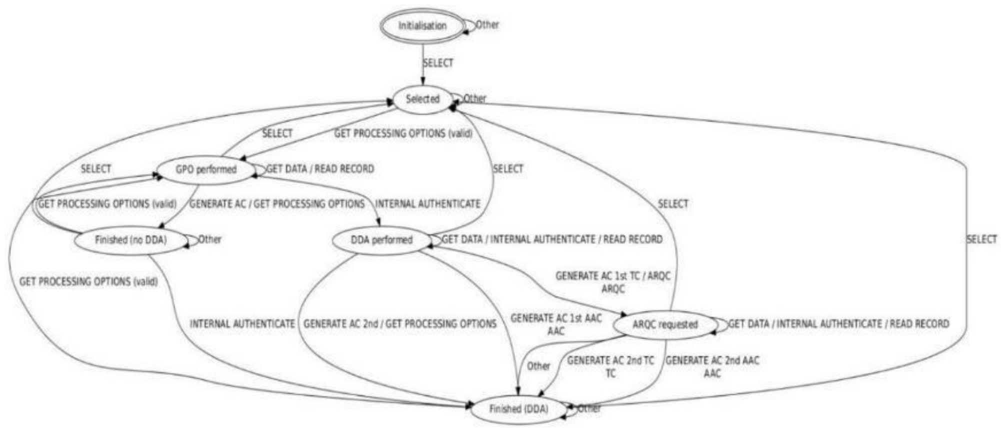


State Machine Learning

aka active learning
aka regular inference
aka automata learning
aka protocol state fuzzing
aka ....

Erik Poll
Radboud University Nijmegen

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

Stateless vs stateful systems

- **Stateless 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
- **Stateful 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

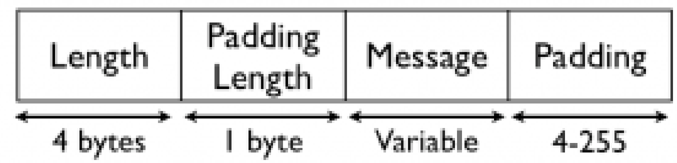
Which systems are harder to test (or fuzz): stateless or stateful systems?

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

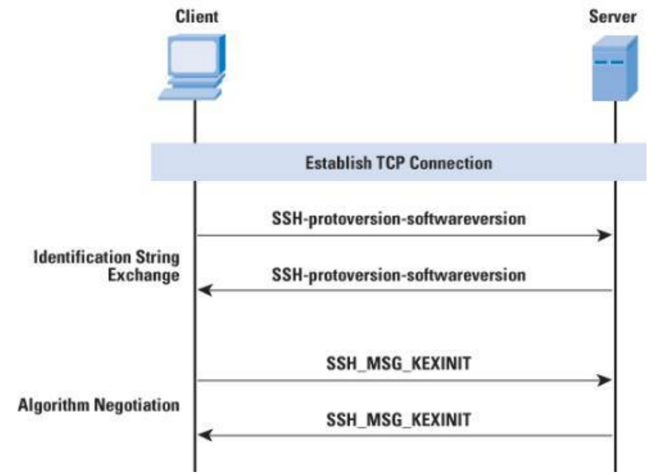
Protocols

Many procotols are stateful and then involve two levels of languages

1) **a language of input messages or packets**



2) **a notion of session, or *sequence* of messages**



Bugs can arise on both levels!

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

How can we test or fuzz these two levels?

For level 1 we can use fuzzing techniques 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 (steps 1–3)

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

key exchange algorithm negotiation (steps 4–5)

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 (steps 6–9)

10. …

session, incl. SSH authentication and connection protocols (step 10)

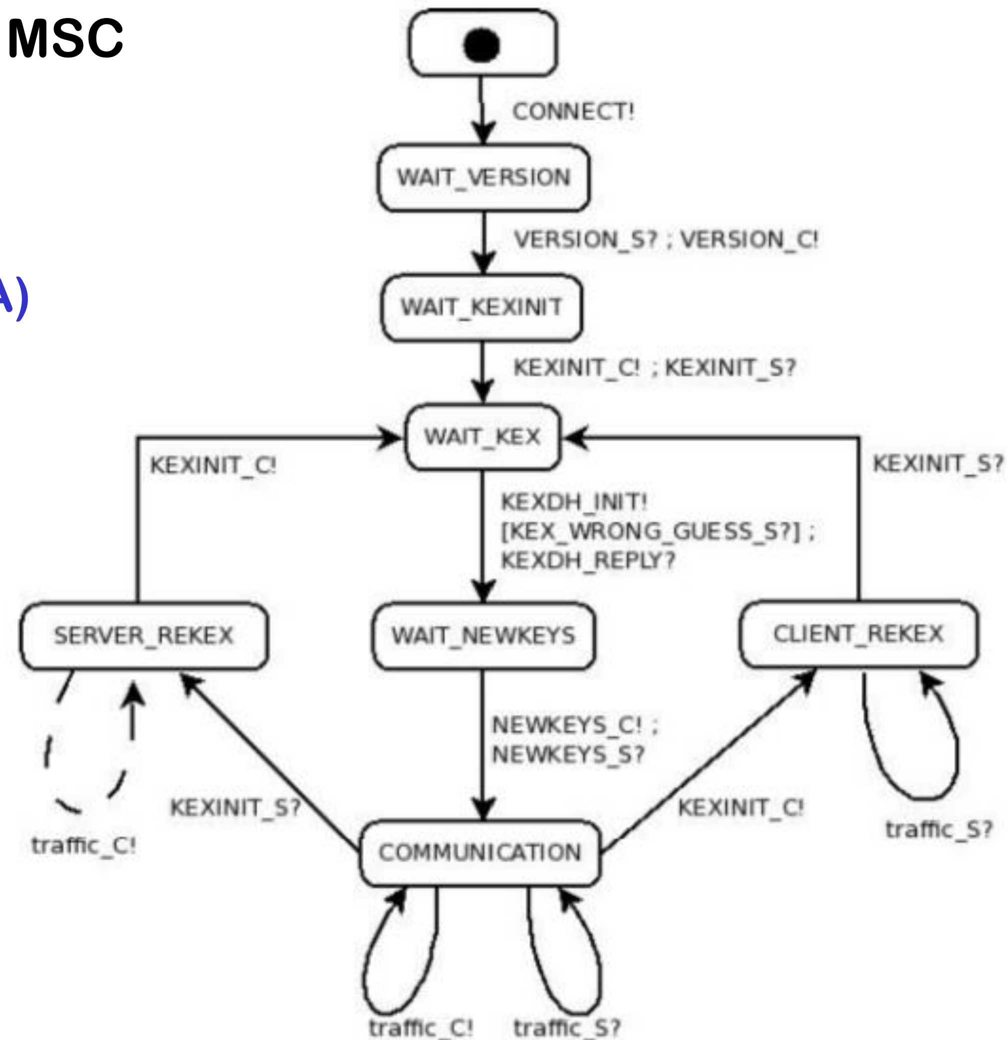
Typical protocol spec given as Message Sequence Chart or in Alice-Bob style.

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)



SSH transport layer

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

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

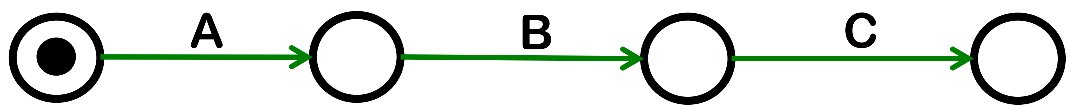
1) some error state,

2) back to the initial state, or

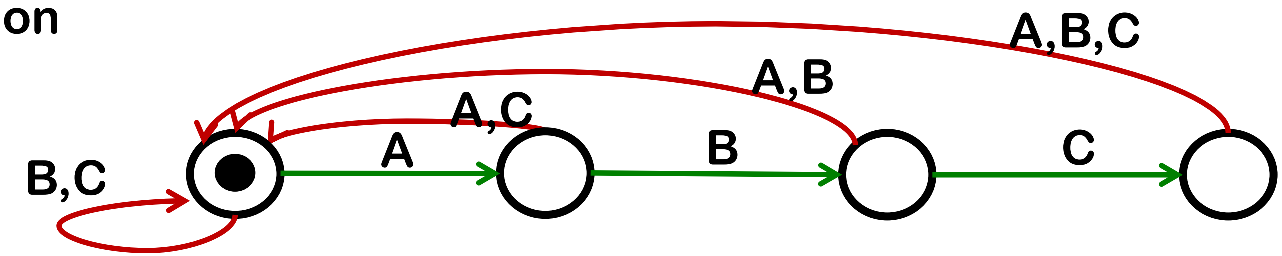
3) are ignored

input enabling

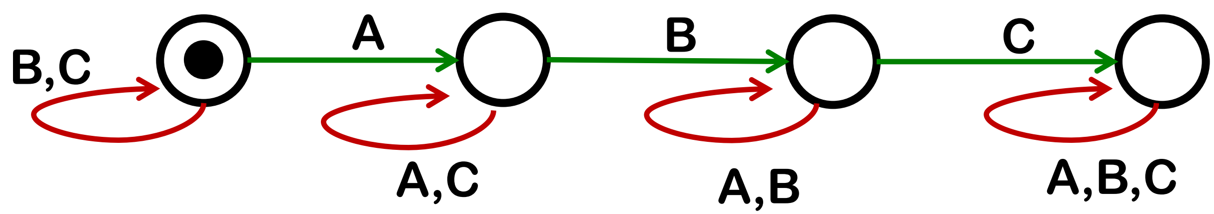
State machine that is not input-enabled



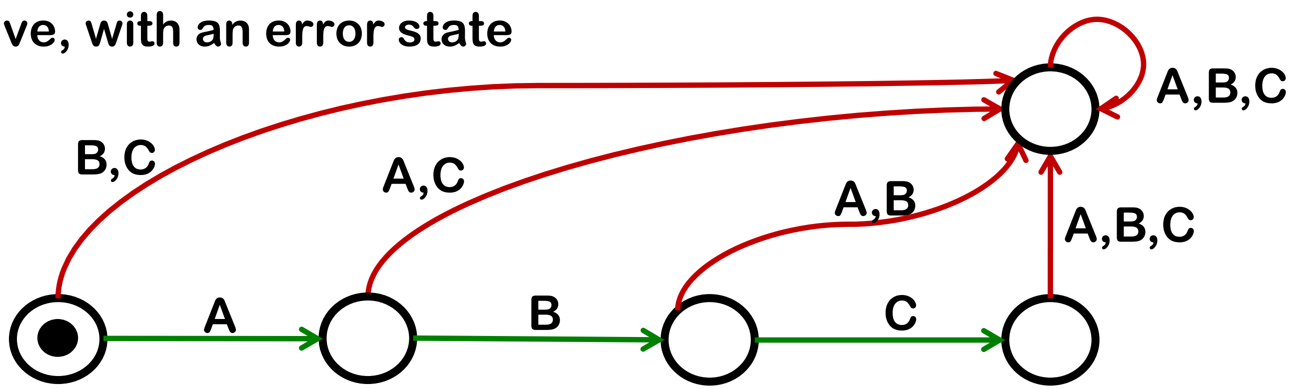
Input enabled version



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

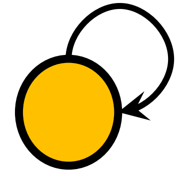
CVE-2018-10933

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 

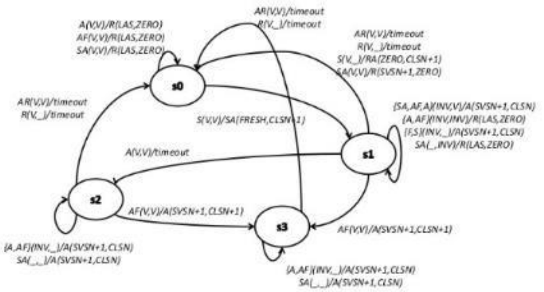
  [Verifying an implementation of SSH, WIST 2007]

