	Padding
4 bytes	1 byte	Variable	4-255



Client
Server
Establish TCP Connection
Identification String Exchange
SSH-protoversion-softwareversion
SSH-protoversion-softwareversion
Algorithm Negotiation
SSH_MSG_KEXINIT
SSH_MSG_KEXINIT

Bugs can arise on both levels!

Bugs on level 2 sometimes called flaws in program logic

How can we develop code for the two levels in a systematic way?

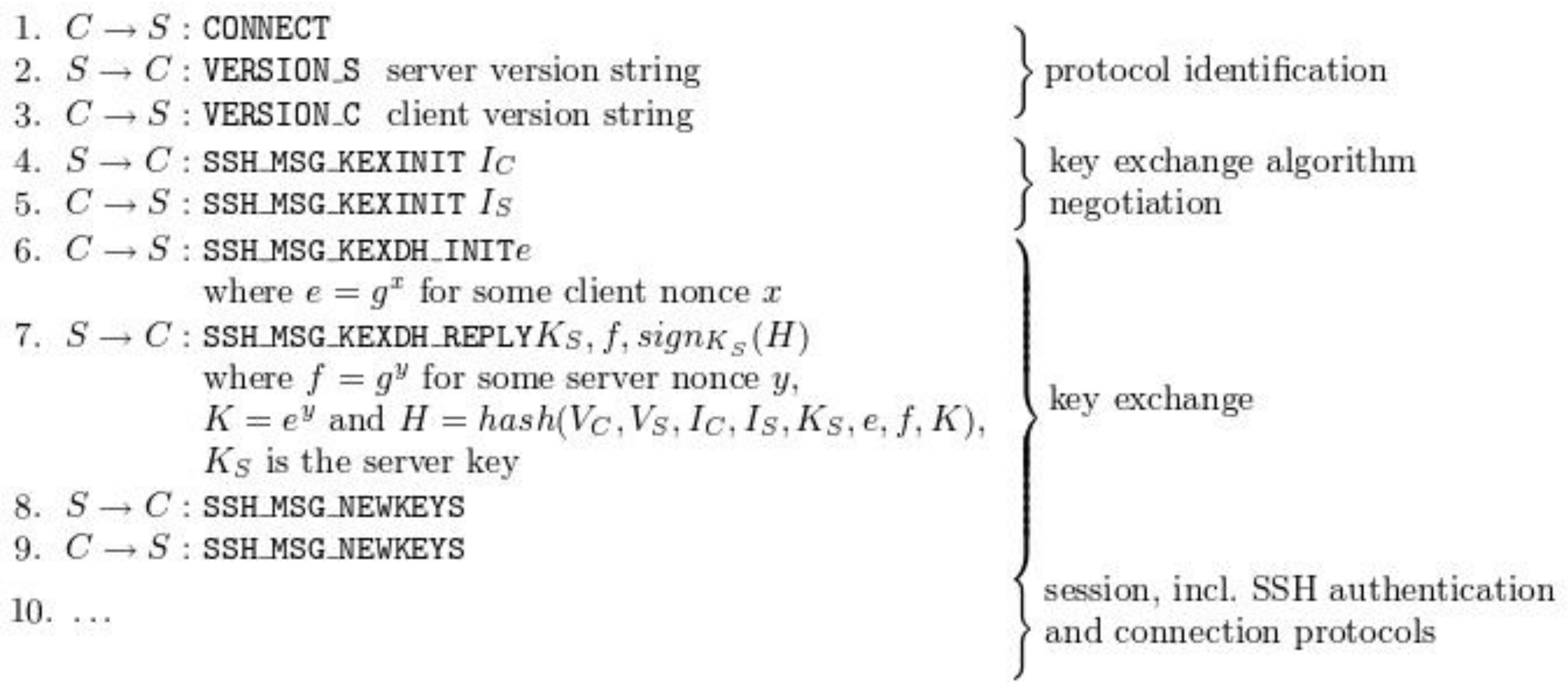
How can we test implementations of these two levels?

For level 1 we can use fuzzing, as discussed earlier

For level 2 we can do something different, as we discuss now

Specification with Message Sequence Charts (MSCs)

Eg for SSH



1. $C \rightarrow S$: CONNECT
2. $S \rightarrow C$: VERSION_S server version string
3. $C \rightarrow S$: VERSION_C client version string

protocol identification

4. $S \rightarrow C$: SSH_MSG_KEXINIT I_C
5. $C \rightarrow S$: SSH_MSG_KEXINIT I_S

key exchange algorithm negotiation

6. $C \rightarrow S$: SSH_MSG_KEXDH_INITe
   where $e = g^x$ for some client nonce x
7. $S \rightarrow C$: SSH_MSG_KEXDH_REPLY$K_S, f, sign_{K_S}(H)$
   where $f = g^y$ for some server nonce y,
   $K = e^y$ and $H = hash(V_C, V_S, I_C, I_S, K_S, e, f, K)$,
   K_S is the server key
8. $S \rightarrow C$: SSH_MSG_NEWKEYS
9. $C \rightarrow S$: SSH_MSG_NEWKEYS

key exchange

10. …

session, incl. SSH authentication and connection protocols

Typical protocol spec given as Message Sequence Chart or in Alice-Bob style.

NB *oversimplifies* because it only specifies *one* correct run, *the happy flow*

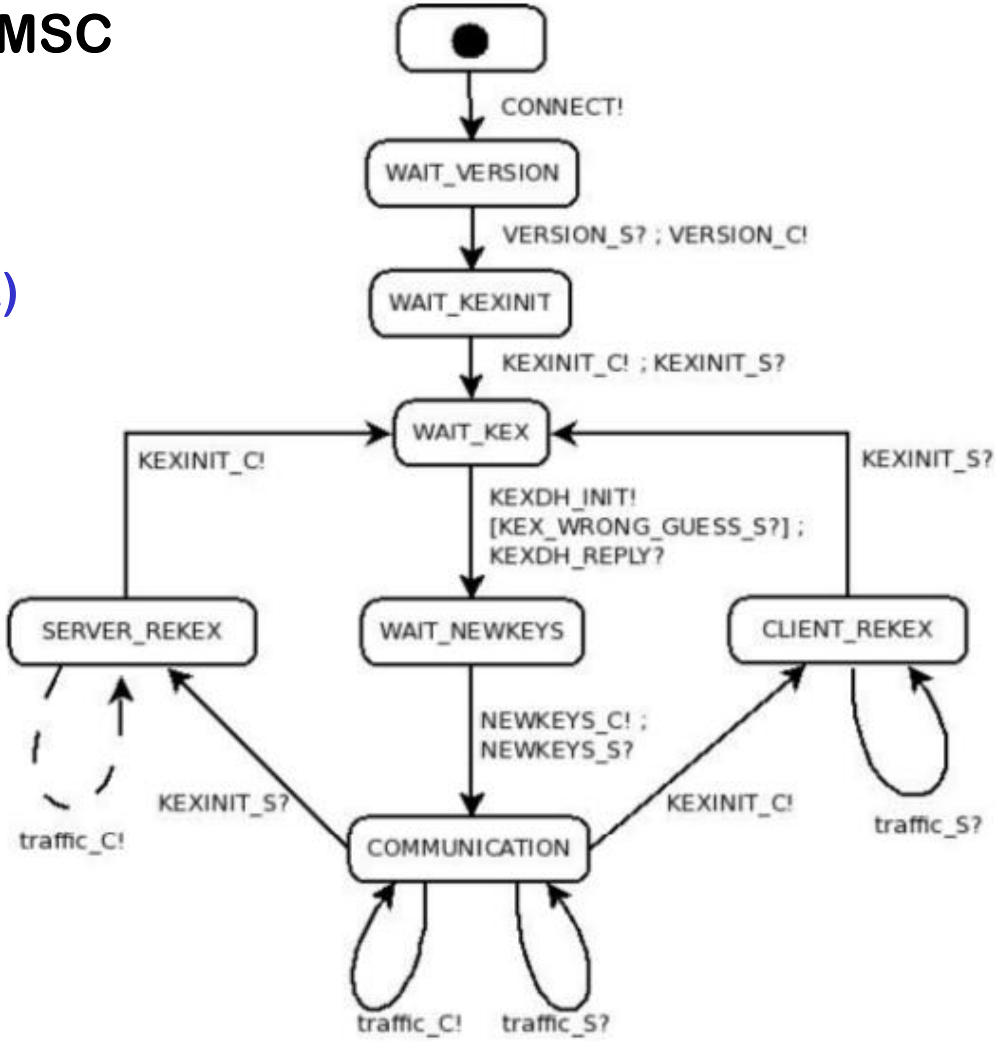
Protocol state machines

Most protocols allow more than just one specific happy flow described by an MSC

A better spec can be given using a Finite State Machine (FSM)
aka Deterministic Finite Automaton (DFA)

This still oversimplifies:
it still only describes happy flows,
albeit several instead of just one

Any implementation of the protocol will have to be input-enabled



SSH transport layer

input enabled state machines

A state machine is input enabled iff

in *every* state

it is able to receive *every* message

Often, many messages go to

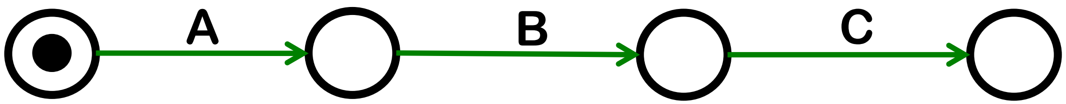
1) some error state,

2) back to the initial state, or

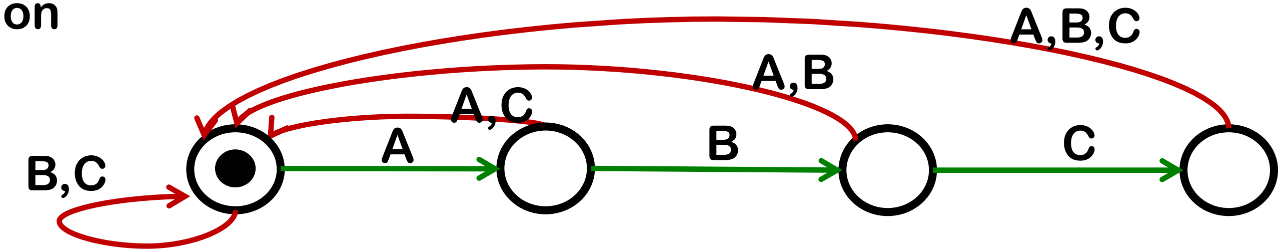
3) are ignored

input enabling

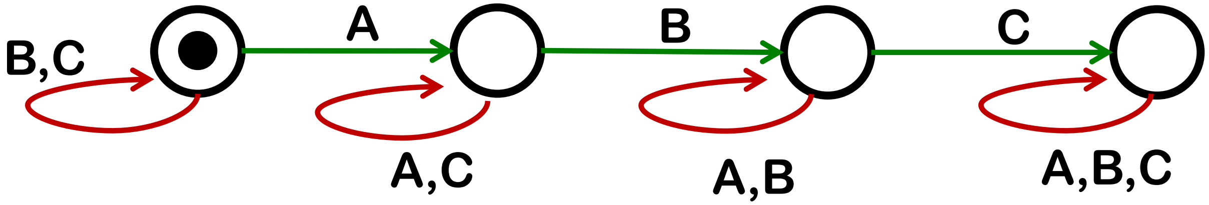
State machine that is not input-enabled



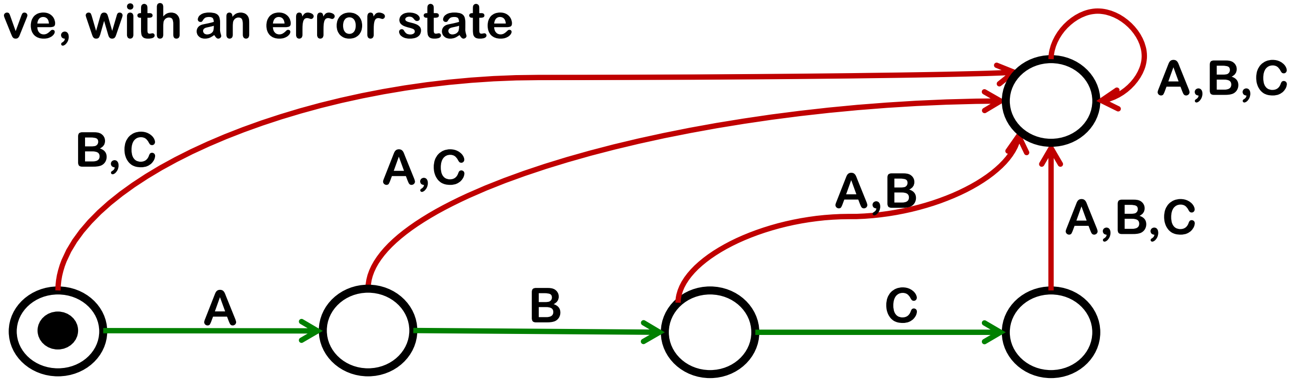
Input enabled version



Alternative input enabled version



Yet another alternative, with an error state



Typical prose specifications: SSH ☹ [RFCs 4251-4254]