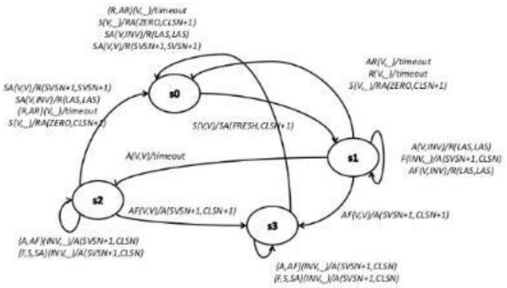
- **e.dentifier2**

  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

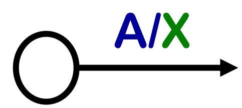
It is a great way to obtain protocol state machines

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

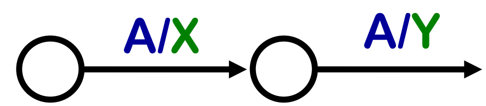
State machine inference, eg using LearnLib

Just try out many sequences of inputs, and observe outputs

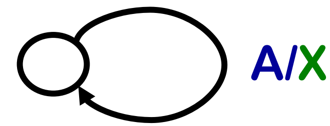
Suppose input A results in output X



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

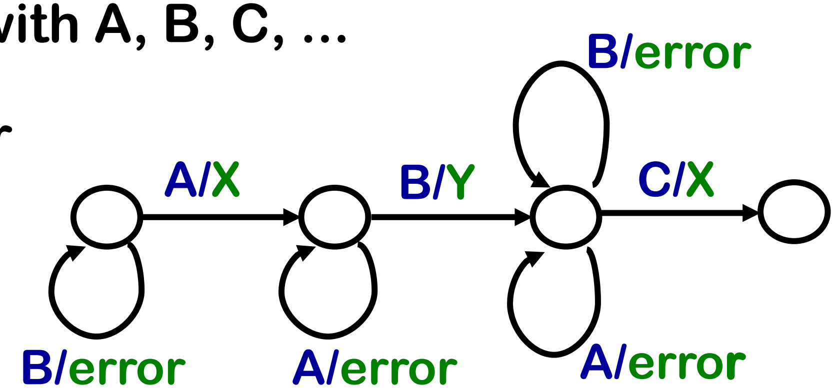


- If second input A results in the *same* output X



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

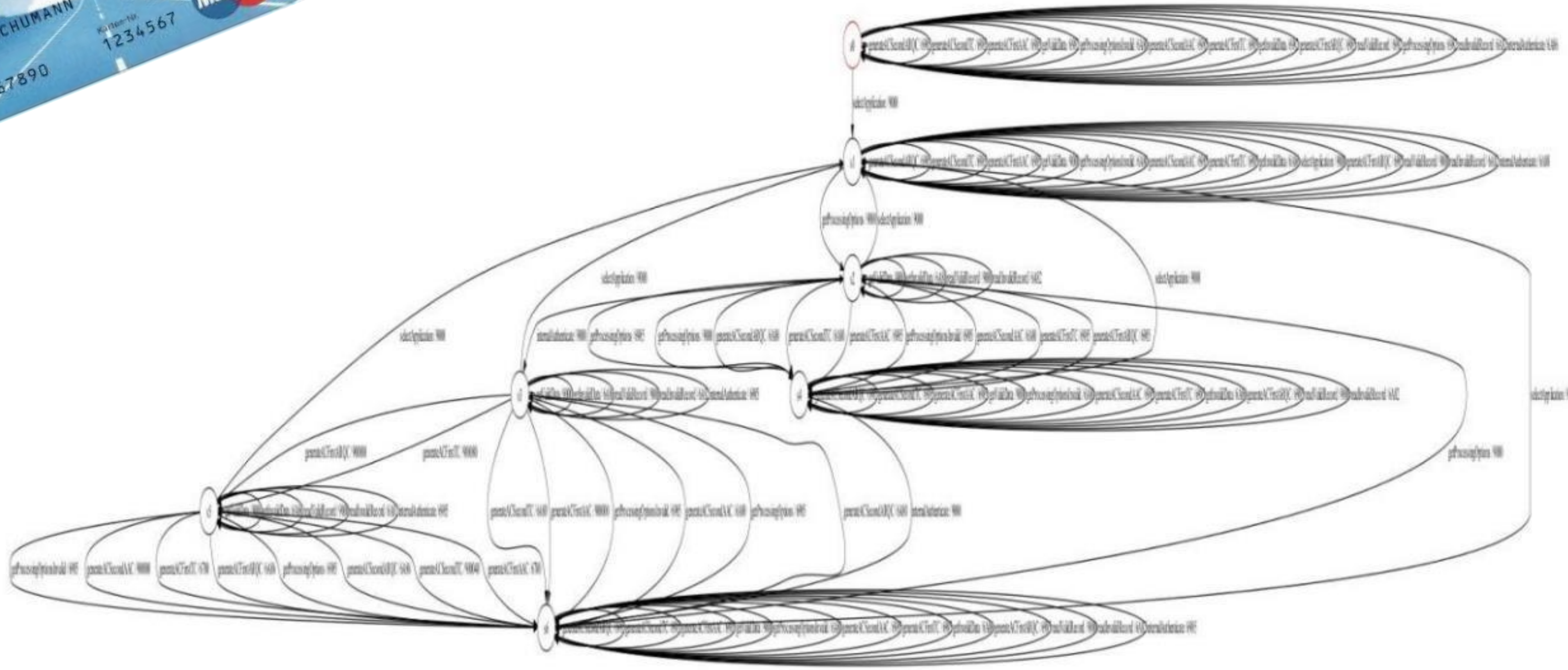
Case study 1: EMV

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

  with *lots* of variants

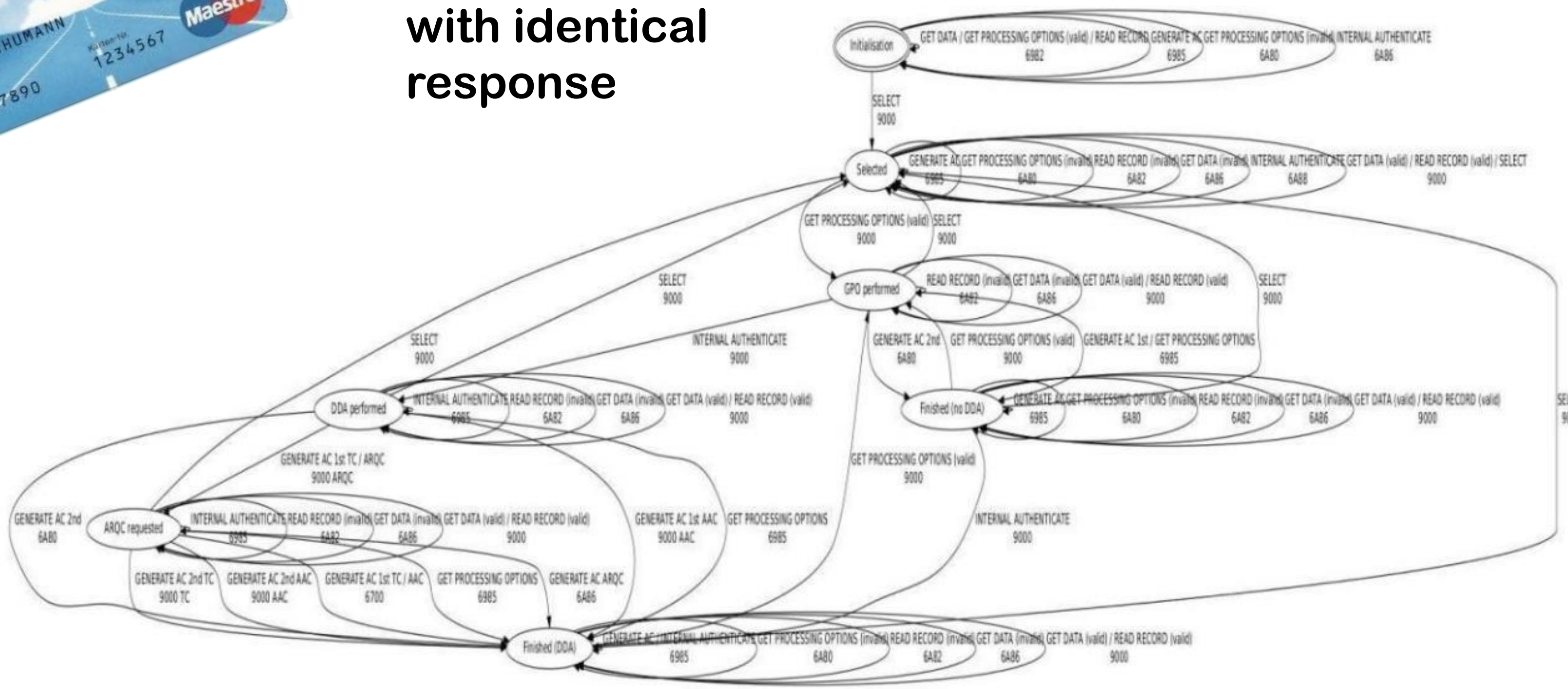
- **Specs controlled by EMVCo which is owned by MasterCard, VISA, American Express, JCB**
- **Specification in 4 books totalling > 700 pages**
- **EMV contactless specs: 10 more books, > 1500 pages**

State machine inference of Maestro card

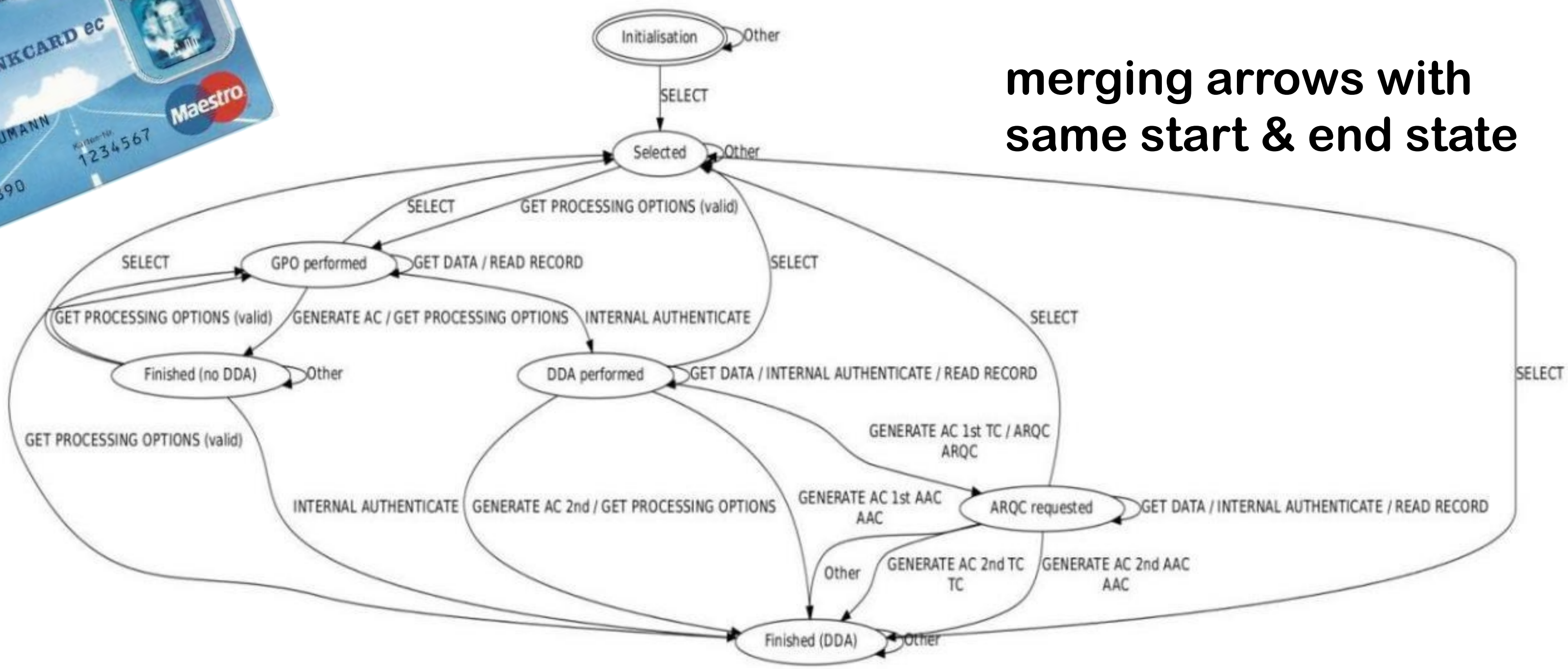


State machine inference of Maestro card

merging arrows
with identical
response



State machine inference of Maestro card



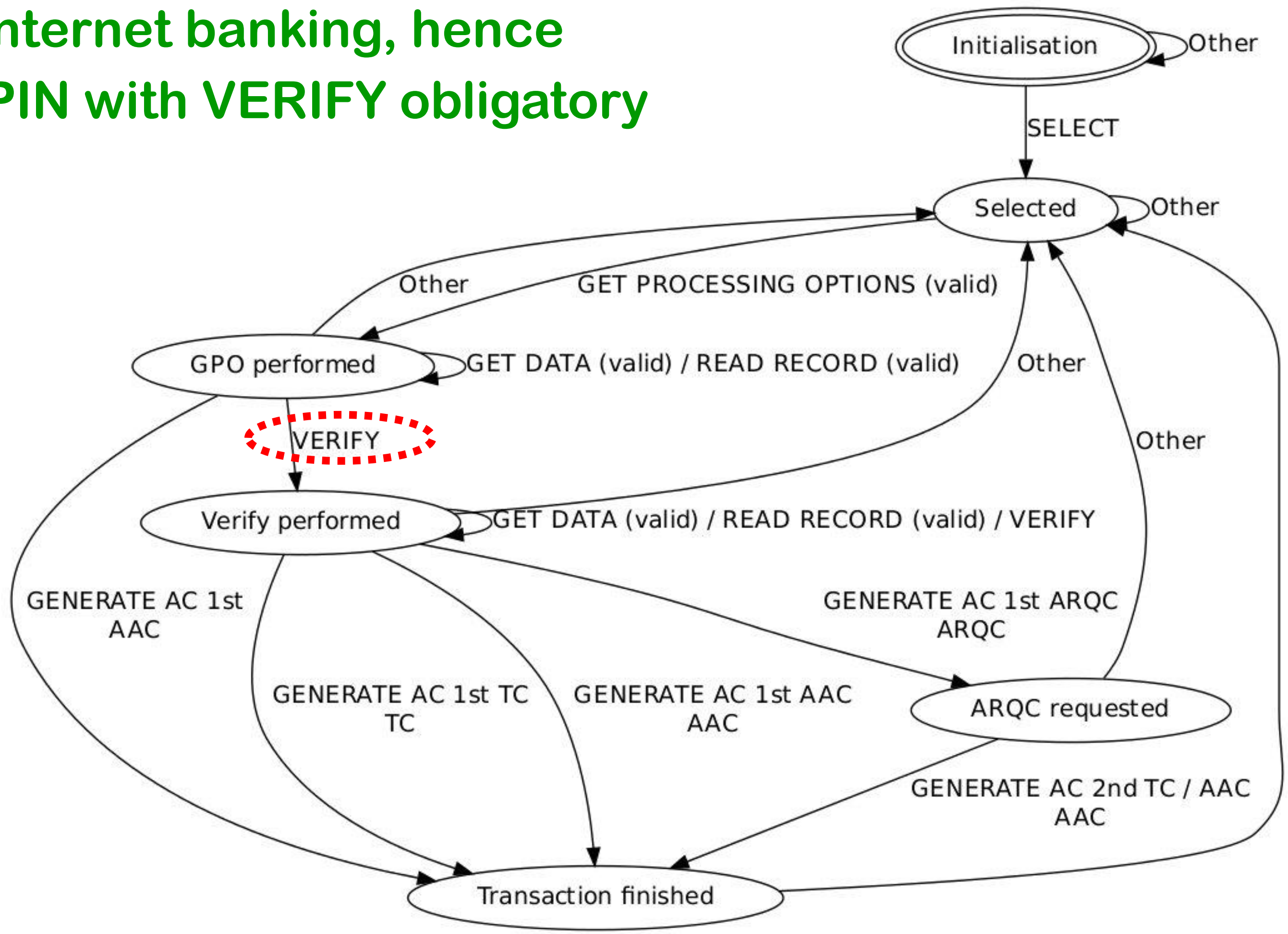
merging arrows with same start & end state

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

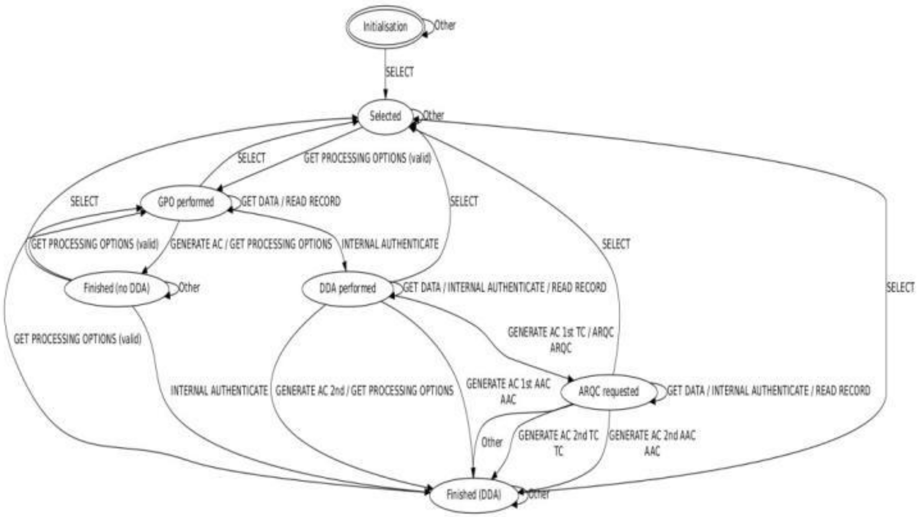
[Fides Aarts et al., Formal models of bank cards for free, SECTEST 2013]

SecureCode application on Rabobank card

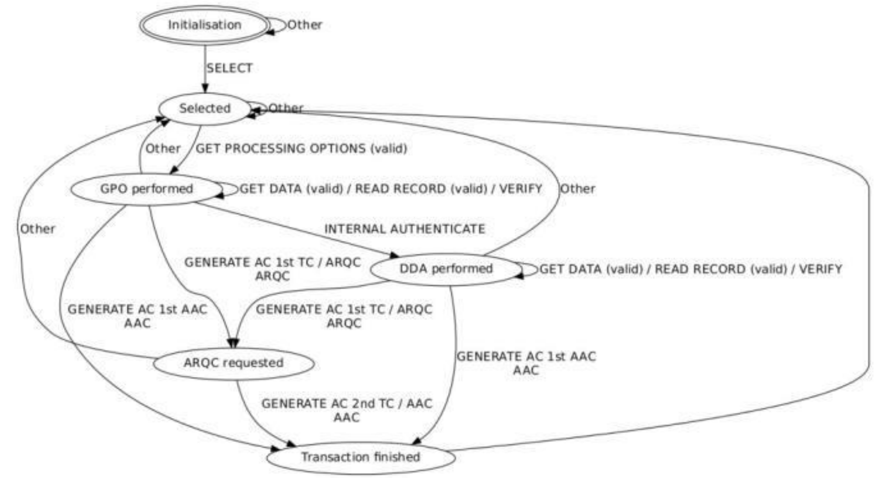
used for internet banking, hence entering PIN with VERIFY obligatory



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?