“Once a party has sent a SSH_MSG_KEXINIT message for key exchange or re-exchange, until it has sent a SSH_MSG_NEWKEYS message, it MUST NOT send any messages other than:

- Transport layer generic messages (1 to 19) (but SSH_MSG_ SERVICE REQUEST and SSH_MSG_SERVICE_ACCEPT MUST NOT be sent);
- Algorithm negotiation messages (20 to 29) (but further SSH_MSG KEXINIT messages MUST NOT be sent);
- Specific key exchange method messages (30 to 49).”

“The provisions of Section 11 apply to unrecognised messages”

In Section 11:

> “An implementation MUST respond to all unrecognised messages with an SSH_MSG_UNIMPLEMENTED. Such messages MUST be otherwise ignored. Later protocol versions may define other meanings for these message types.”

Understanding protocol state machine from prose is hard!

Example security flaw due to flawed state machine

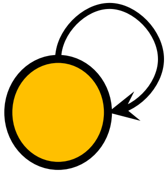
CVE-2018-10933

libssh versions 0.6 and above have an authentication bypass vulnerability in the server code. By presenting the server an SSH2_MSG_USERAUTH_*SUCCESS* message in place of the SSH2_MSG_USERAUTH_*REQUEST* message which the server would expect to initiate authentication, the attacker could successfully authenticate without any credentials.

https://www.libssh.org/security/advisories/CVE-2018-10933.txt

More example security flaws due to flawed state machines

- **MIDPSSH**

  no state machine implemented at all 

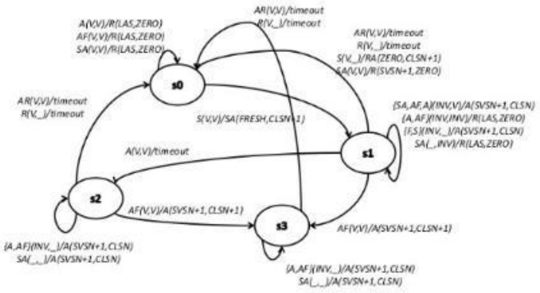
  [Verifying an implementation of SSH, WIST 2007]

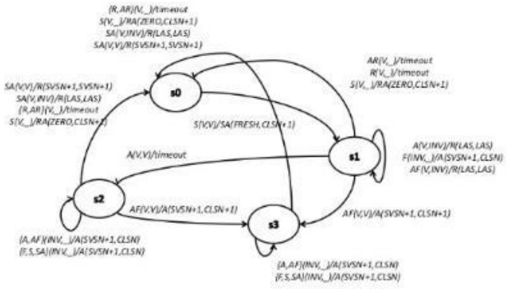
- **e.dentifier2**

  strange sequence of USB commands by-passes OK

  [Designed to fail: a USB-connected reader for online banking , NordSec 2012]

There can also be fingerprinting possibilities due to differences in implemented protocol state machines, eg in e-passports from different countries or in TCP implementations on Windows/Linux





Extracting protocol state machines from code

We can infer finite state machines from implementations by black box testing using state machine inference/learning

- using L* algorithm, as implemented in eg. LearnLib

This is effectively a form of 'stateful' fuzzing using a test harness that sends typical protocol messages.

For fuzzing we send *strange inputs*,

for state machine learning we send *strange sequences of normal inputs*

It can also be regarded as a form of automated reverse engineering

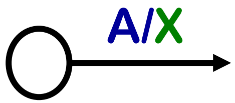
It is a great way to obtain protocol state machines

- *without reading specs!*
- *without reading code!*

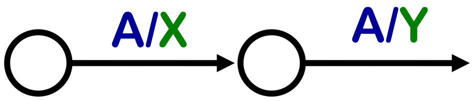
State machine inference, eg using LearnLib

Just try out many sequences of inputs, and observe outputs

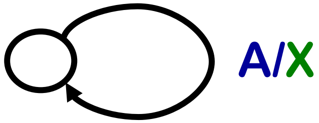
Suppose input A results in output X



- If second input A results in *different* output Y

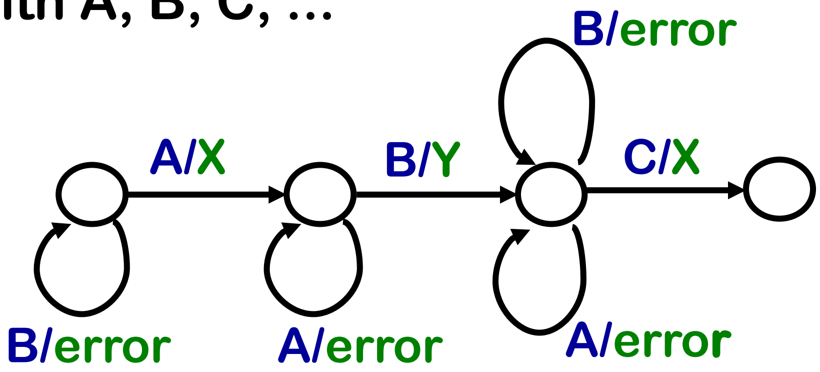


- If second input A results in the *same* output X



Now try more sequences of inputs with A, B, C, ...

to e.g. infer



The inferred state machine is an *under*-approximation of real system

Case study 1: EMV

- Most banking smartcards implement a variant of EMV
- EMV (Europay-Mastercard-Visa) defines set of protocols

  with *lots* of variants

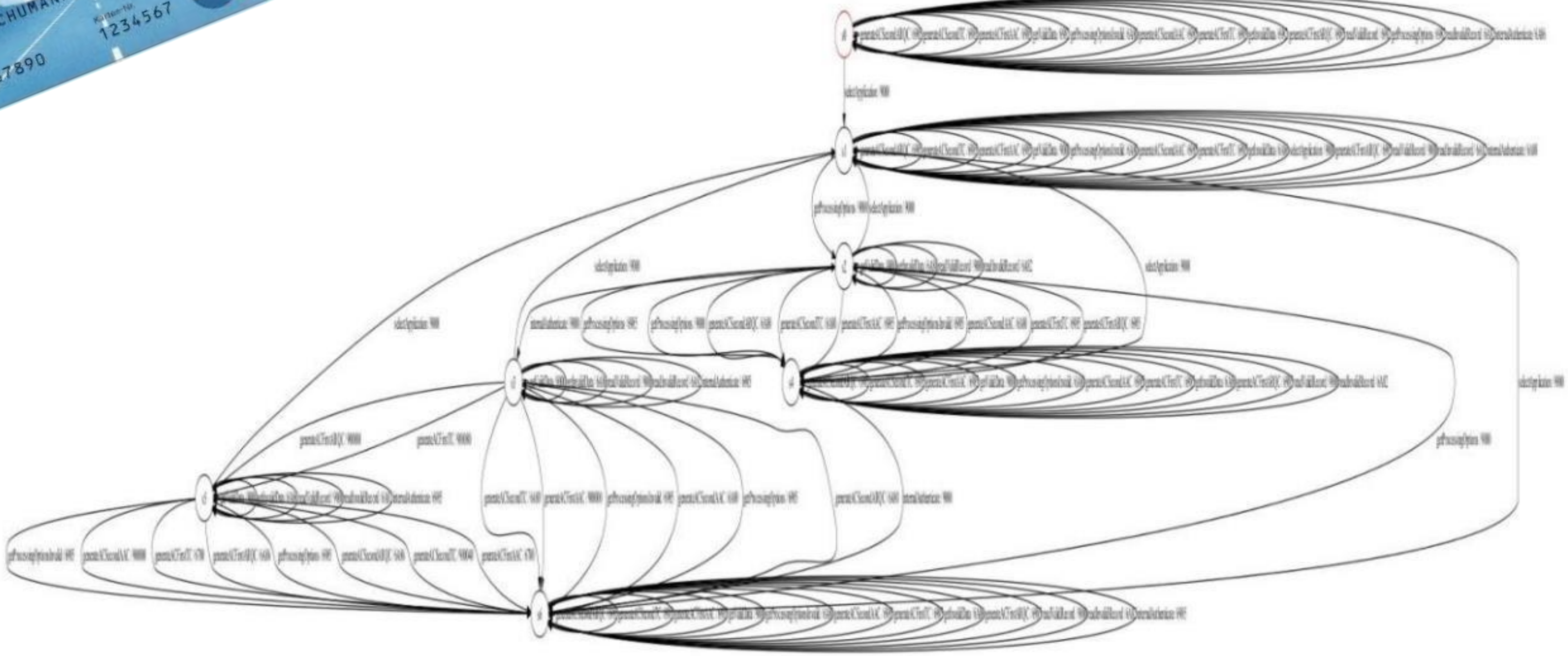
- Specs controlled by EMVCo which is owned by MasterCard VISA AMERICAN EXPRESS JCB
- Specification in 4 books totalling > 700 pages
- EMV contactless: 10 more books, > 1500 pages