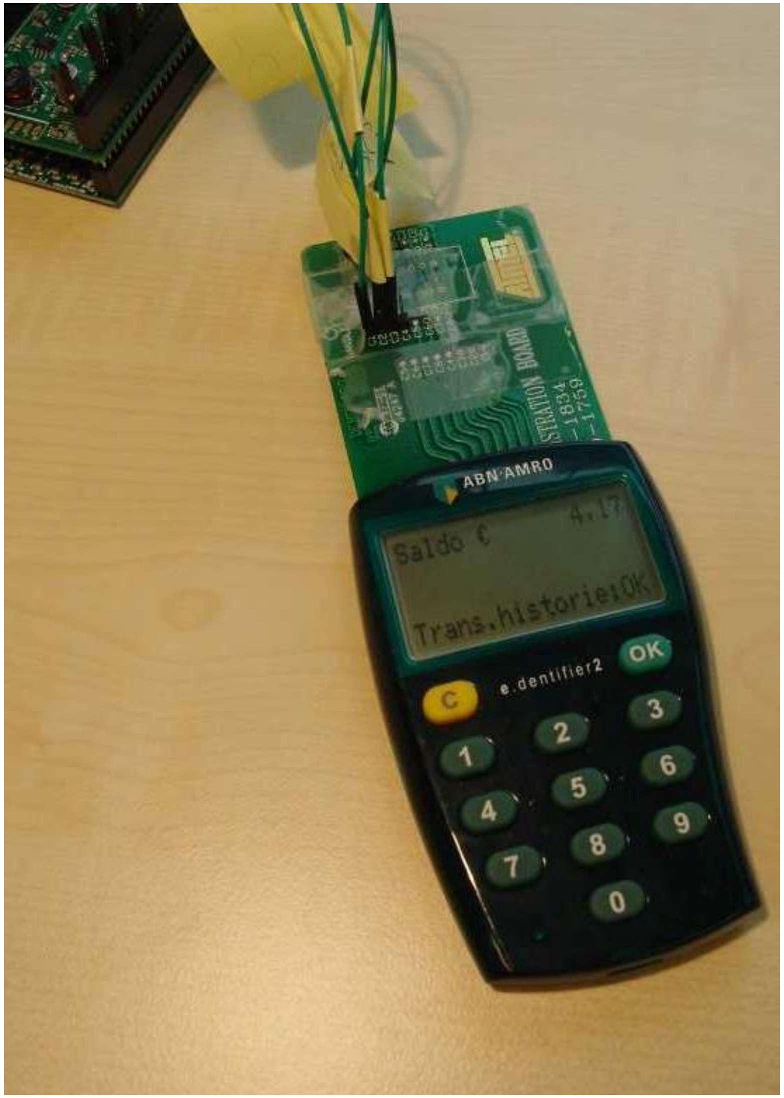
Case study 2: the USB-connected e.dentifier

Can we use state machine learning with

- **USB commands**
- **user actions via keyboard**

to obtain the state machine

of the ABN-AMRO e.dentifier2?

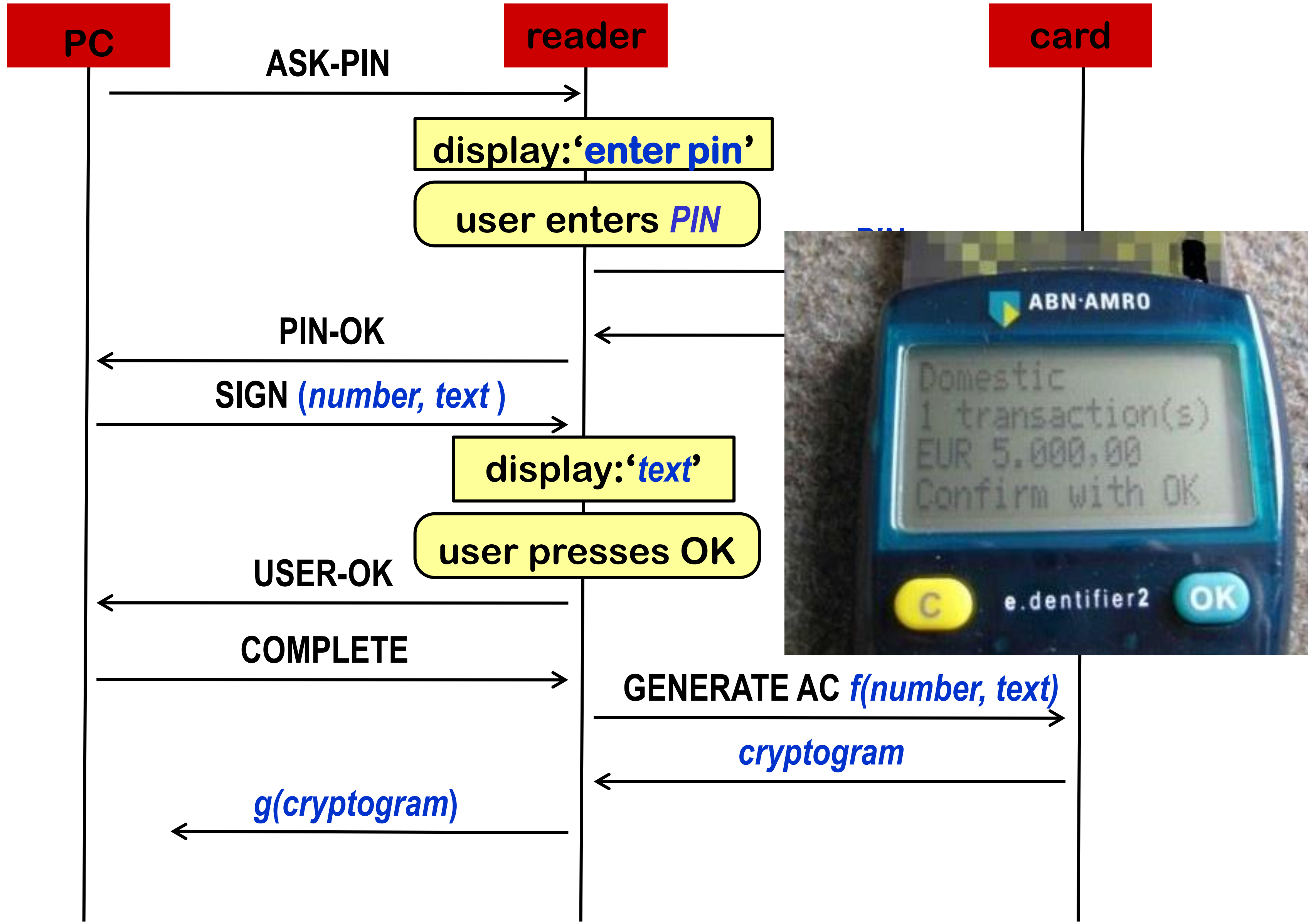


Earlier manual analysis

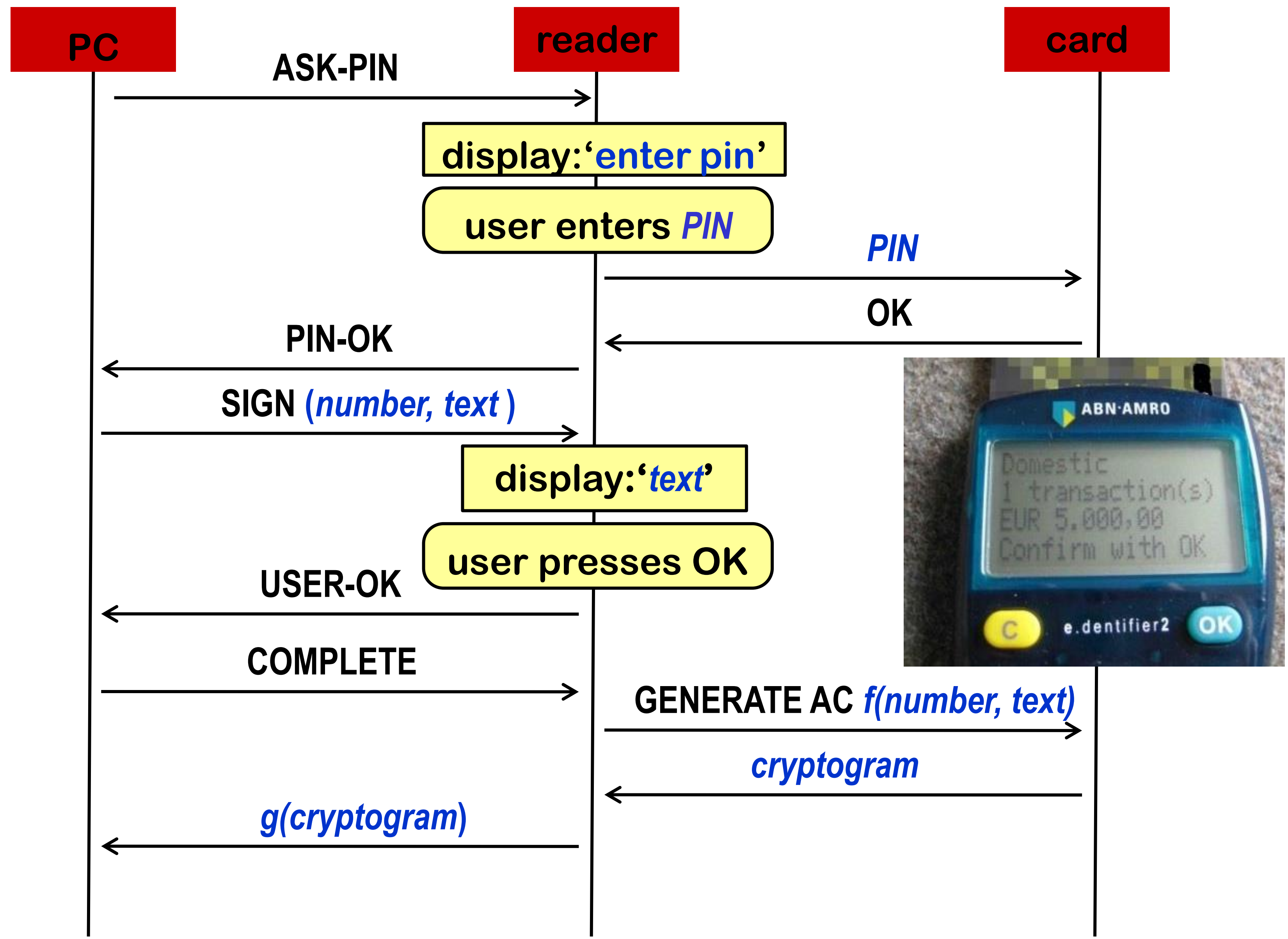
revealed the USB connection

has a flaw

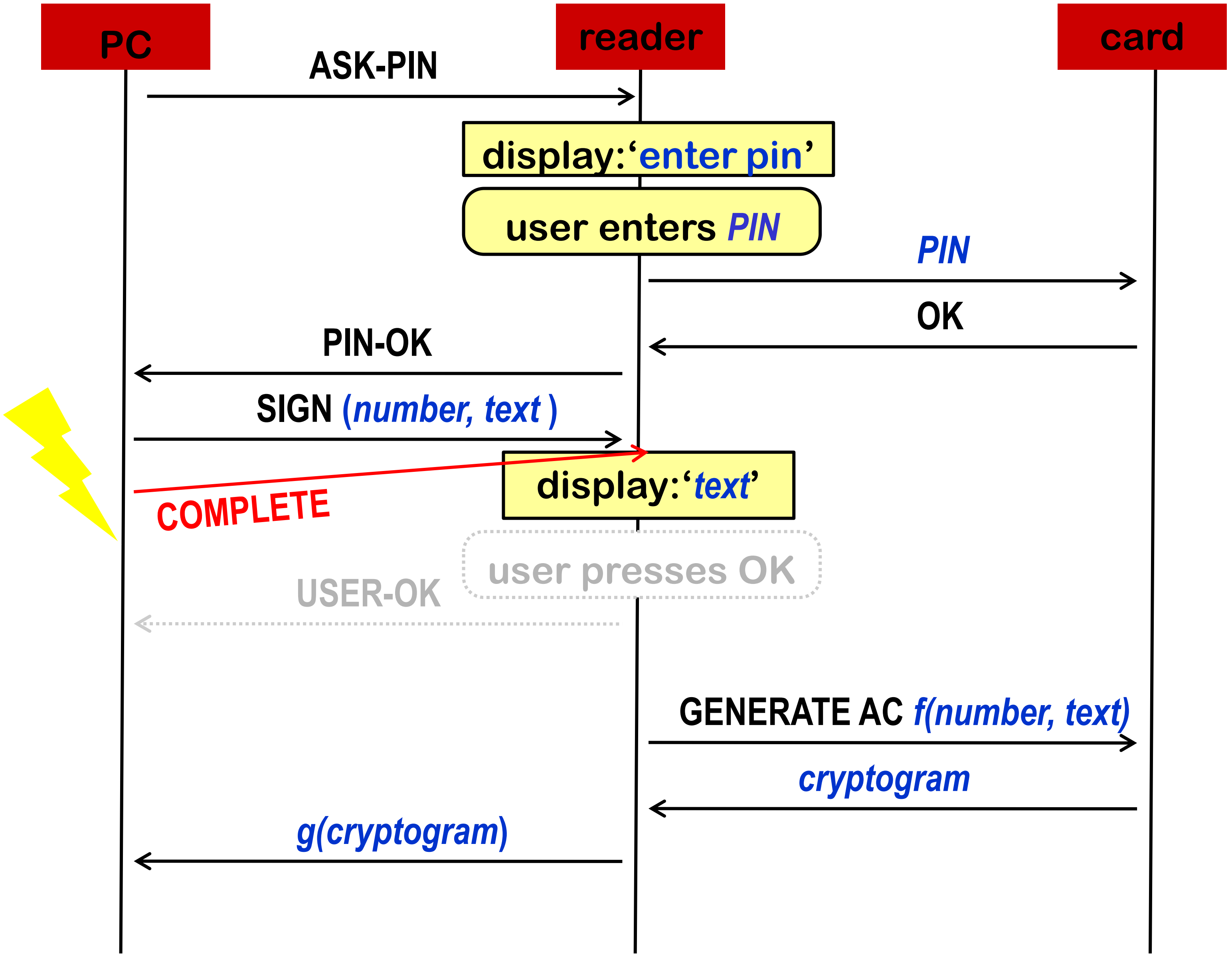
(Manually) reverse-engineered Protocol



Spot the defect!

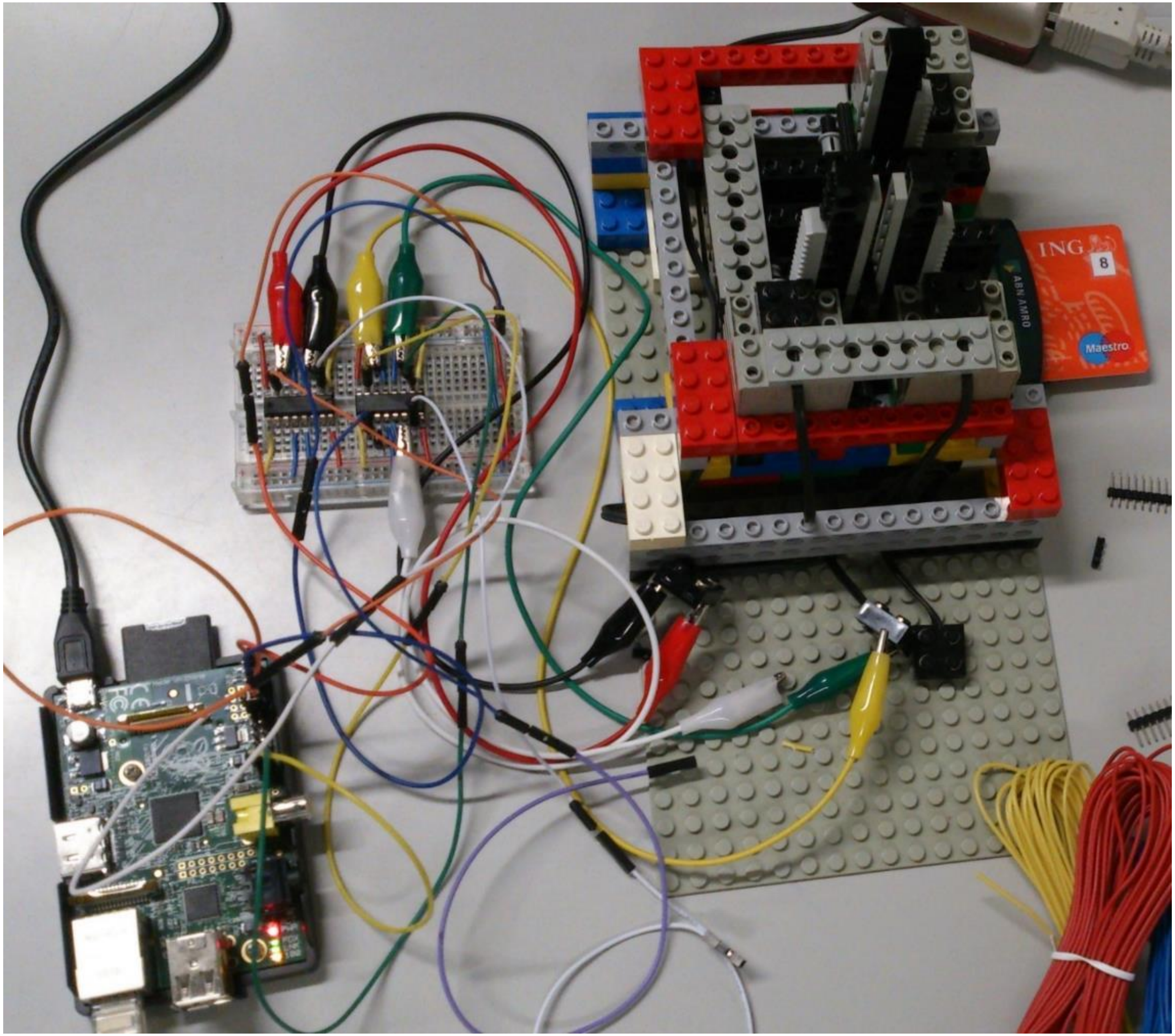


Attack!



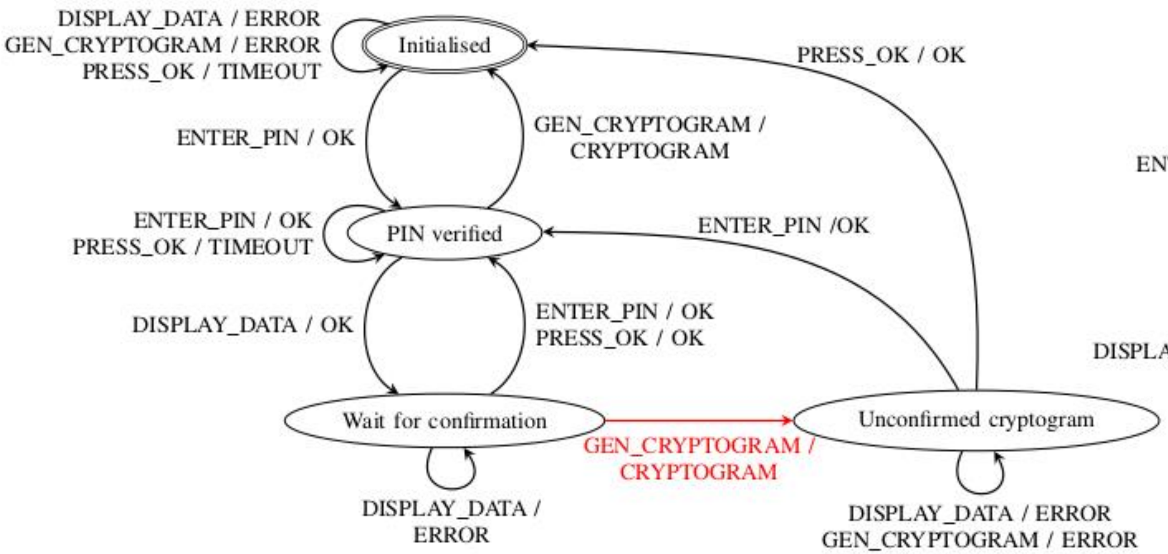
PC
reader
card
ASK-PIN
display:'enter pin'
user enters PIN
PIN
OK
PIN-OK
SIGN (number, text)
COMPLETE
display:'text'
user presses OK
USER-OK
GENERATE AC f(number, text)
cryptogram
g(cryptogram)