Problem: complexity

One sentence taken from one of the EMV contactless specs

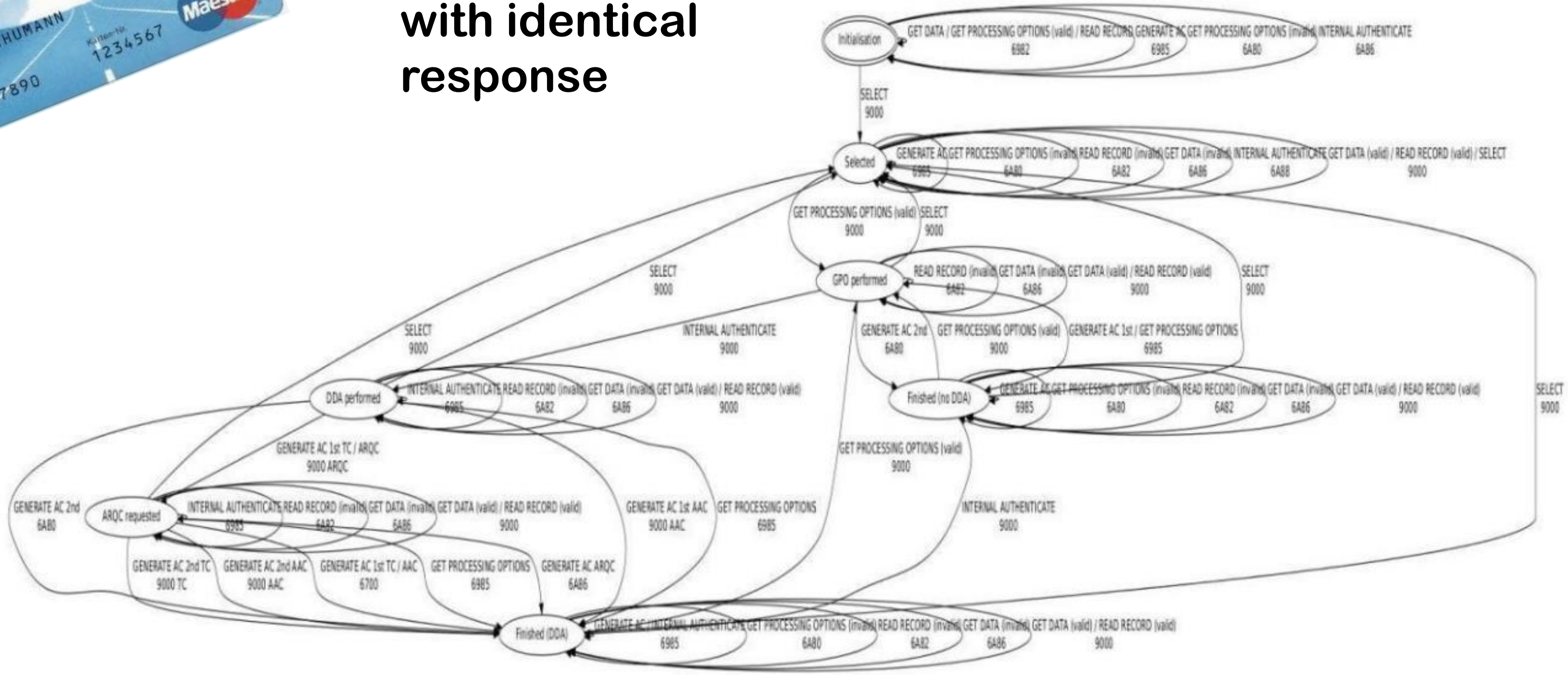
"If the card responds to GPO with SW1 SW2 = x9000 and AIP byte 2 bit 8 set to 0,
and if the reader supports qVSDC and contactless VSDC,
then if the Application Cryptogram (Tag '9F26') is present in the GPO response,
then the reader shall process the transaction as qVSDC,
and if Tag '9F26' is not present,
then the reader shall process the transaction as VSDC."

State machine inference of Maestro card

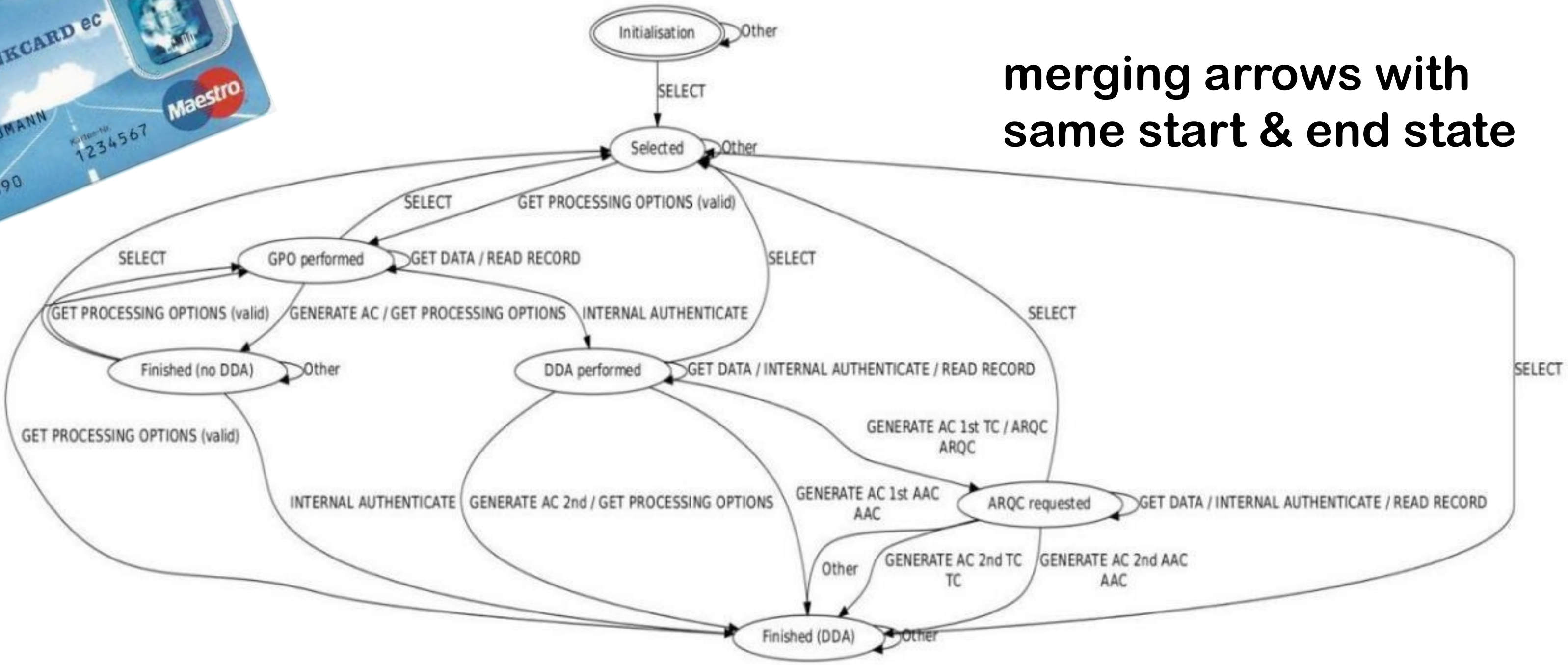


State machine inference of Maestro card

merging arrows with identical response



State machine inference of Maestro card



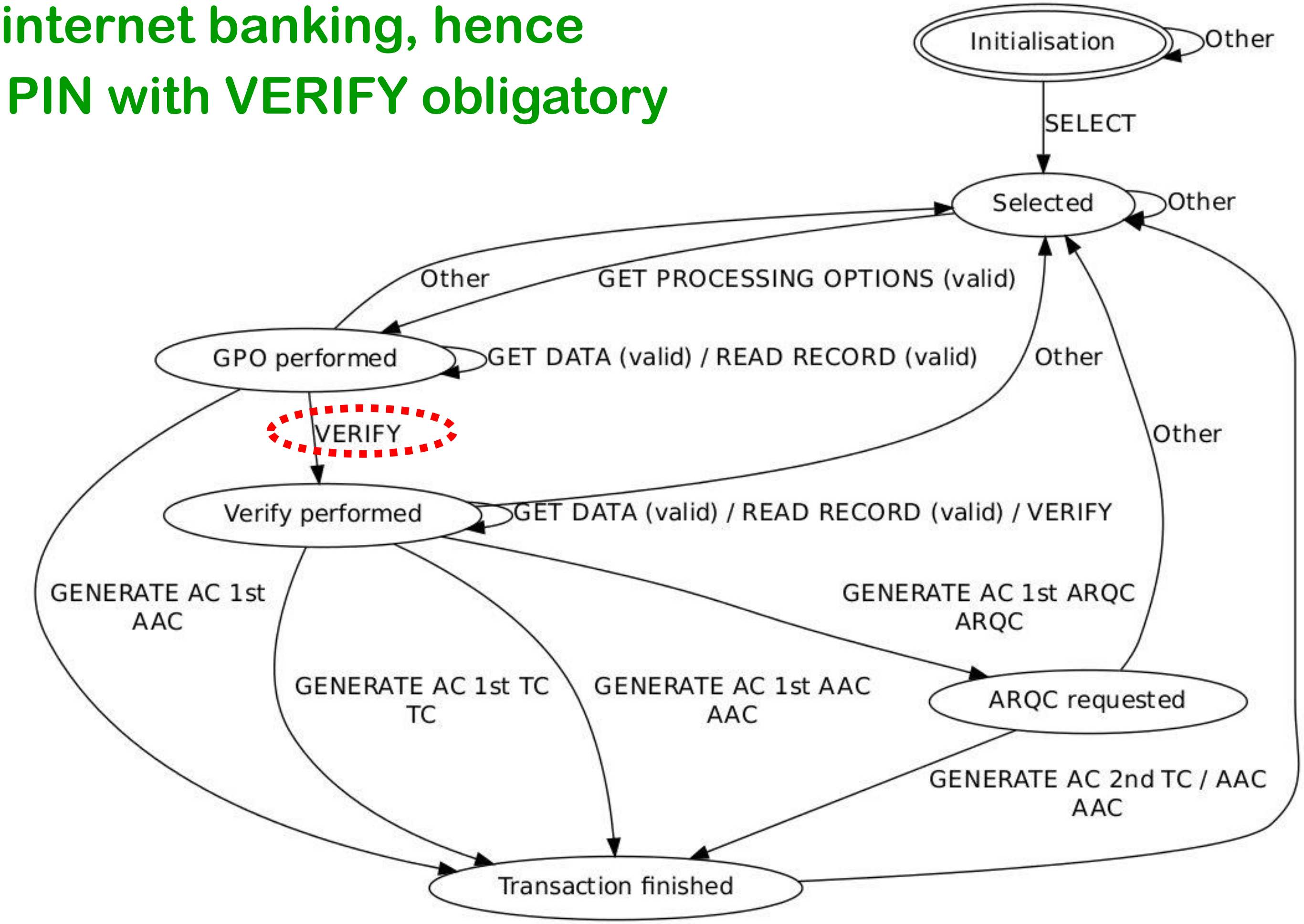
merging arrows with
same start & end state

We found no bugs, but lots of variety between cards.

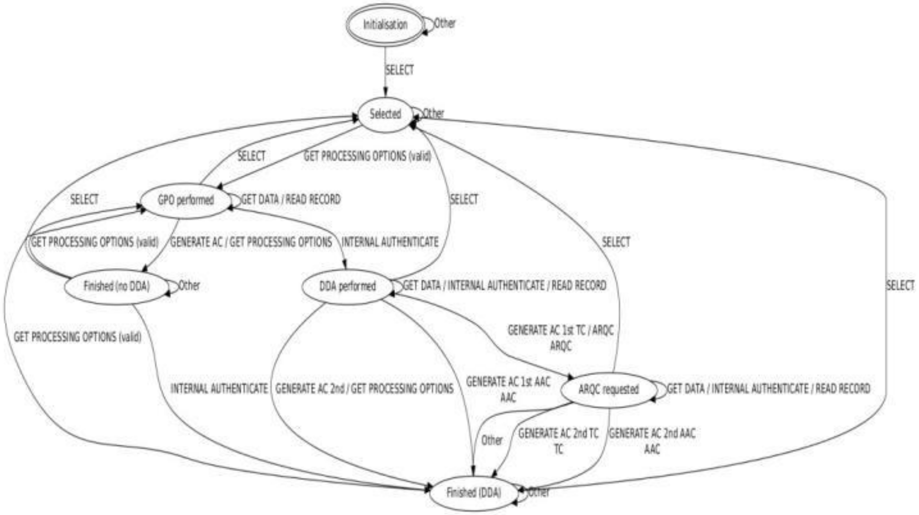
[Fides Aarts et al., Formal models of bank cards for free, SECTEST 2013]

SecureCode application on Rabobank card

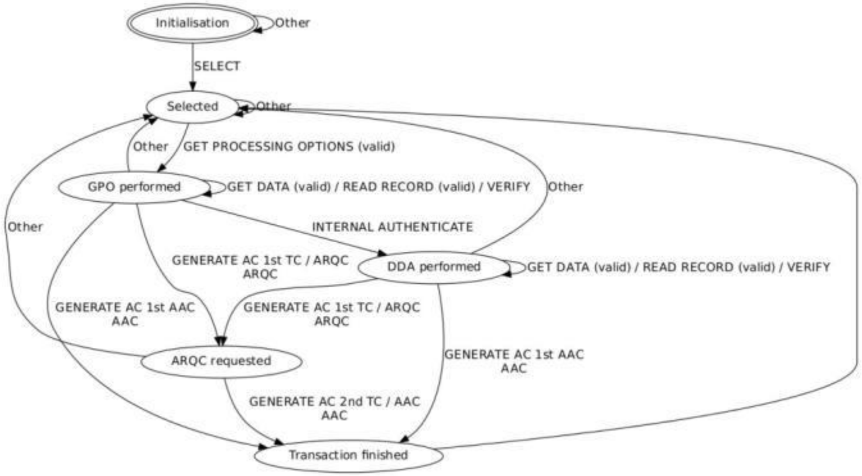
used for internet banking, hence entering PIN with VERIFY obligatory