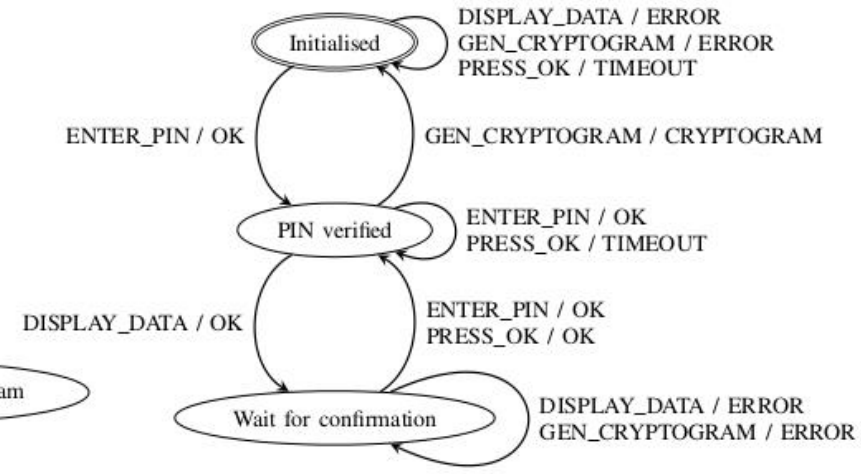
Operating the keyboard using LEGO

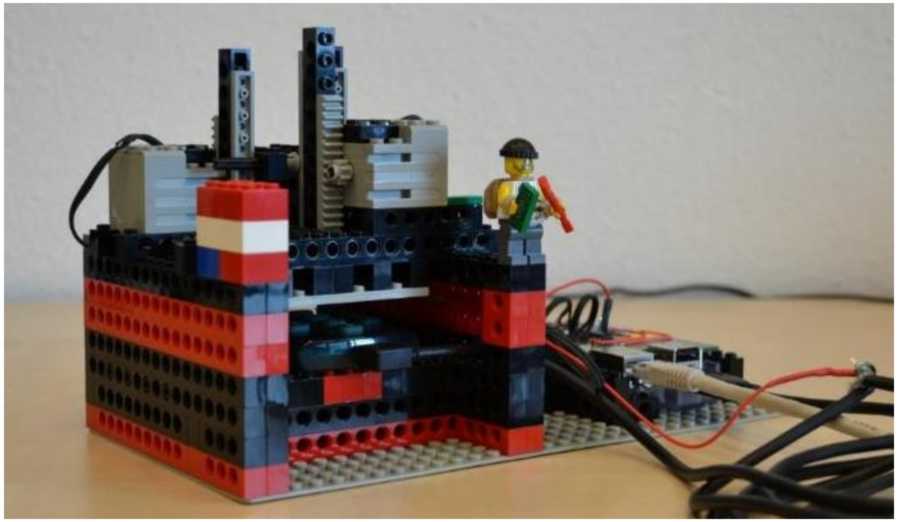

ING
8
ABN AMRO
Maestro

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

State machines of old vs new e.dentifier2

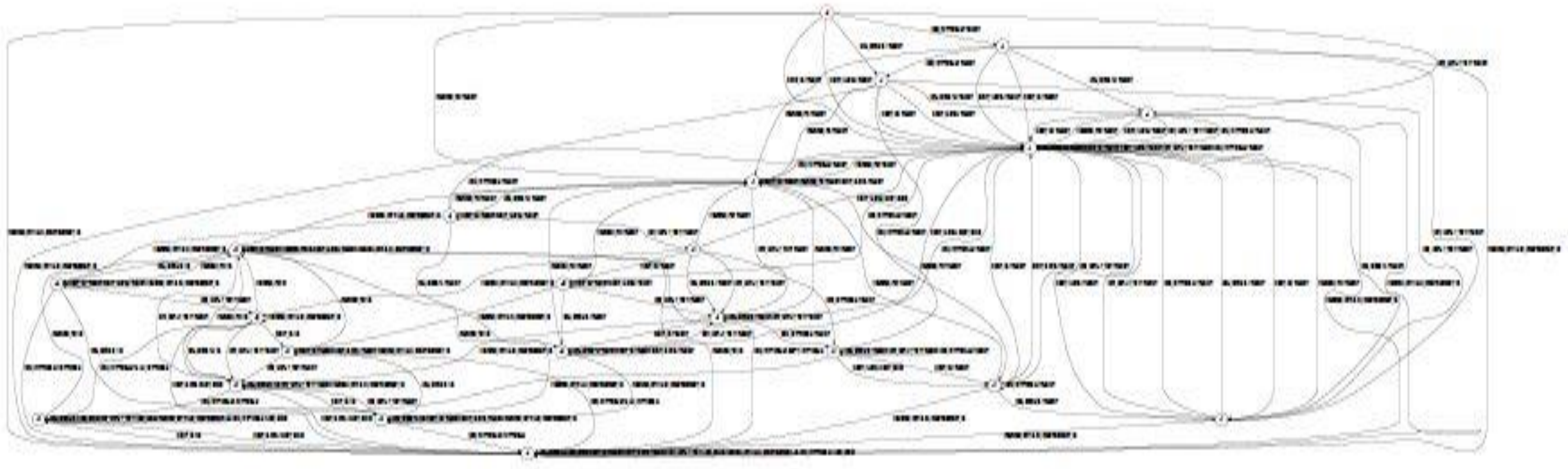






https://www.youtube.com/watch?v=hyQubPvAyq4

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?

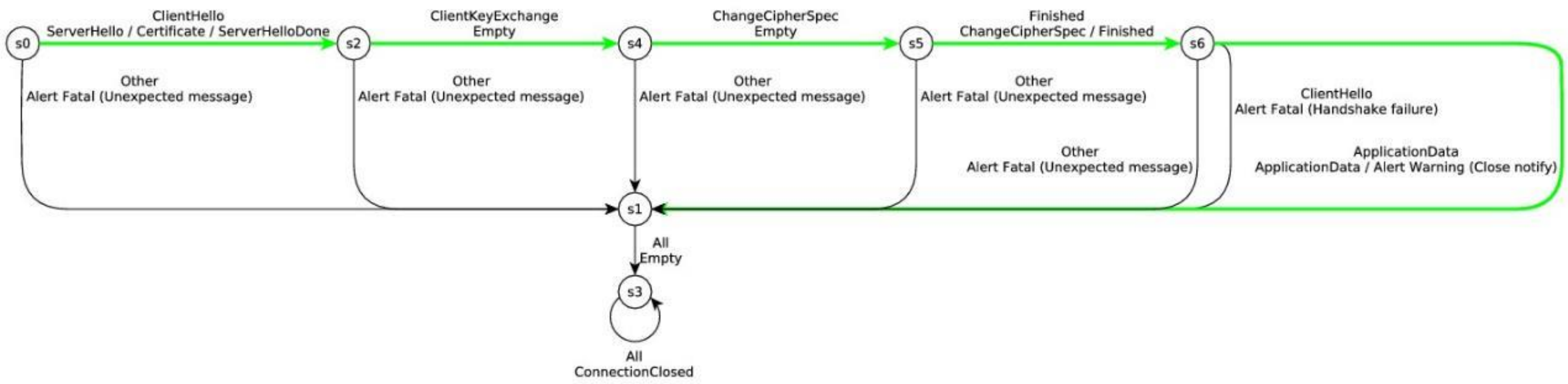
Results with learning state machines for e.dentifier2

- **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

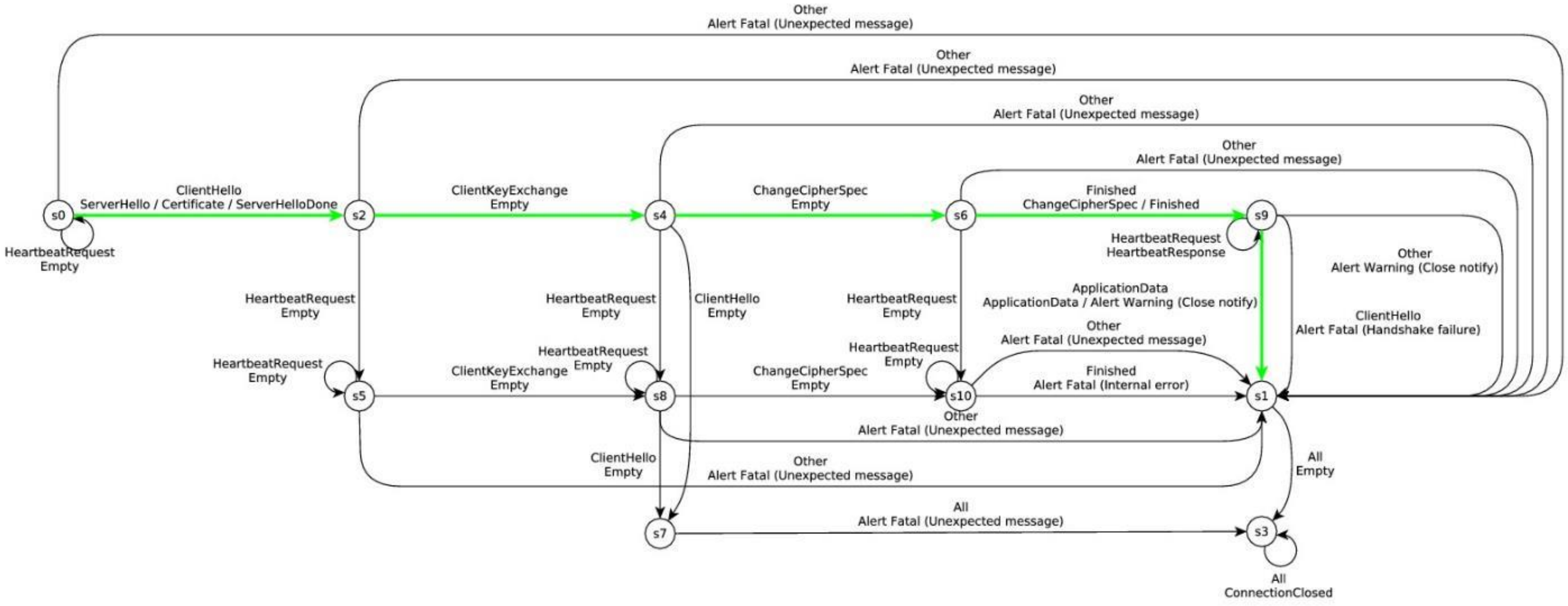
Case study 3: TLS



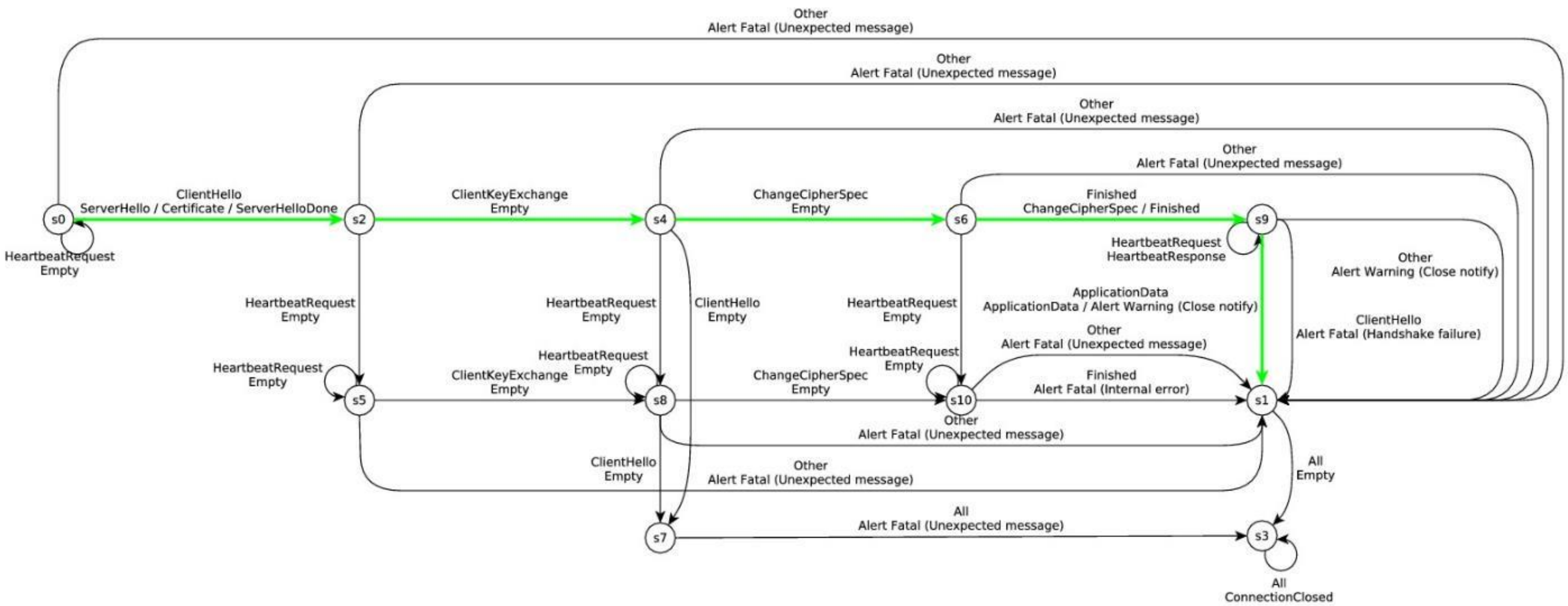
State machine inferred from NSS implementation

Comforting to see this is so simple!