Understanding & comparing EMV implementations



Volksbank Maestro implementation



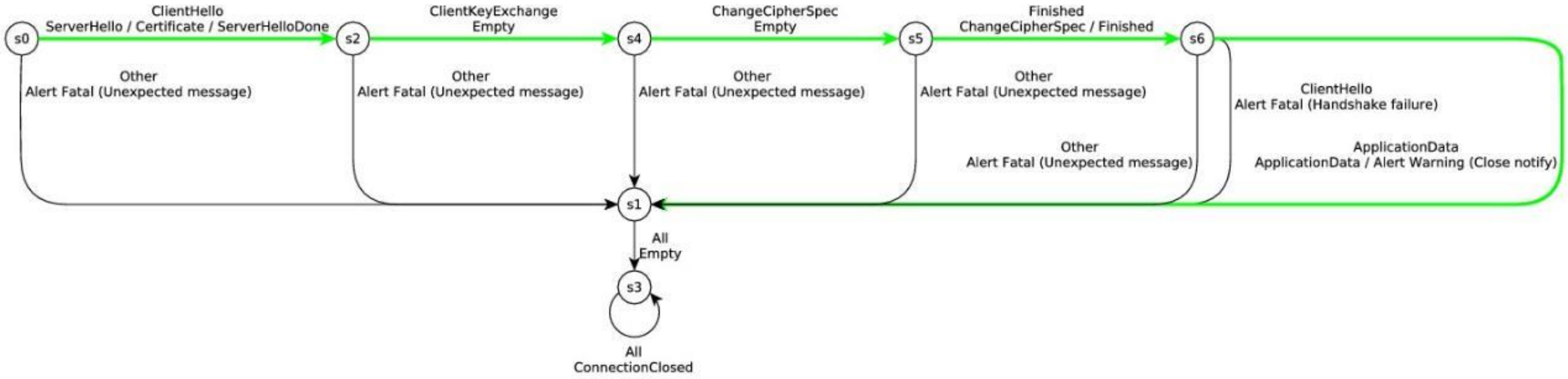
Rabobank Maestro implementation

Are both implementations correct & secure? And compatible?

Presumably they both pass a Maestro compliance test-suite...

So some paths (and maybe some states) are superfluous?

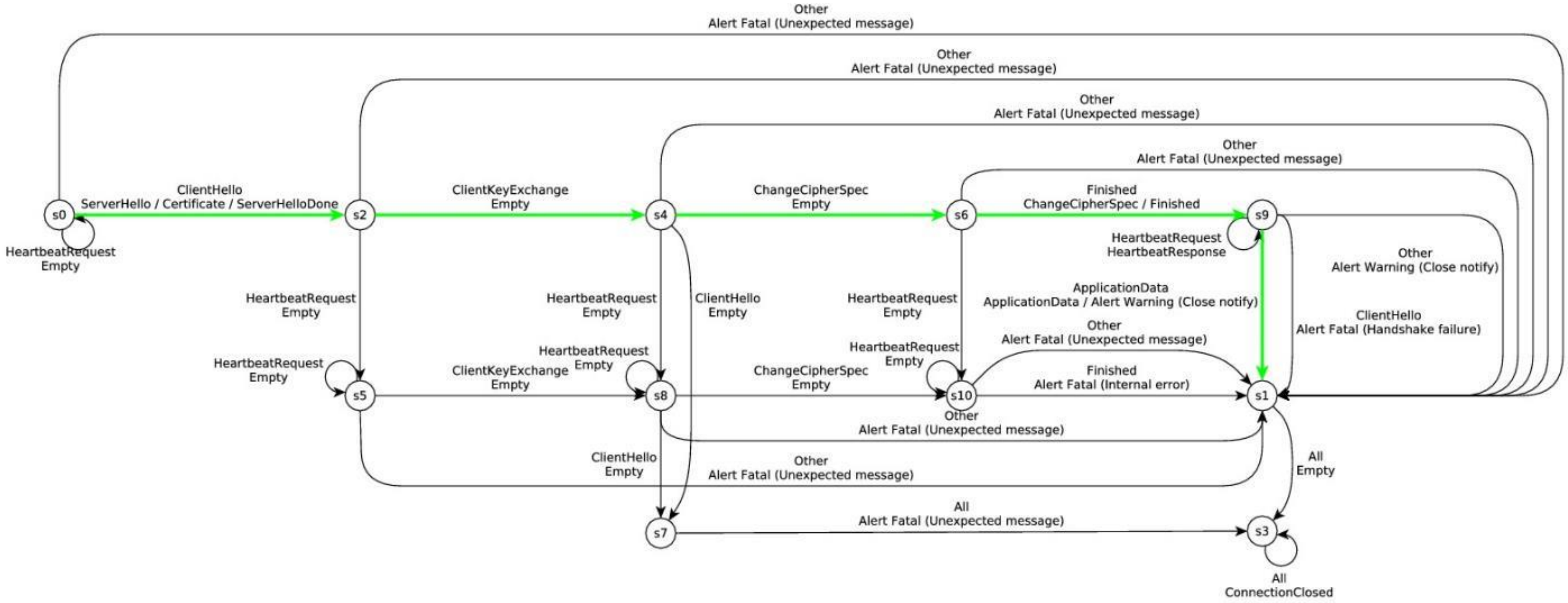
Case study 2: TLS


s0
s1
s2
s3
s4
s5
s6
ClientHello
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
Finished
ChangeCipherSpec / Finished
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
ClientHello
Alert Fatal (Handshake failure)
ApplicationData
ApplicationData / Alert Warning (Close notify)
All
Empty
All
ConnectionClosed

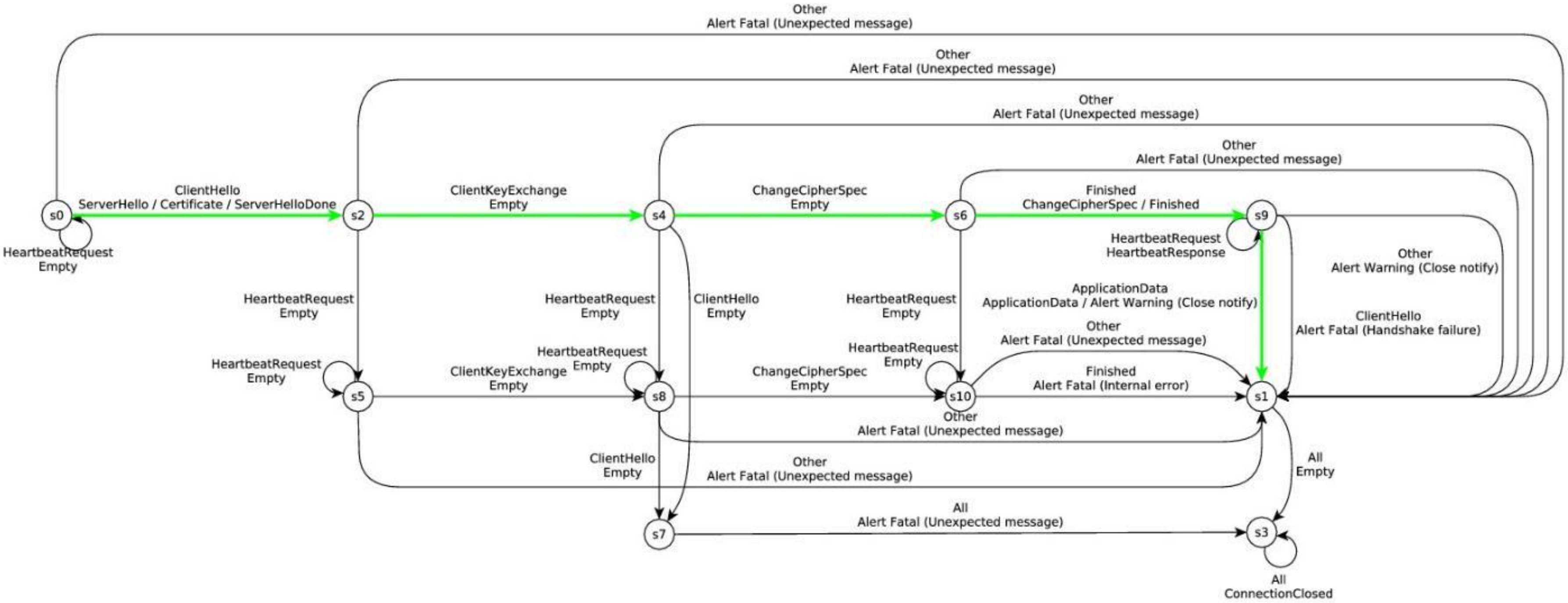
State machine inferred from NSS implementation

Comforting to see this is so simple!

TLS… according to GnuTLS

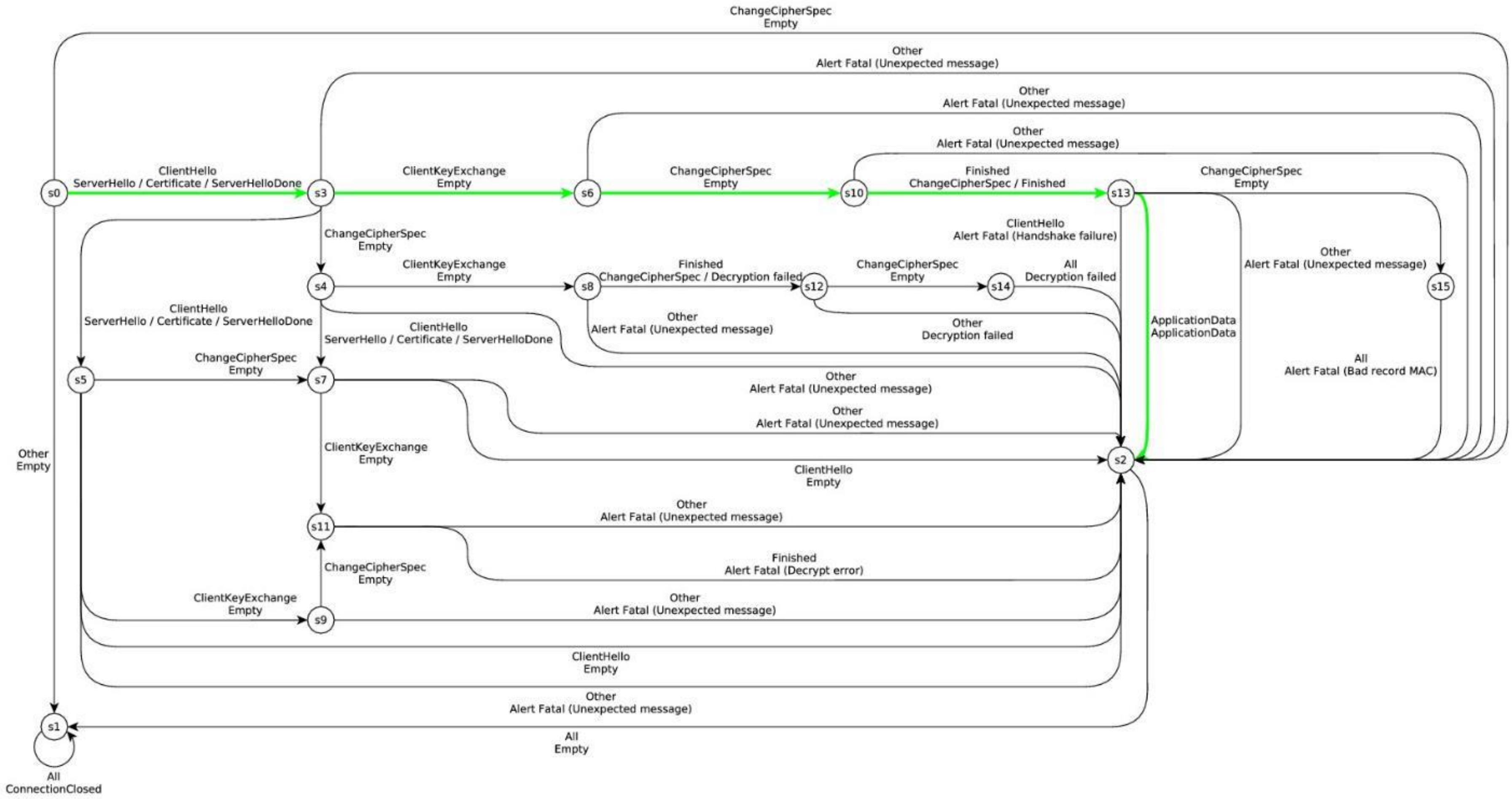

Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
s0
ClientHello
ServerHello / Certificate / ServerHelloDone
s2
ClientKeyExchange
Empty
s4
ChangeCipherSpec
Empty
s6
Finished
ChangeCipherSpec / Finished
s9
HeartbeatRequest
Empty
HeartbeatRequest
HeartbeatResponse
Other
Alert Warning (Close notify)
HeartbeatRequest
Empty
HeartbeatRequest
Empty
ClientHello
Empty
HeartbeatRequest
Empty
ApplicationData
ApplicationData / Alert Warning (Close notify)
ClientHello
Alert Fatal (Handshake failure)
Other
Alert Fatal (Unexpected message)
HeartbeatRequest
Empty
HeartbeatRequest
Empty
ClientKeyExchange
Empty
HeartbeatRequest
Empty
ChangeCipherSpec
Empty
Finished
Alert Fatal (Internal error)
s5
s8
s10
s1
Other
Alert Fatal (Unexpected message)
ClientHello
Empty
Other
Alert Fatal (Unexpected message)
All
Empty
All
Alert Fatal (Unexpected message)
s7
s3
All
ConnectionClosed

TLS... according to GnuTLS

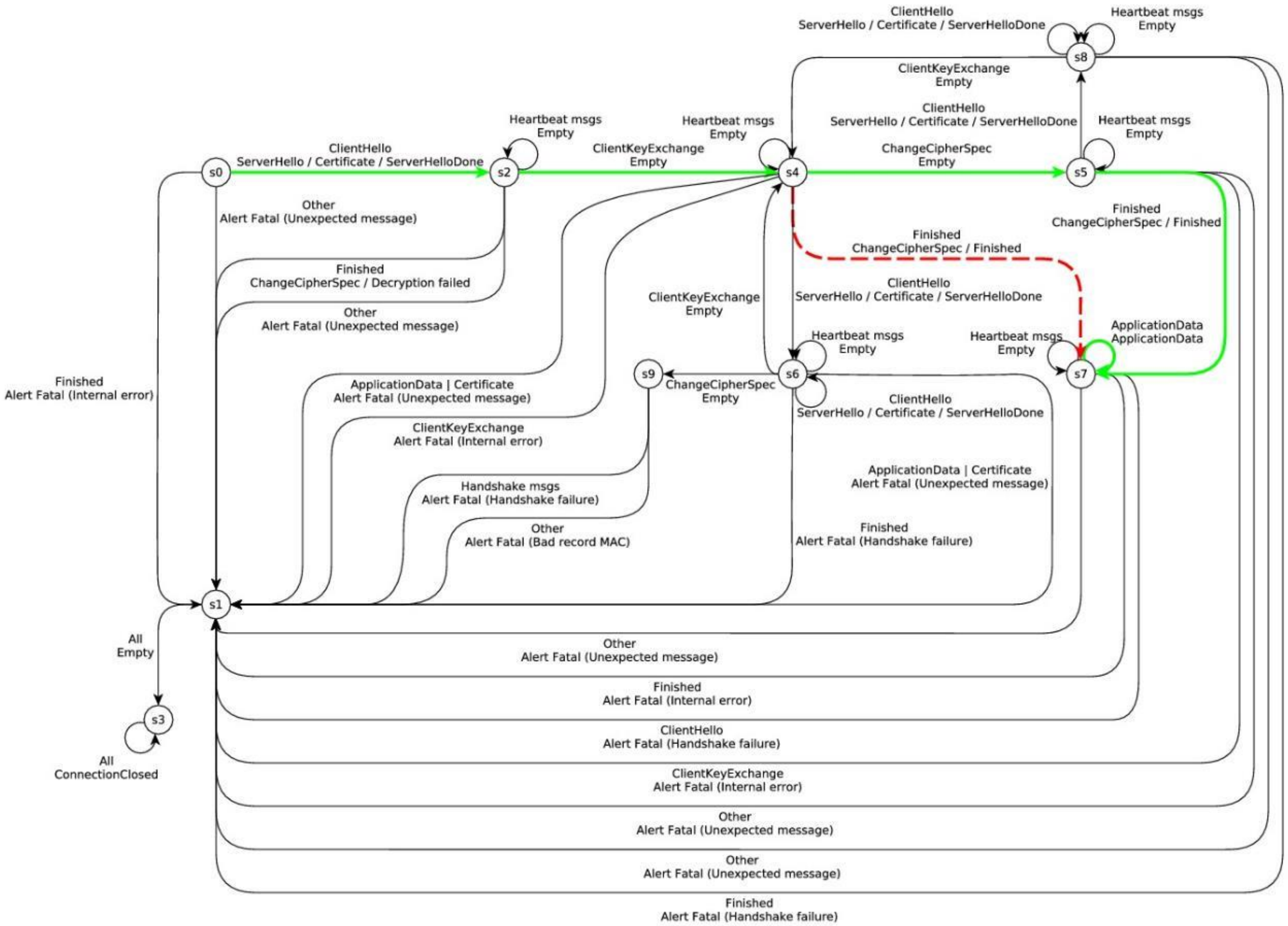


Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
ClientHello
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
Finished
ChangeCipherSpec / Finished
s0
s2
s4
s6
s9
HeartbeatRequest
Empty
HeartbeatRequest
HeartbeatResponse
Other
Alert Warning (Close notify)
HeartbeatRequest
Empty
HeartbeatRequest
Empty
ClientHello
Empty
HeartbeatRequest
Empty
ApplicationData
ApplicationData / Alert Warning (Close notify)
ClientHello
Alert Fatal (Handshake failure)
Other
Alert Fatal (Unexpected message)
HeartbeatRequest
Empty
HeartbeatRequest
Empty
HeartbeatRequest
Empty
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
Finished
Alert Fatal (Internal error)
s5
s8
s10
s1
Other
Alert Fatal (Unexpected message)
ClientHello
Empty
Other
Alert Fatal (Unexpected message)
All
Empty
All
Alert Fatal (Unexpected message)
s7
s3
All
ConnectionClosed

TLS... according to OpenSSL

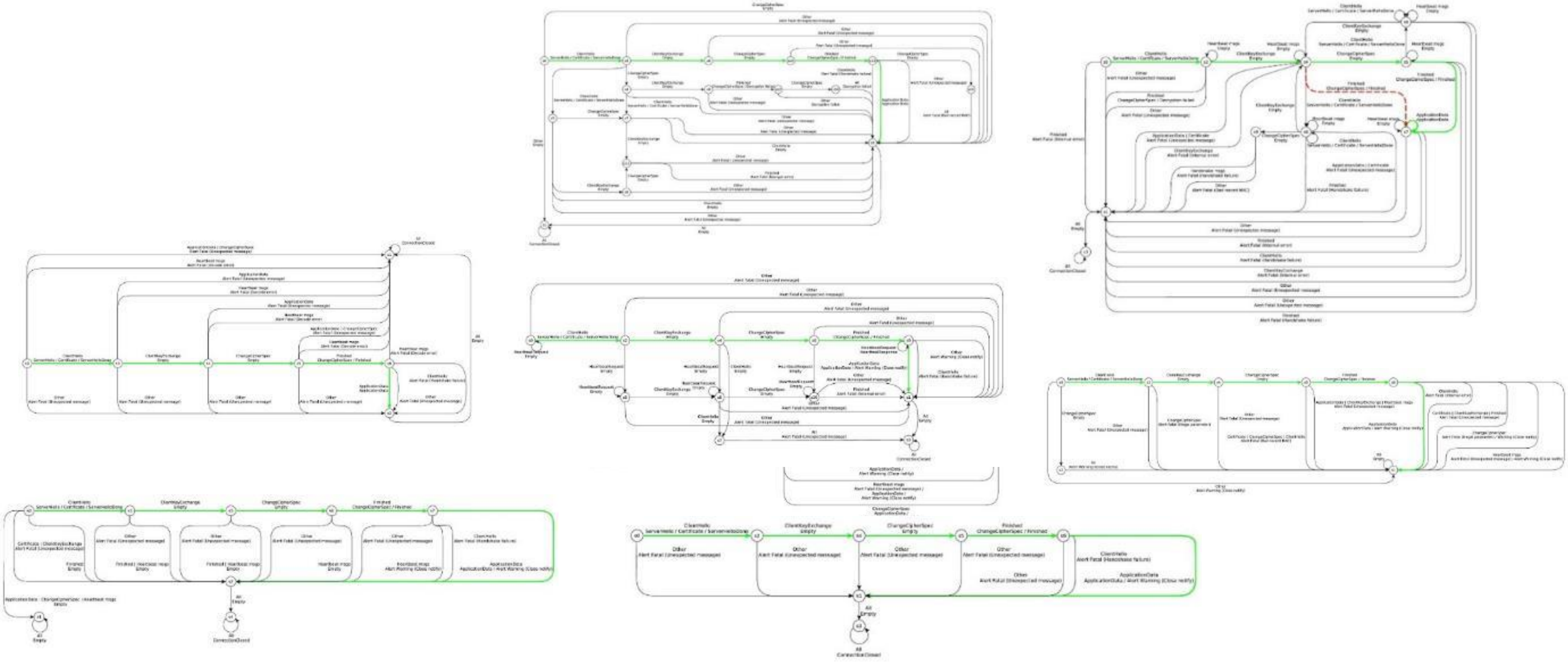

ChangeCipherSpec
Empty
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
s0
ClientHello
ServerHello / Certificate / ServerHelloDone
s3
ClientKeyExchange
Empty
s6
ChangeCipherSpec
Empty
s10
Finished
ChangeCipherSpec / Finished
s13
ChangeCipherSpec
Empty
ChangeCipherSpec
Empty
ClientHello
Alert Fatal (Handshake failure)
Other
Alert Fatal (Unexpected message)
s4
ClientKeyExchange
Empty
s8
Finished
ChangeCipherSpec / Decryption failed
s12
ChangeCipherSpec
Empty
s14
All
Decryption failed
s15
ClientHello
ServerHello / Certificate / ServerHelloDone
Other
Alert Fatal (Unexpected message)
ClientHello
ServerHello / Certificate / ServerHelloDone
Other
Decryption failed
ApplicationData
ApplicationData
ChangeCipherSpec
Empty
s5
s7
Other
Alert Fatal (Unexpected message)
All
Alert Fatal (Bad record MAC)
Other
Alert Fatal (Unexpected message)
Other
Empty
ClientKeyExchange
Empty
s2
ClientHello
Empty
Other
Alert Fatal (Unexpected message)
s11
Finished
Alert Fatal (Decrypt error)
ChangeCipherSpec
Empty
ClientKeyExchange
Empty
s9
Other
Alert Fatal (Unexpected message)
ClientHello
Empty
Other
Alert Fatal (Unexpected message)
s1
All
Empty
All
ConnectionClosed

TLS... according to Java Secure Socket Exension



ClientHello
ServerHello / Certificate / ServerHelloDone
Heartbeat msgs
Empty
s8
ClientKeyExchange
Empty
ClientHello
ServerHello / Certificate / ServerHelloDone
Heartbeat msgs
Empty
Heartbeat msgs
Empty
Heartbeat msgs
Empty
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
ClientHello
ServerHello / Certificate / ServerHelloDone
s0
s2
s4
s5
Other
Alert Fatal (Unexpected message)
Finished
ChangeCipherSpec / Finished
Finished
ChangeCipherSpec / Finished
Finished
ChangeCipherSpec / Decryption failed
ClientHello
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Empty
Other
Alert Fatal (Unexpected message)
Heartbeat msgs
Empty
Heartbeat msgs
Empty
ApplicationData
ApplicationData
Finished
Alert Fatal (Internal error)
s9
s6
s7
ApplicationData | Certificate
Alert Fatal (Unexpected message)
ChangeCipherSpec
Empty
ClientHello
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Alert Fatal (Internal error)
ApplicationData | Certificate
Alert Fatal (Unexpected message)
Handshake msgs
Alert Fatal (Handshake failure)
Finished
Alert Fatal (Handshake failure)
Other
Alert Fatal (Bad record MAC)
s1
All
Empty
Other
Alert Fatal (Unexpected message)
Finished
Alert Fatal (Internal error)
s3
ClientHello
Alert Fatal (Handshake failure)
All
ConnectionClosed
ClientKeyExchange
Alert Fatal (Internal error)
Other
Alert Fatal (Unexpected message)
Other
Alert Fatal (Unexpected message)
Finished
Alert Fatal (Handshake failure)

Which TLS implementations are correct? or secure?



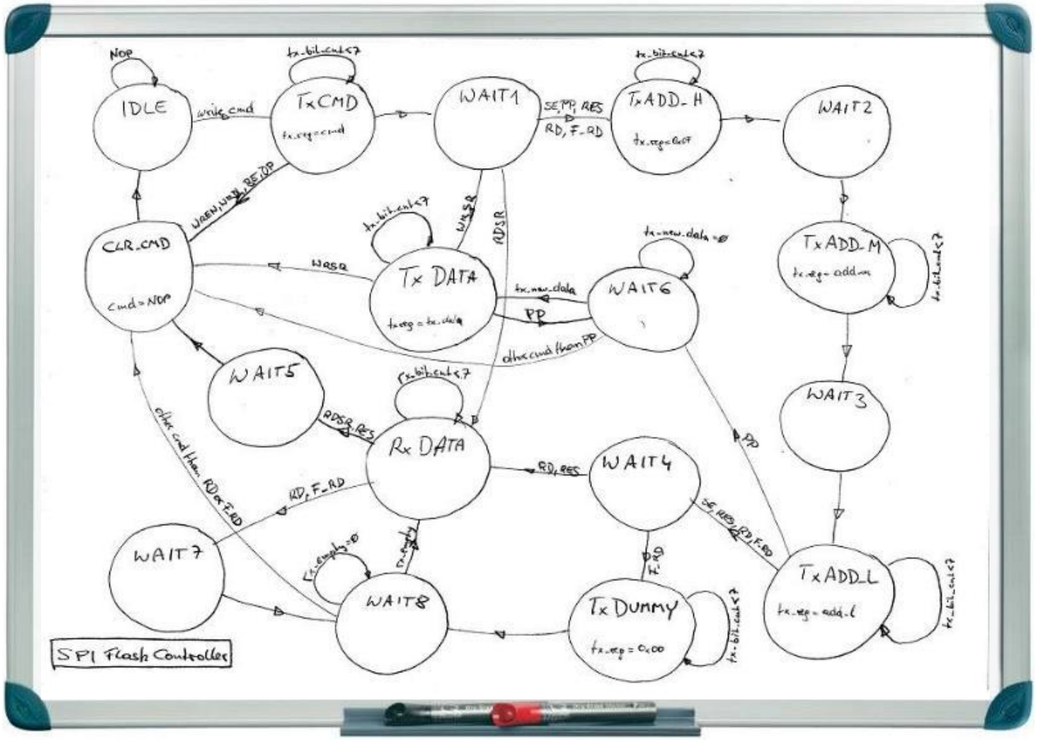
[Joeri de Ruiter et al., Protocol state fuzzing of TLS implementations, Usenix Security 2015]

Results with learning state machines for TLS

- For most TLS implementations, models can be learned within 1 hour
- Three security flaws can be found this way, in
  - OpenSSL
  - GnuTLS
  - Java Secure Socket Extention (JSSE)
- One (not security-critical) flaw found in newly proposed reference implementation nqbs-TLS

People who write specs, or make implementations, or do security analyses probably all draw state machines on their whiteboards...

But will it they all draw an identical one?



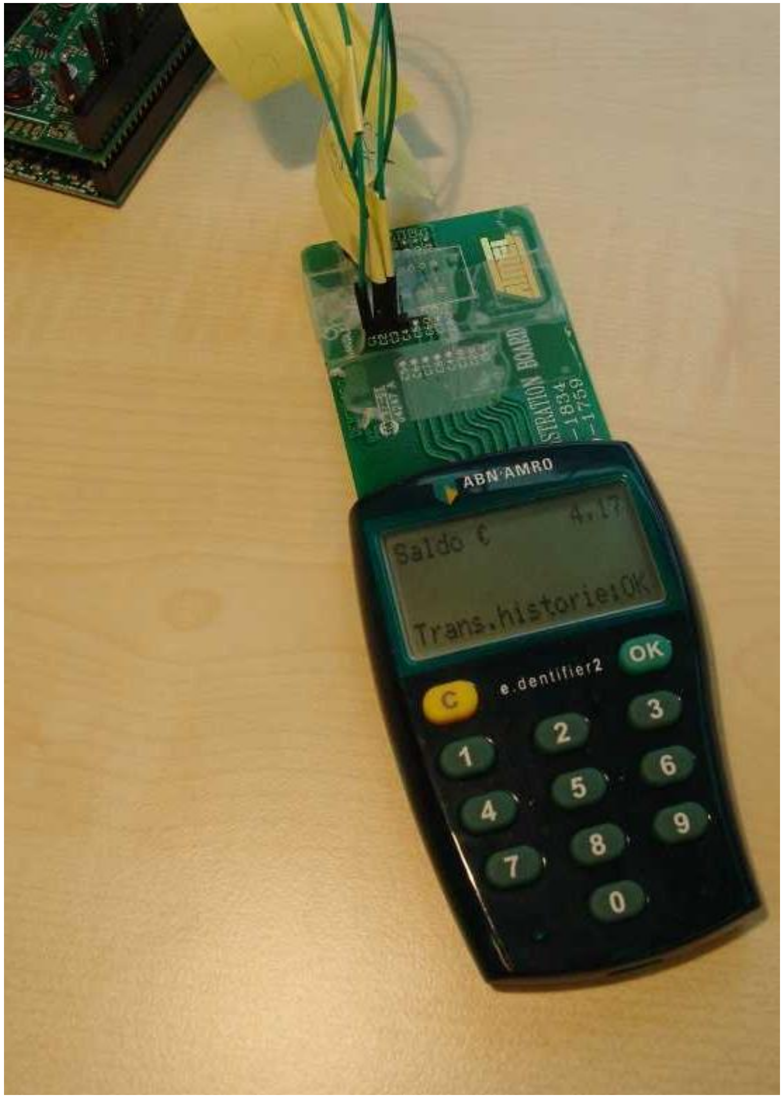
Case study 3: the USB-connected e.dentifier

Can we use state machine learning with

- USB commands
- user actions via keyboard

to obtain the state machine

of the ABN-AMRO e.dentifier2?

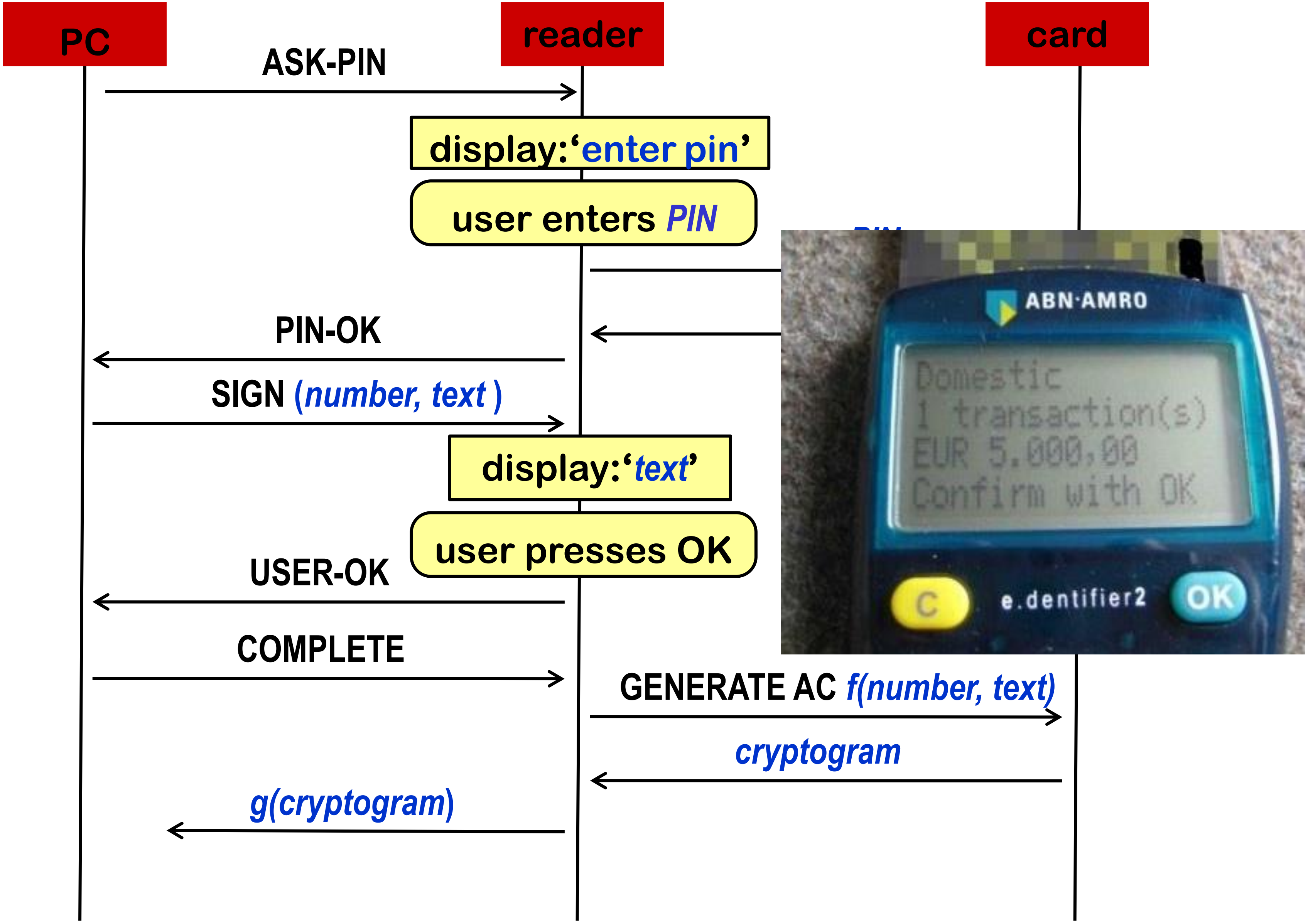


Earlier manual analysis

revealed the USB connection

has a flaw

(Manually) reverse-engineered protocol



PC | reader | card

ASK-PIN

display:'enter pin'

user enters *PIN*

PIN-OK

SIGN (*number, text*)

display:'*text*'

user presses OK

USER-OK

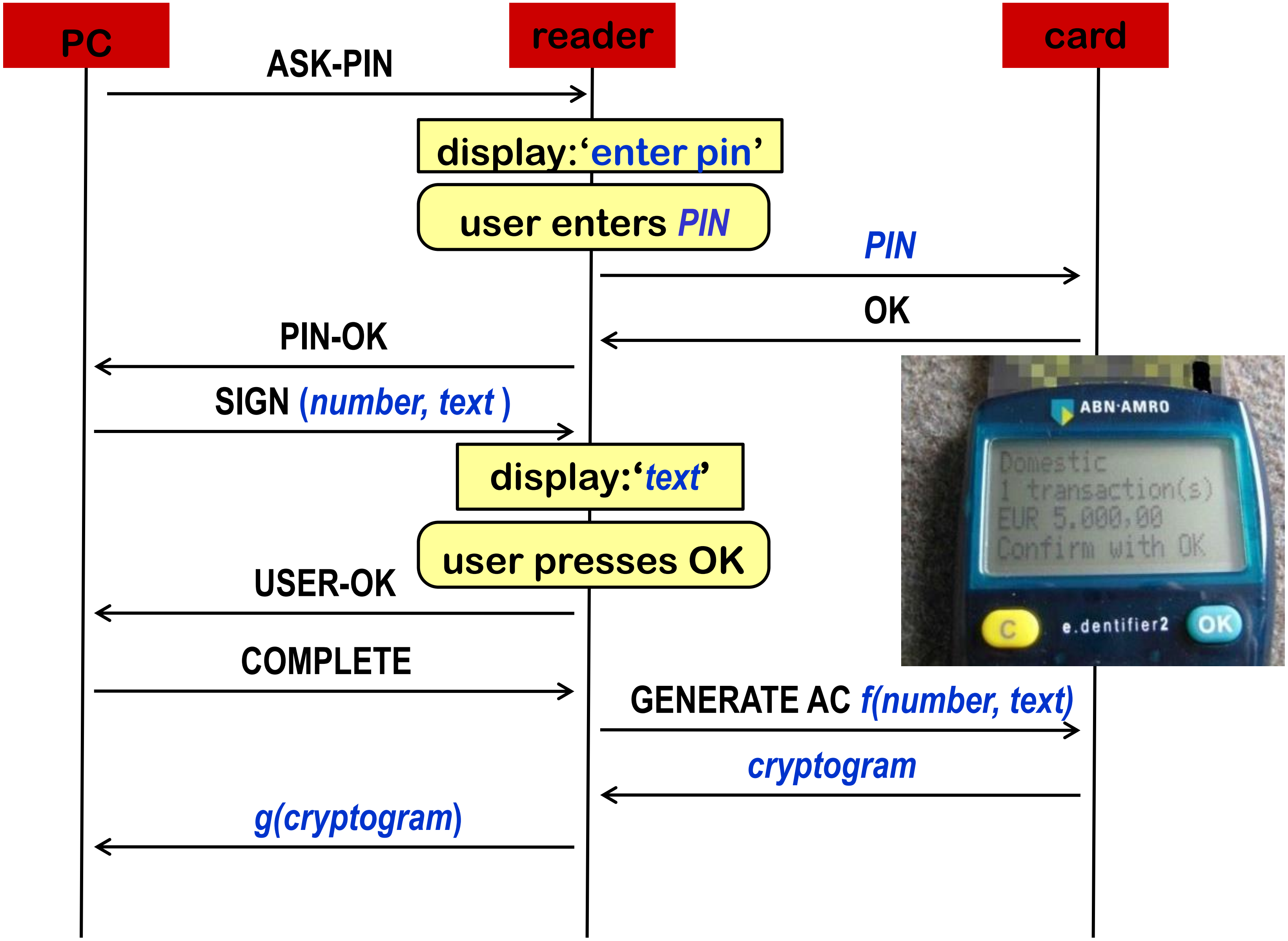
COMPLETE

GENERATE AC *f(number, text)*

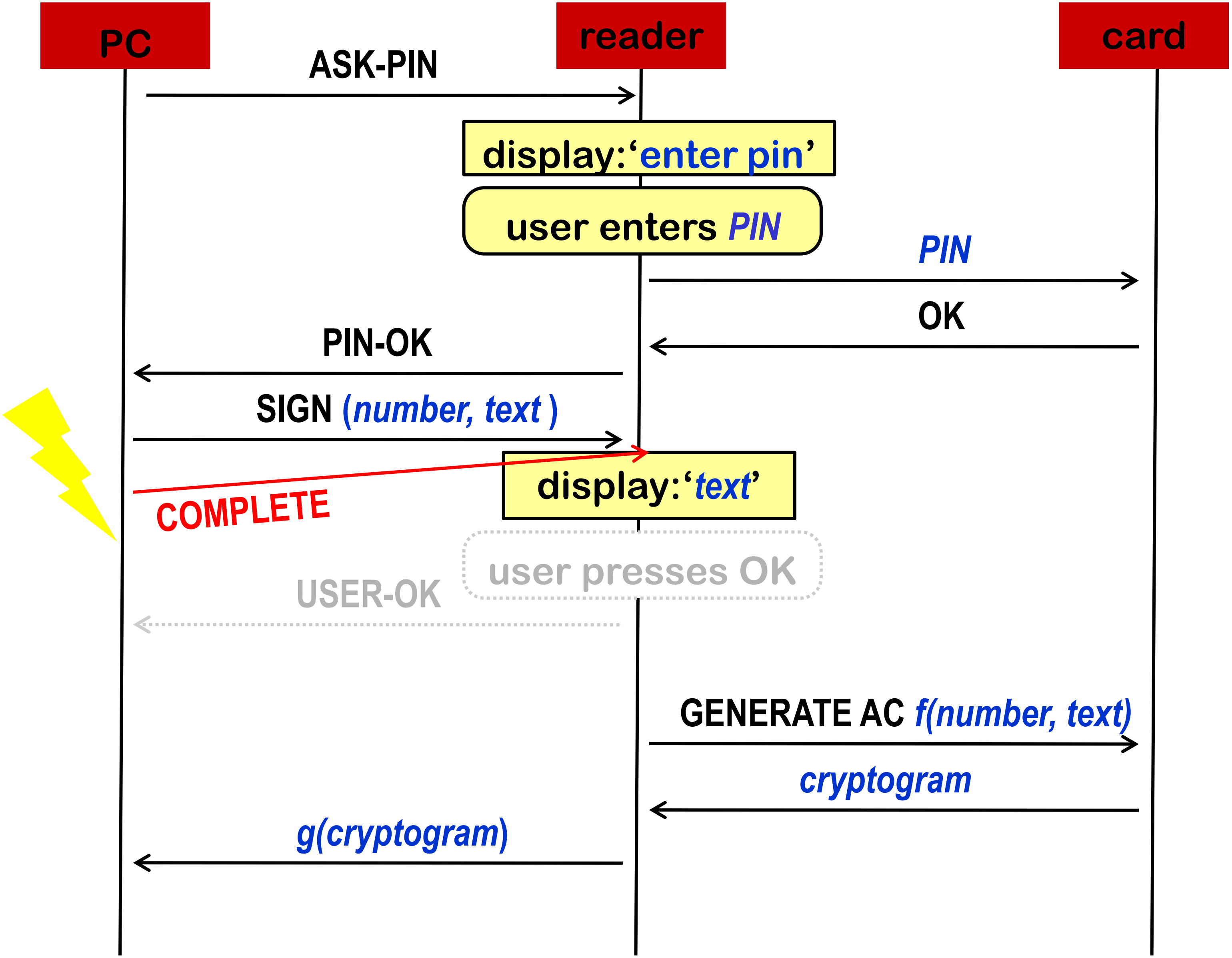
cryptogram

g(cryptogram)

Spot the defect!



Attack!



PC

reader

card

ASK-PIN

display:'*enter pin*'

user enters *PIN*

PIN

OK

PIN-OK

SIGN (*number, text*)

COMPLETE

display:'*text*'

user presses OK

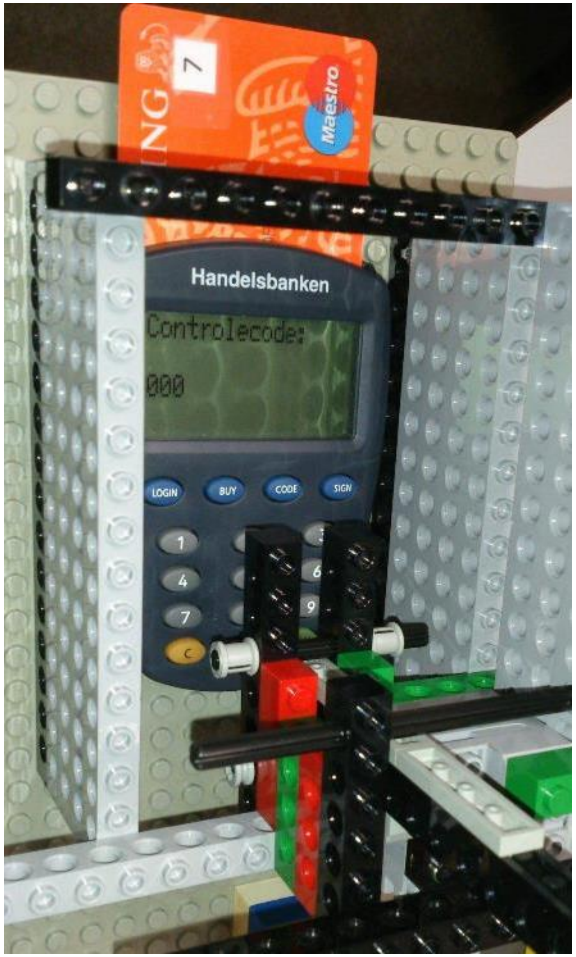
USER-OK

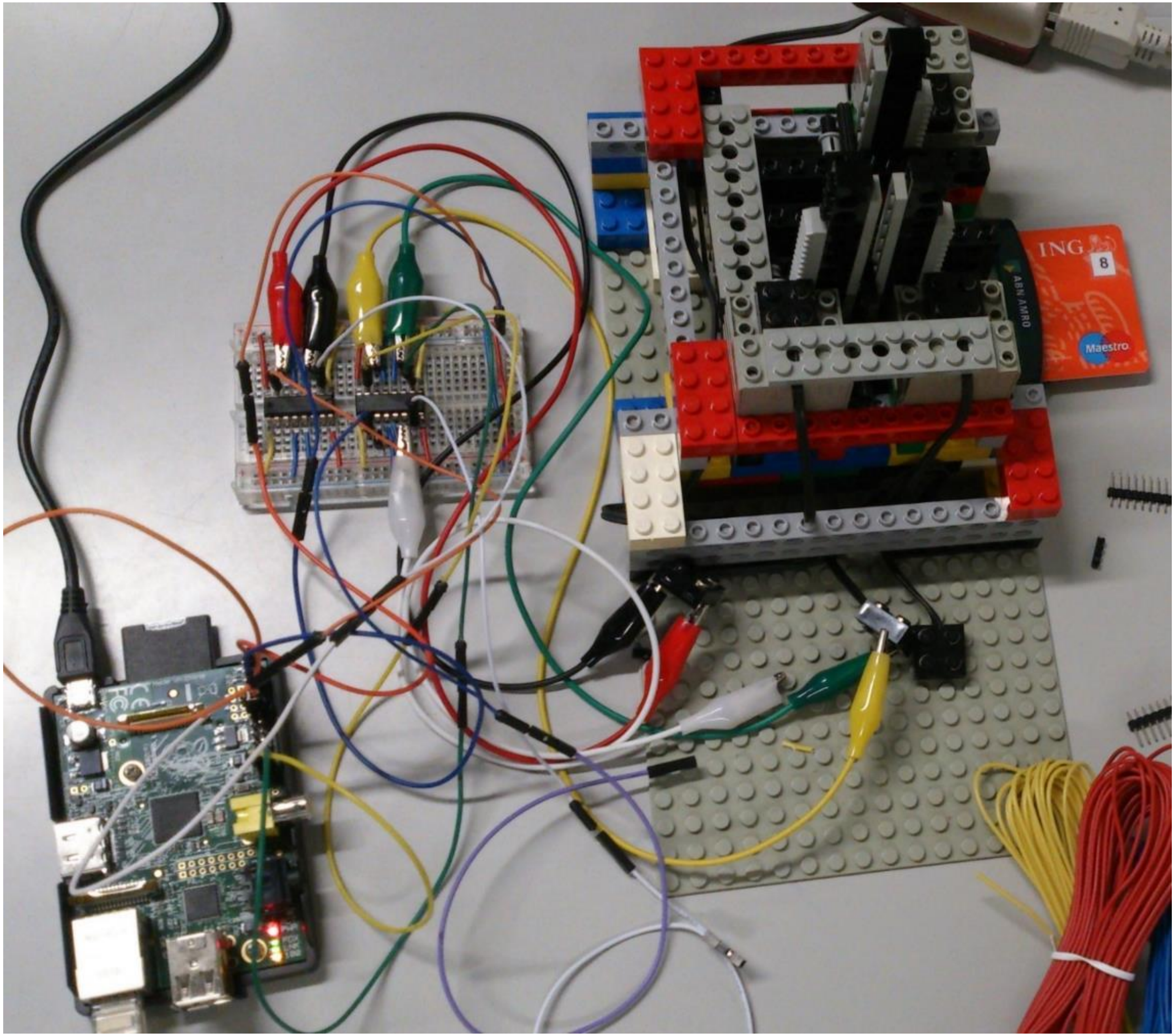
GENERATE AC *f(number, text)*

cryptogram

g(cryptogram)

Operating the keyboard using LEGO

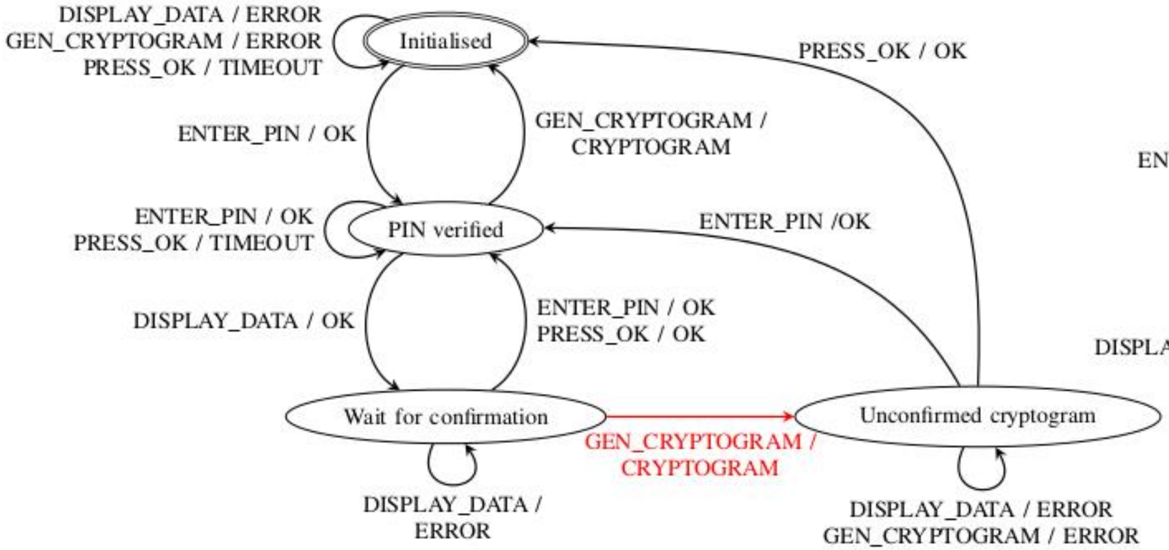


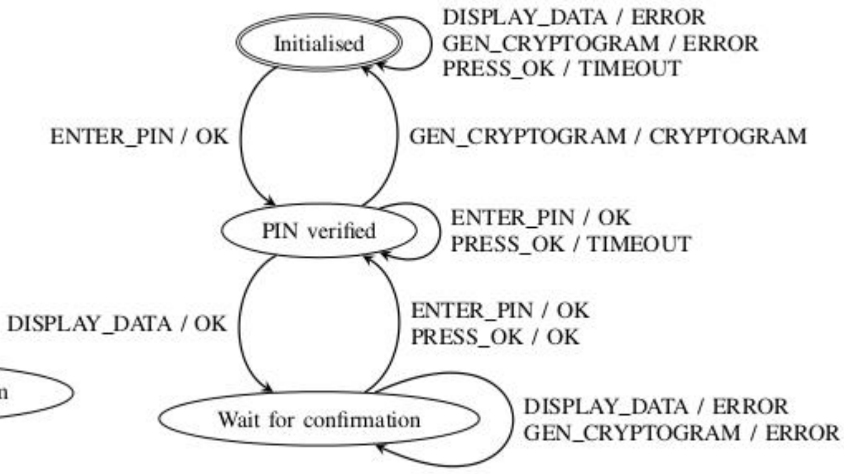


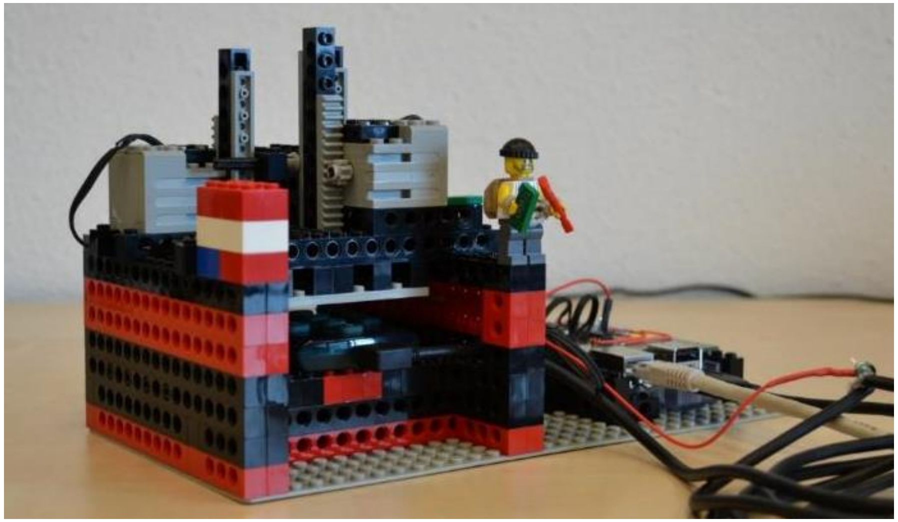
ING
8
ABN AMRO
Maestro

ABN·AMRO

State machines of old vs new e.dentifier2

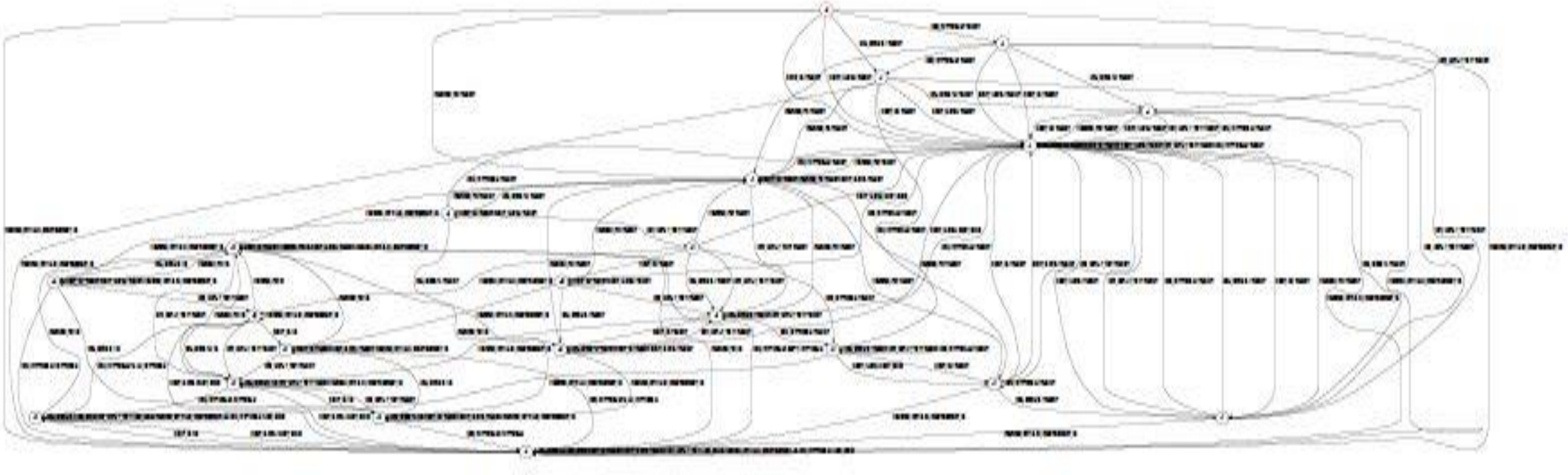






https://www.youtube.com/watch?v=hyQubPvAyq4

Would you trust this to be secure?



More detailed inferred state machine, using richer input alphabet.

Do you think whoever designed or implemented this is confident that this is secure?

Or that all this behaviour is necessary?

Results with learning state machines for e.dentifier2

- Coarse models, with a limited input alphabet, can be learnt in a few hours
  - detailed enough to show presence of the known security flaw in the old e.dentifier, and absence of this flaw in the new one
- The most detailed models required 8 hours or more
- The complexity of the obtained models suggest there was no clear protocol design as the basis for the implementation

[Georg Chalupar et al., Automated Reverse Engineering using Lego, WOOT 2014]

https://www.youtube.com/watch?v=hyQubPvAyq4

Conclusions: Protocol state machines

Rigorous & clear specifications of protocols using finite state machines can improve security:

- by avoiding ambiguities
- useful for programmer

In spec does not clearly specify a state machines, we can extract state machines from implementations using state machine learning. This is great for

- security testing & analysis of implementations
- obtaining reference state machines for legacy systems

Uses of protocol state machines

1. Analysing the models by hand, or with model checker, for flaws
   - to see if *all paths* are correct & secure
2. Using model when doing a manual code review
3. Fuzzing or model-based testing
   - using the diagram as basis for "deeper" fuzz testing

     eg fuzzing also parameters of commands
4. Program verification
   - *proving* that there is no functionality beyond that in the diagram, which using just testing you can never be sure of

The road we followed

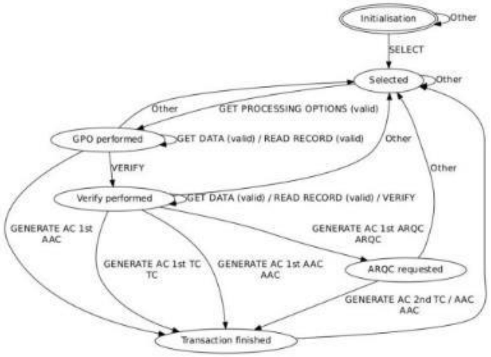
specs

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

state machine learning



model

Ideally specs would include a state machine!

specs

implementing

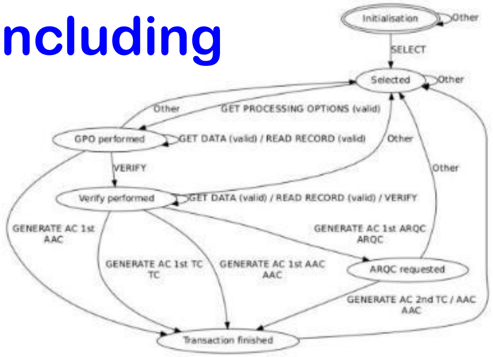
```
                                 Untitled
import java.util.*;
import java.text.*;

//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

model-based

testing

including



model

Or maybe we could generate code?

NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES
USA