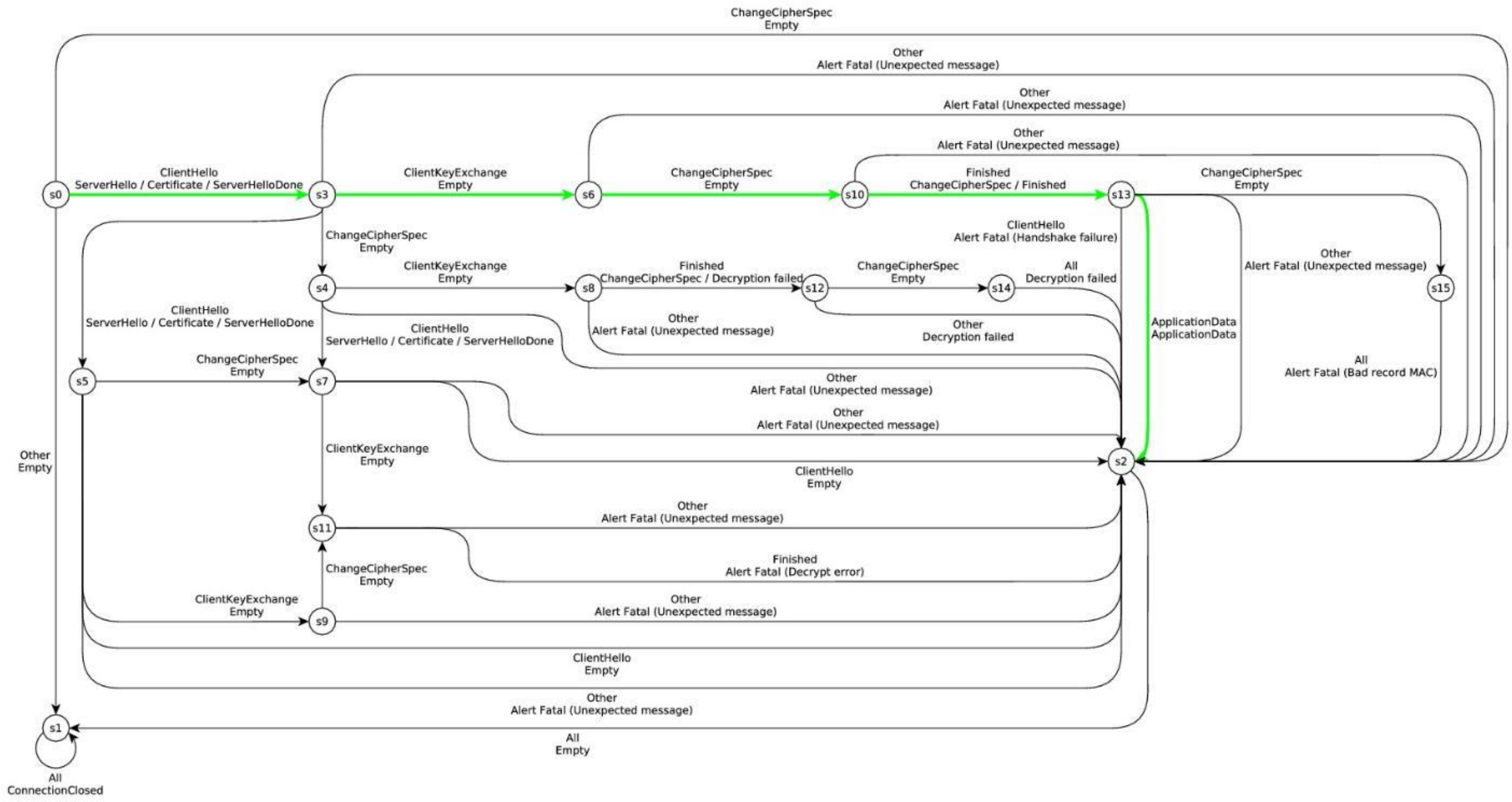
TLS... according to GnuTLS



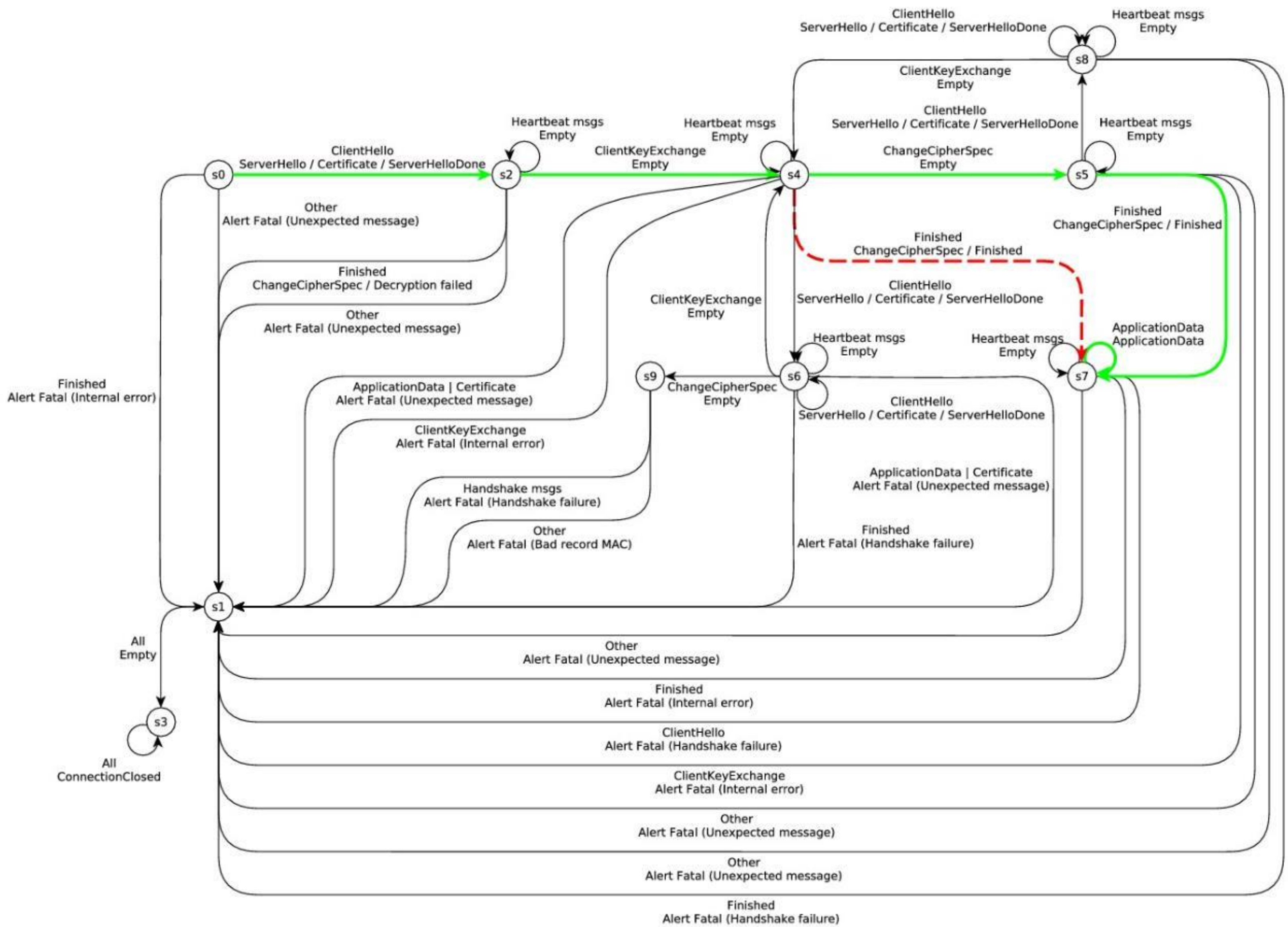
TLS... according to GnuTLS



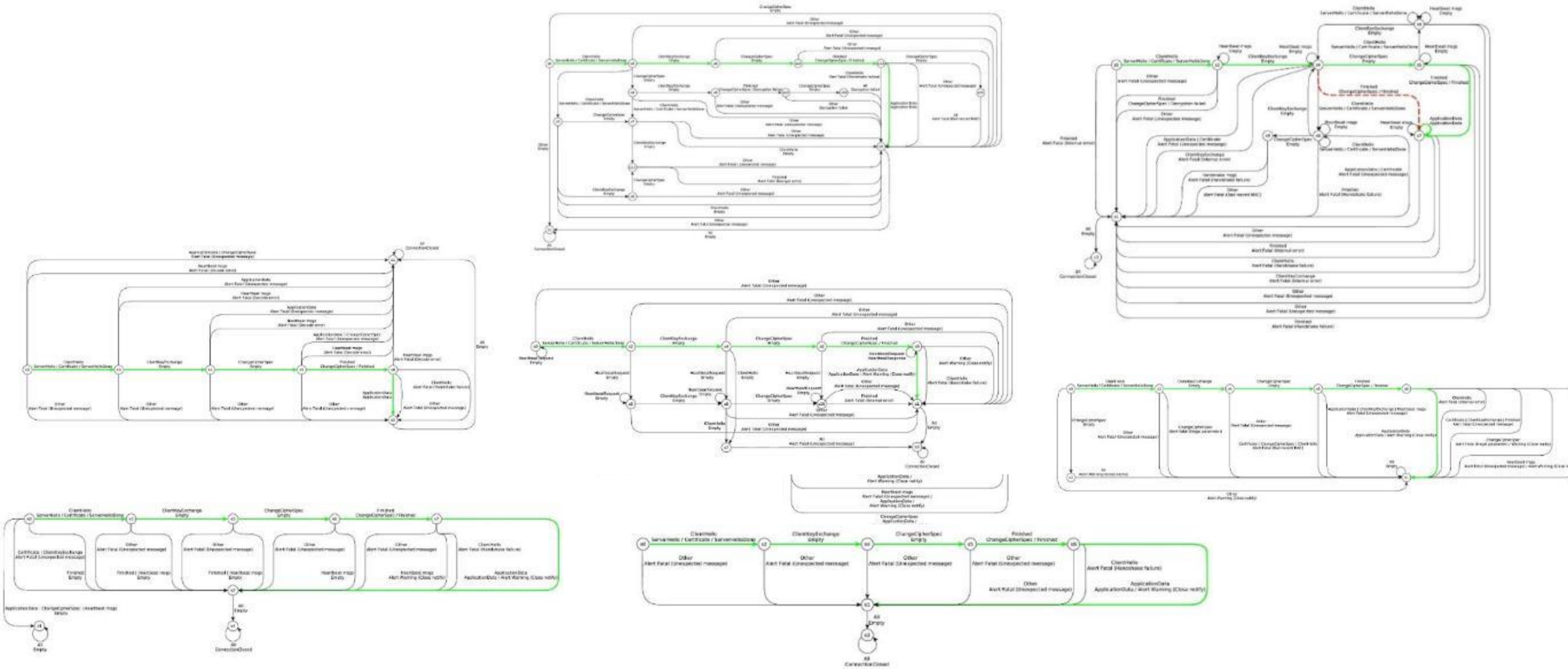
TLS... according to OpenSSL



TLS... according to Java Secure Socket Exension



Which TLS implementations are correct? or secure?



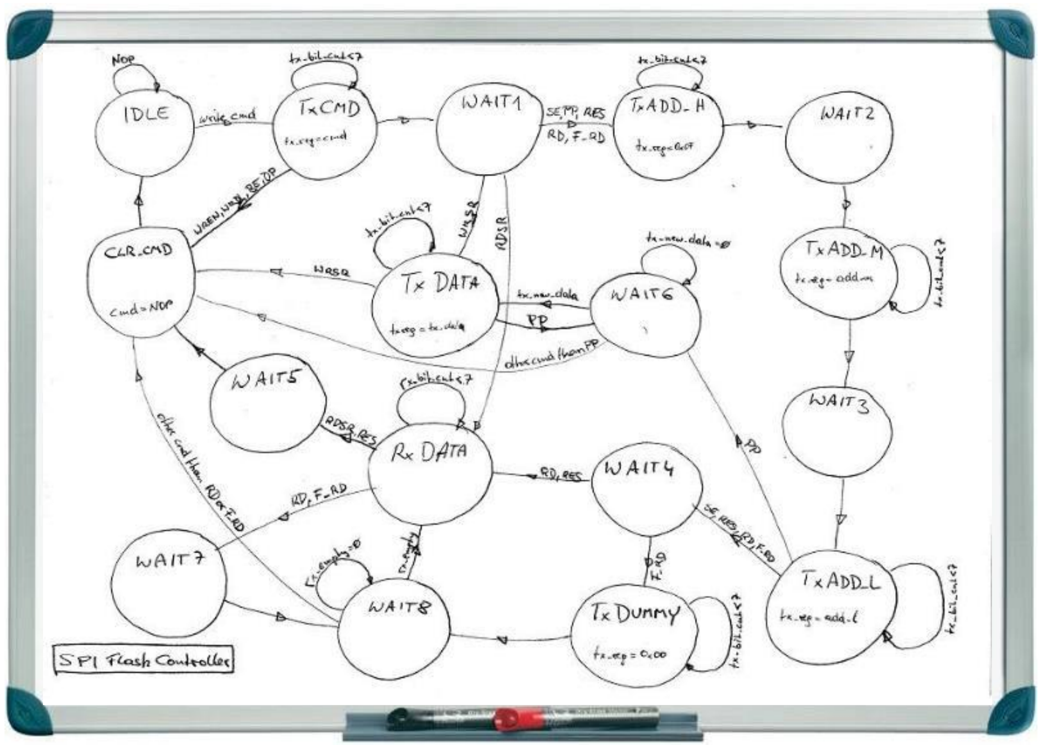
[Joeri de Ruiter et al., Protocol state fuzzing of TLS implementations, Usenix Security 2015]

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?



Protocol state machines

Rigorous & clear specifications using protocol state machines can improve security:

- by avoiding ambiguities
- useful for programmer

In spec does not clearly specify a state machines, extracting state machines from code using state machine learning 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

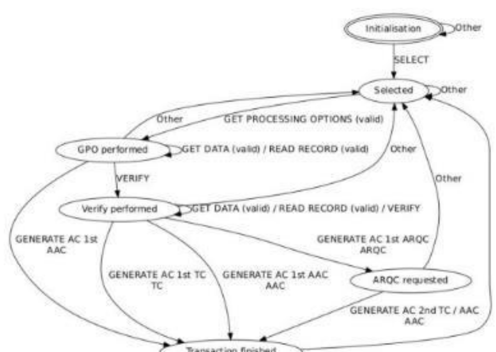
The road we followed

specs

implementing

code

state machine learning



model

Ideally specs would include a state machine!

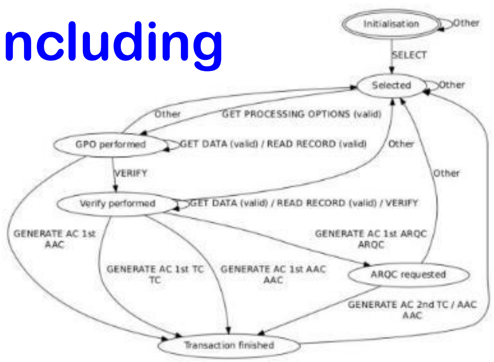
specs

implementing

model-based testing

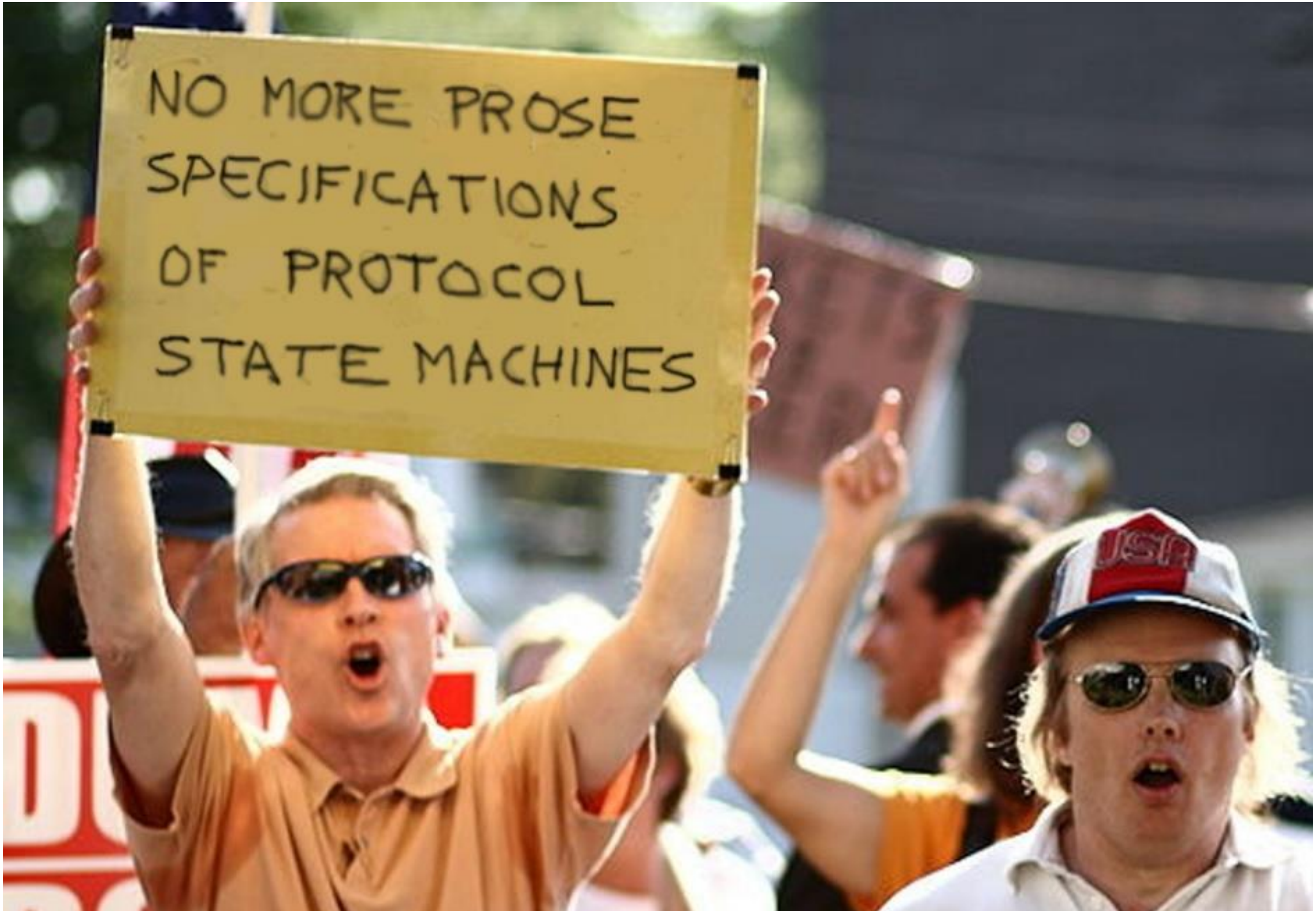
code

including



model

Or maybe we could generate code?


NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES