State machine learning

or

Formal models for free!

Erik Poll

joint work with Joeri de Ruiter & many others

Motivation

Security looks like ideal application area for formal verification.

Most security problems due to software (not crypto!)

Can we specify interesting security properties to verify?

What to specify for security? ☹

- functional specifications? *full* functional correctness?
  - Often hard/impossible to write

    eg, how would you specify a web-browser, or an internet banking app?

  - Security is often not about what a program should do, but about what a program should *not* do

    eg: “this password manager should not leak keys”

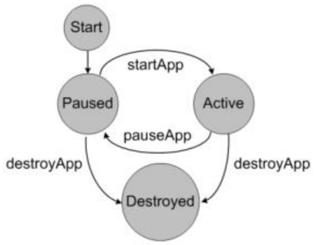
  - Possible good news: maybe security properties can be independent of functionality?

What to specify for security? ☺

- no (uncaught) runtime exceptions
  - simple to specify, independent of functionality
  - rules out some Denial-of-Service (DoS) attacks
- invariants on data (object invariants)
- information flow properties
  - recall Werner's talk yesterday
- temporal properties

  eg "X can only happen after entering the PIN code"

  or state machine behaviour



Focus of this talk

Note: the categories above concern different aspects of behaviour

Case study: SSH

High-level formal spec of SSH

1. $C \rightarrow S$: CONNECT
2. $S \rightarrow C$: VERSION_S server version string
3. $C \rightarrow S$: VERSION_C client version string
4. $S \rightarrow C$: SSH_MSG_KEXINIT I_C
5. $C \rightarrow S$: SSH_MSG_KEXINIT I_S
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
10. ...

Nice specification, and can be formally verified (eg using ProVerif)

But it *oversimplifies* - it only specifies *one correct, happy flow*

More detailed info in the RFCs (lots of it!)

"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"

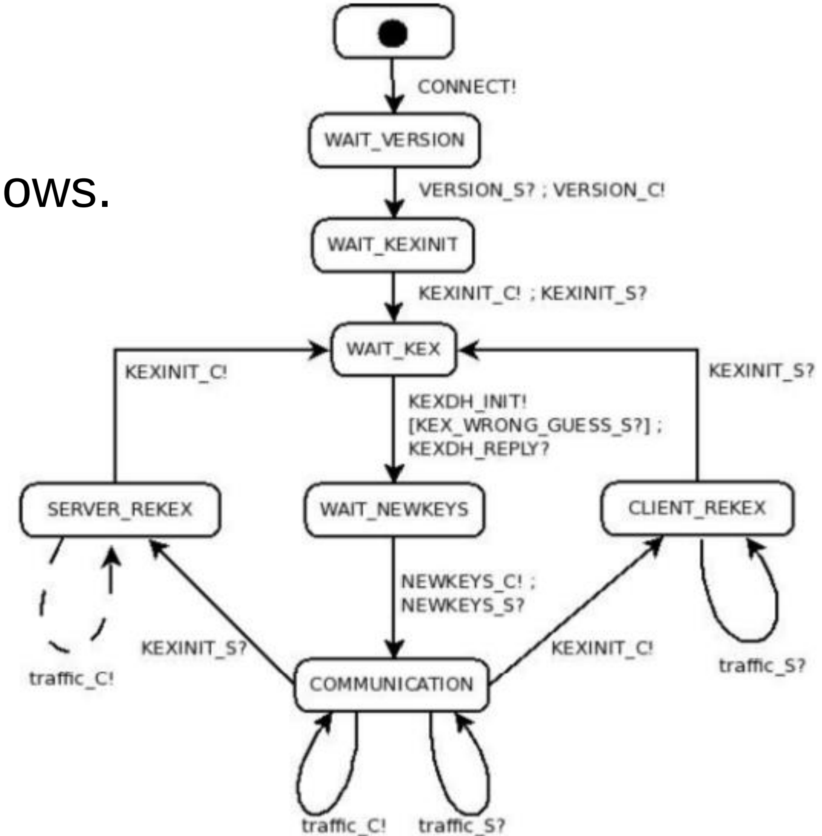
...

"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."

More detailed formal spec: using state machine

More complete,

because it includes several happy flows.



But it *still oversimplifies*:

an implementation will have

to be *input-enabled,*

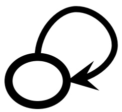
ie in every state every message

may be received

Typical response to unexpected messages: ignore or abort

Implementations can get this wrong...

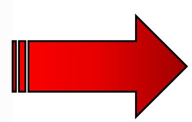
- Protocol state machine of SSH implementation we were verifying:

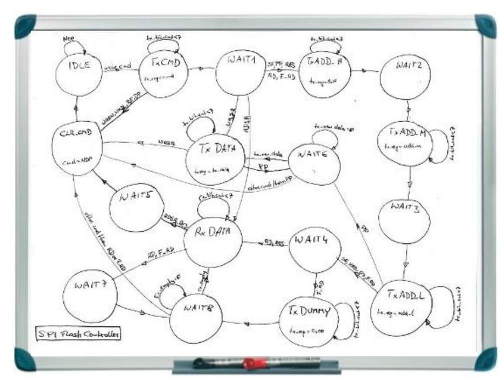


This works fine, but is not secure...

[Erik Poll and Aleksy Schubert, Verifying an implementation of SSH,WITS'07]

- *It is annoying that specs usually don't include state diagrams!*






NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES

State machine learning

Extracting state machines from implementation

Given a test harness that sends typical protocol messages

we can infer a finite state machine by black box testing

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

This is a great way to obtain a protocol state machine

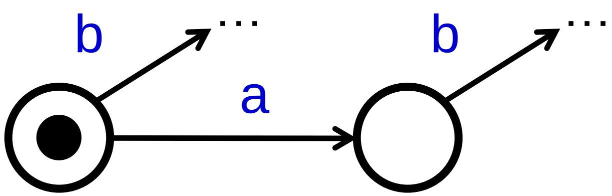
- without reading specs!
- without reading code!

State machine learning with L*

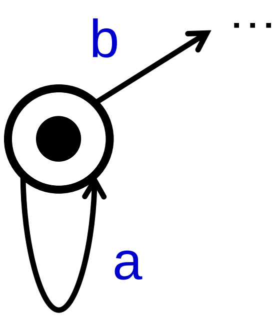
Basic idea: compare response of a deterministic system to different input sequences, eg.

1. b
2. a ; b

If response to b is different, then



otherwise



.

State machine learning

- The inferred state machine is only an approximation.

  There may be paths & states you don't find, due to
  - limits in the test harness
  - limits in the length of longest test runs

- So you can find flaws in program logic, but not a well-hidden backdoor...

- State machine learning involves a form of model-based testing
- It can be seen as a form of fuzz testing aka fuzzing

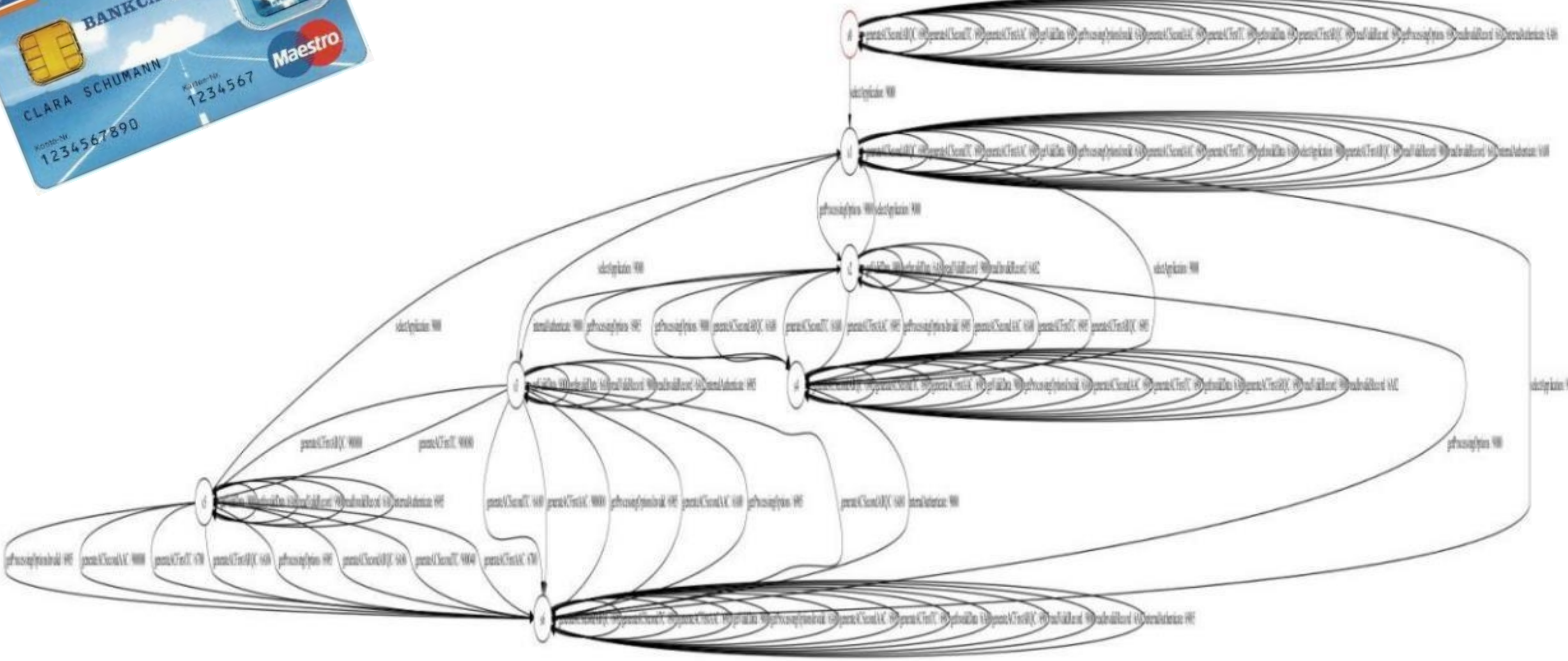
Case study: EMV

EMV

The standard for smartcards used for banking

- started 1993 by EuroPay, MasterCard, Visa
- Specs controlled by EMVCo which is owned by
- Specs defines a set of protocols with *lots* of variants
- Specification in 4 books totalling > 700 pages

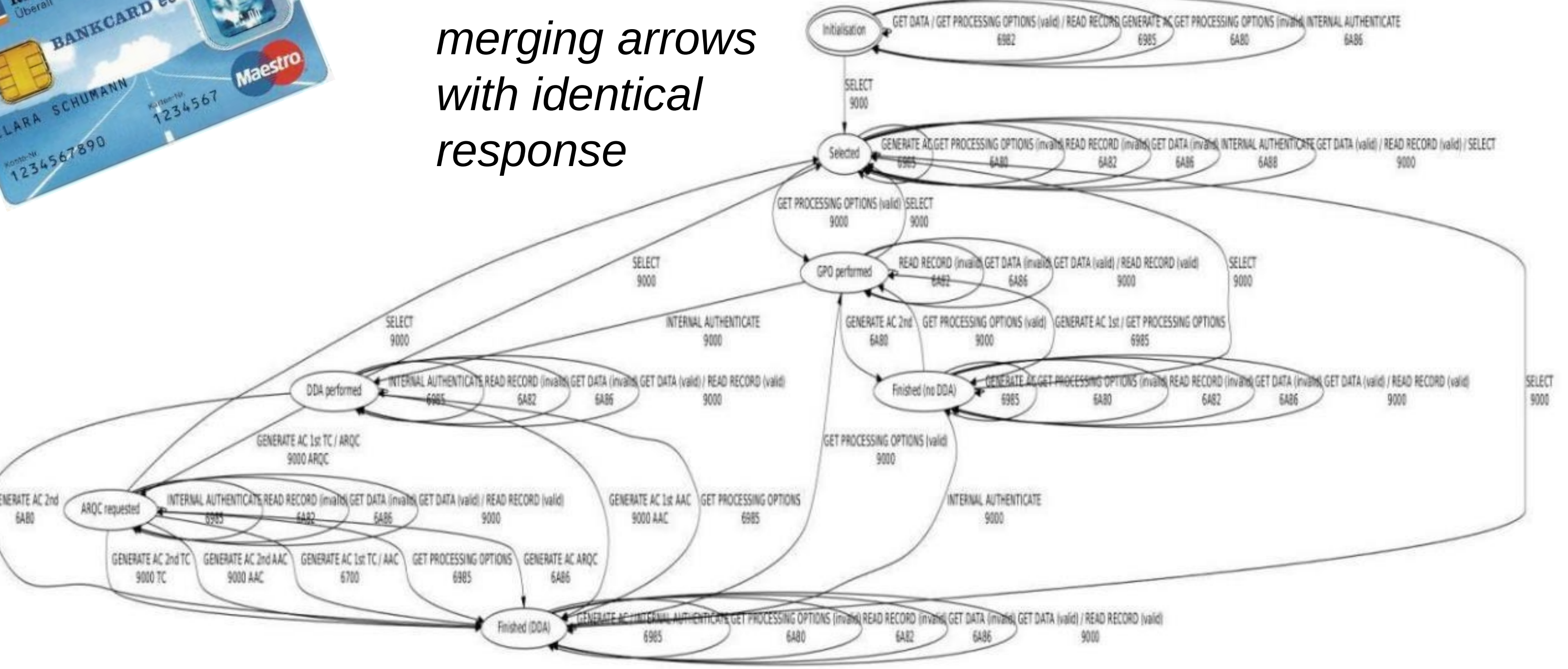
State machine learning of Maestro card



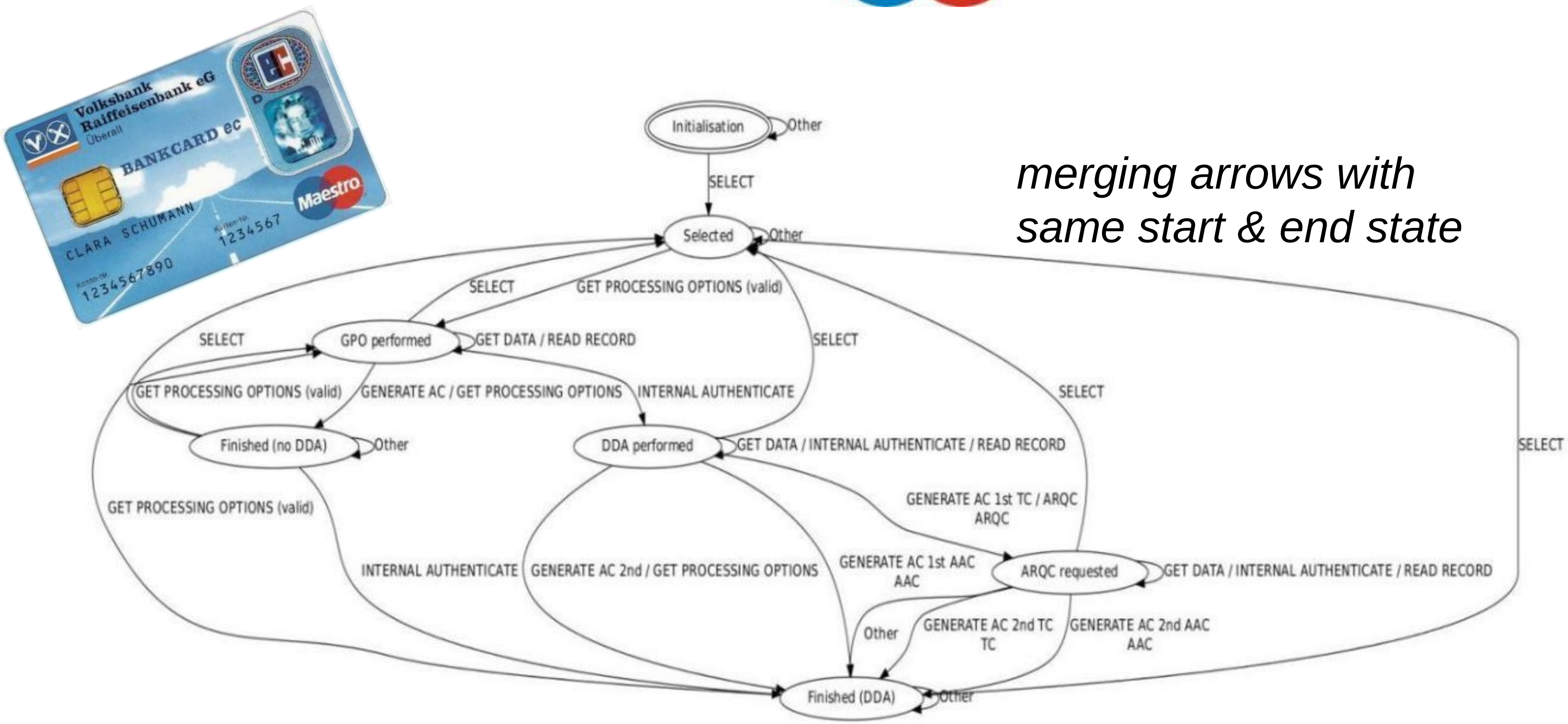
Result obtained after 10-20 minutes testing, of a dozen standard messages.

State machine learning of Maestro card

merging arrows with identical response



State machine learning of Maestro card

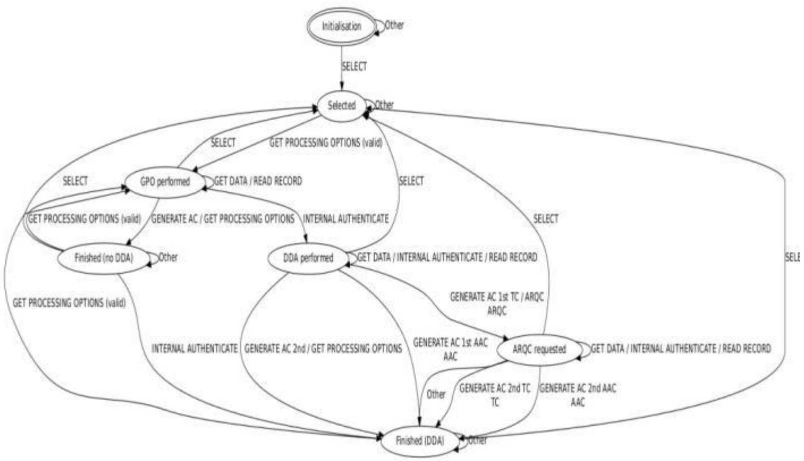


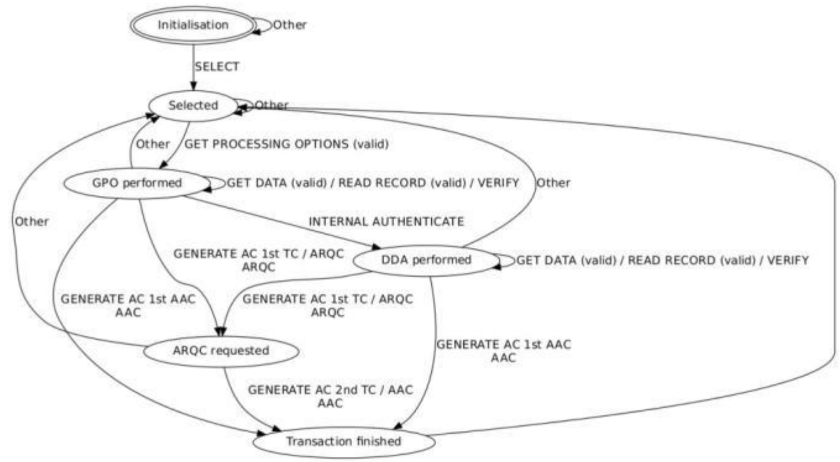
merging arrows with same start & end state

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

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

Using state machines for comparison





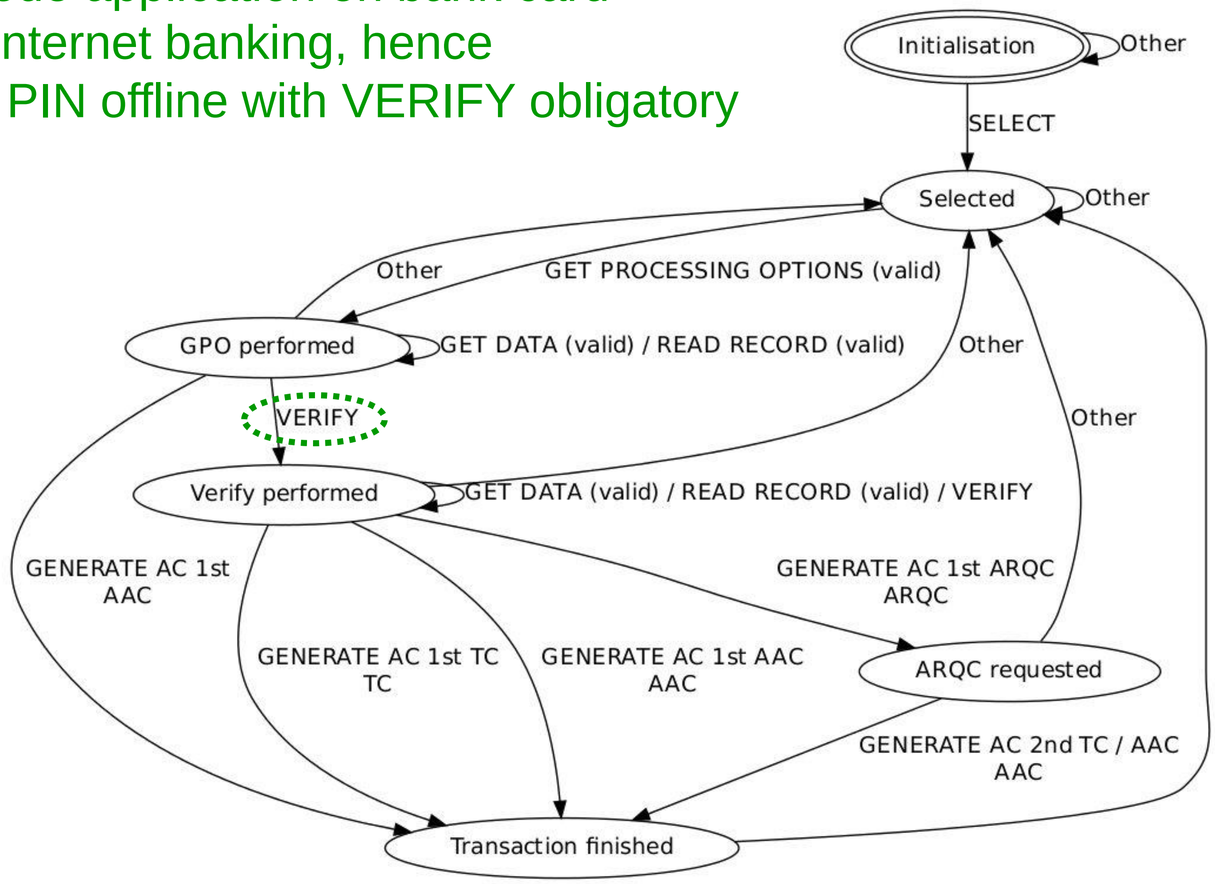
Volksbank Maestro implementation

Rabobank Maestro implementation

Are both implementations correct & secure? Or compatible?

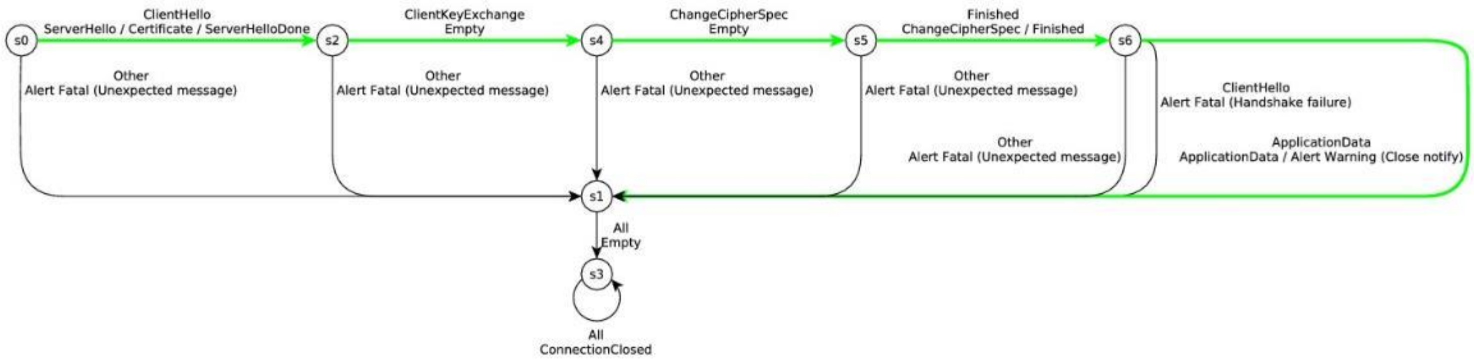
Using state machines for analysis

SecureCode application on bank card
used for internet banking, hence
checking PIN offline with VERIFY obligatory



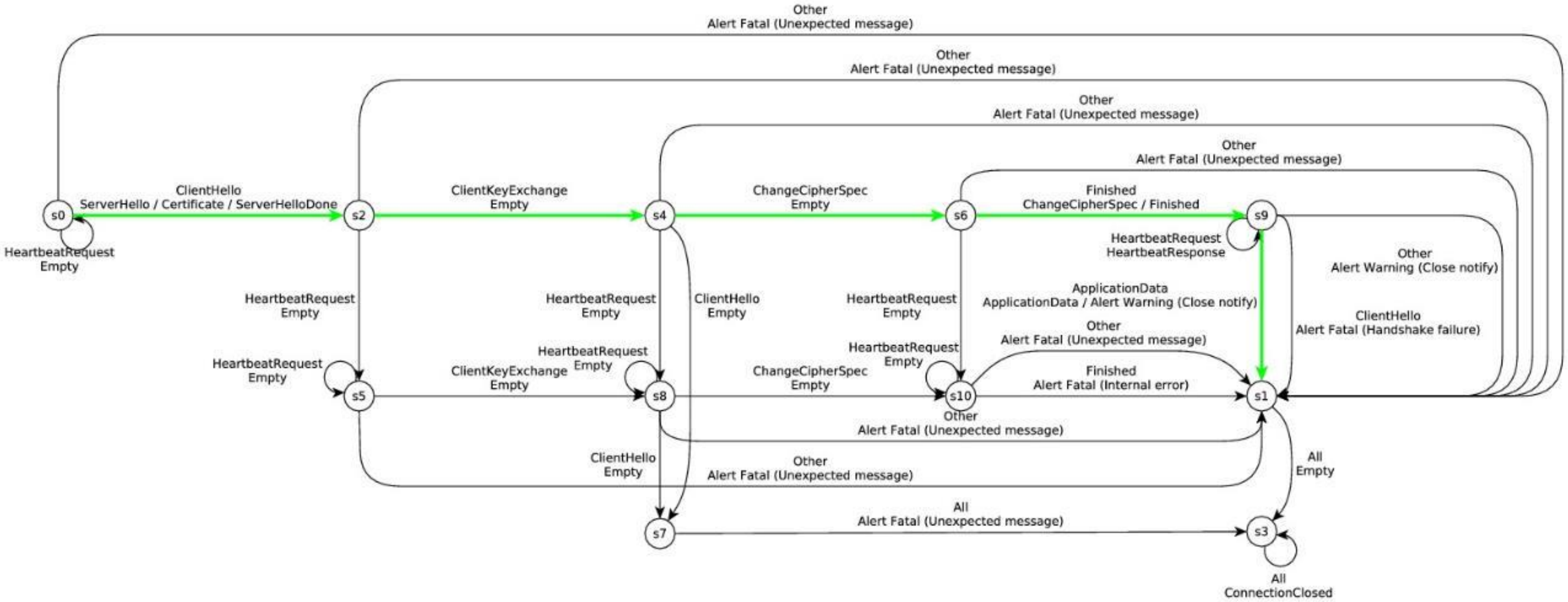
Case study: TLS

TLS state machine extracted from NSS

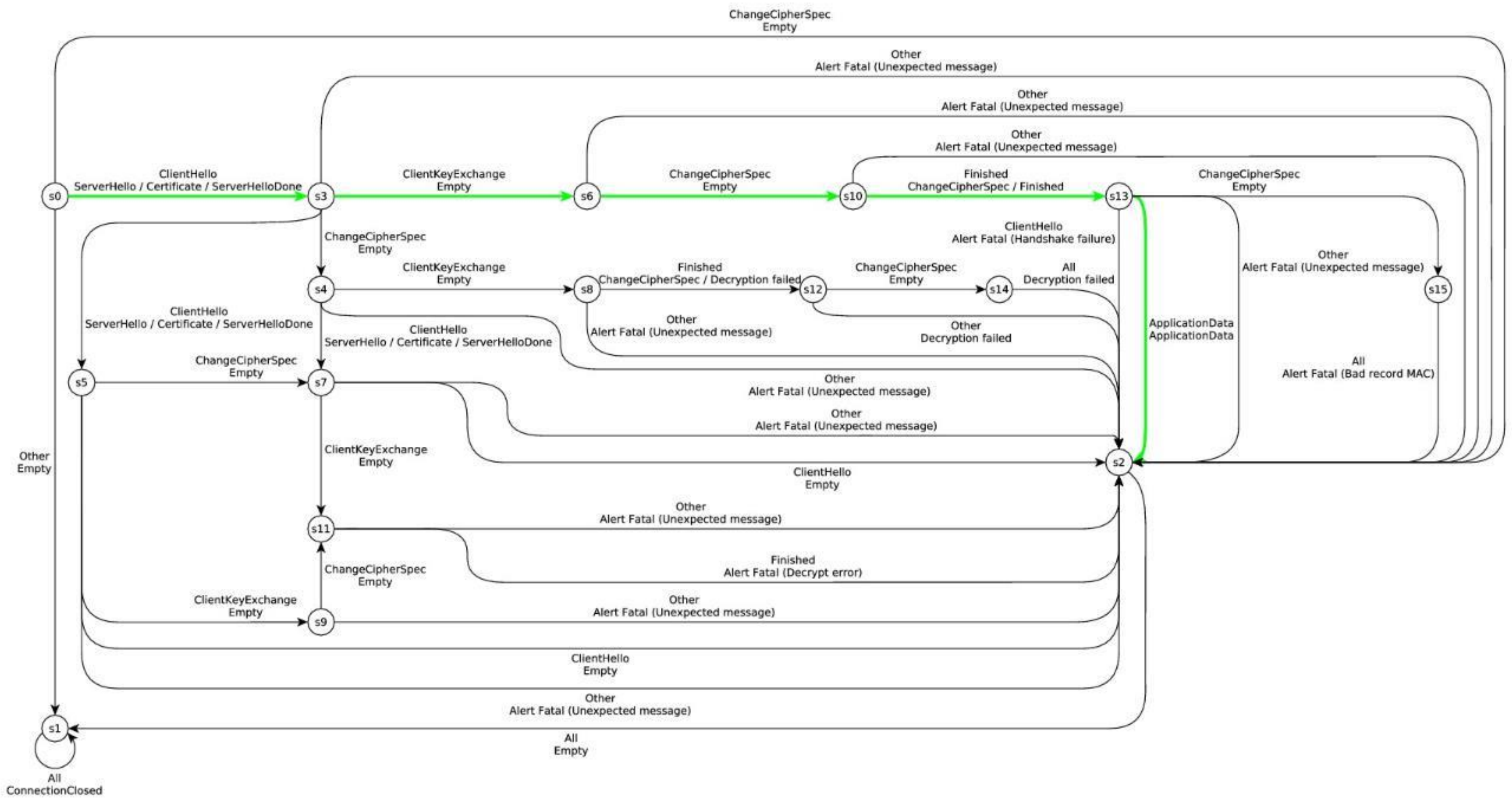


Comforting to see this is so simple!

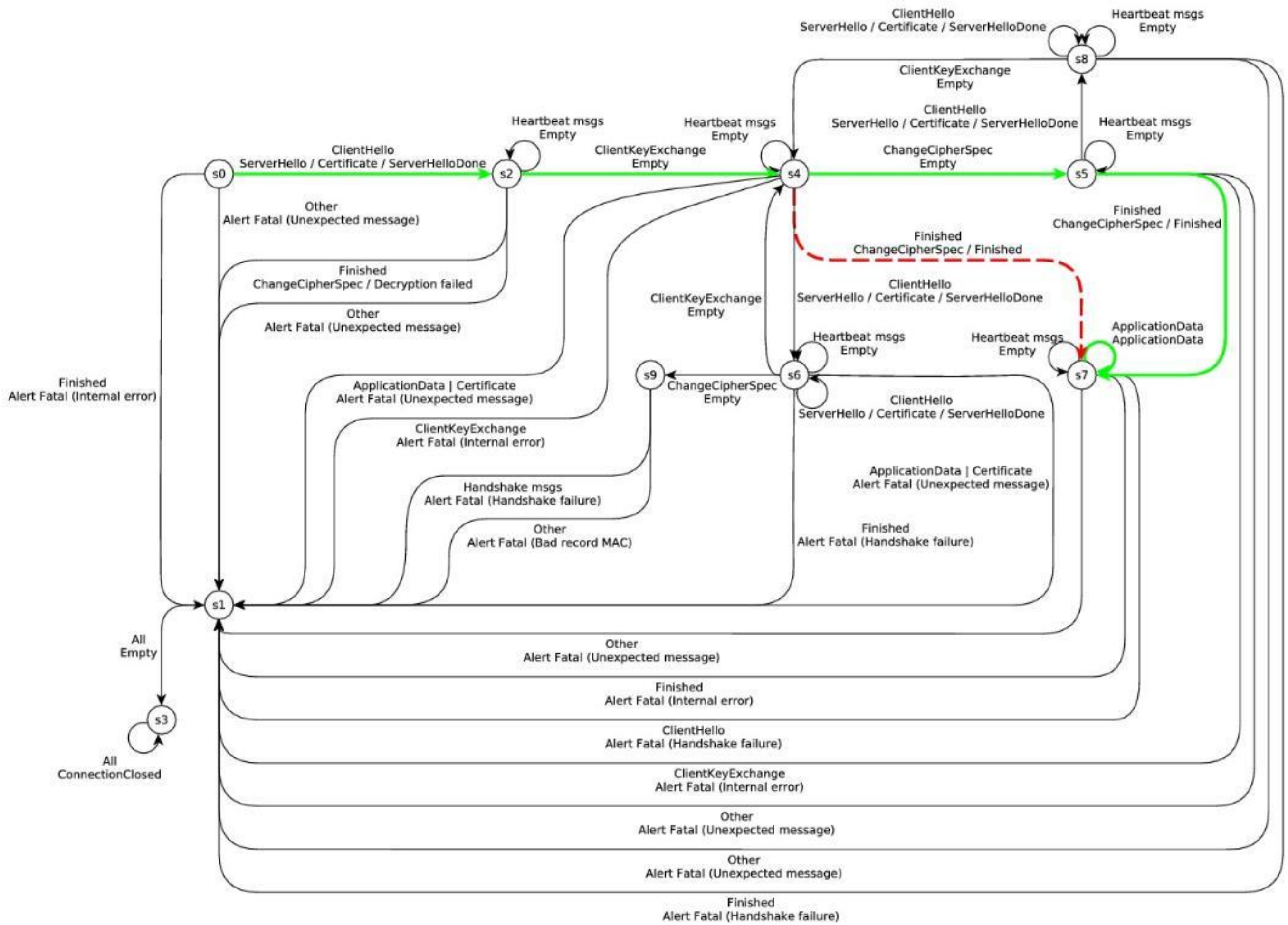
TLS state machine extracted from GnuTLS



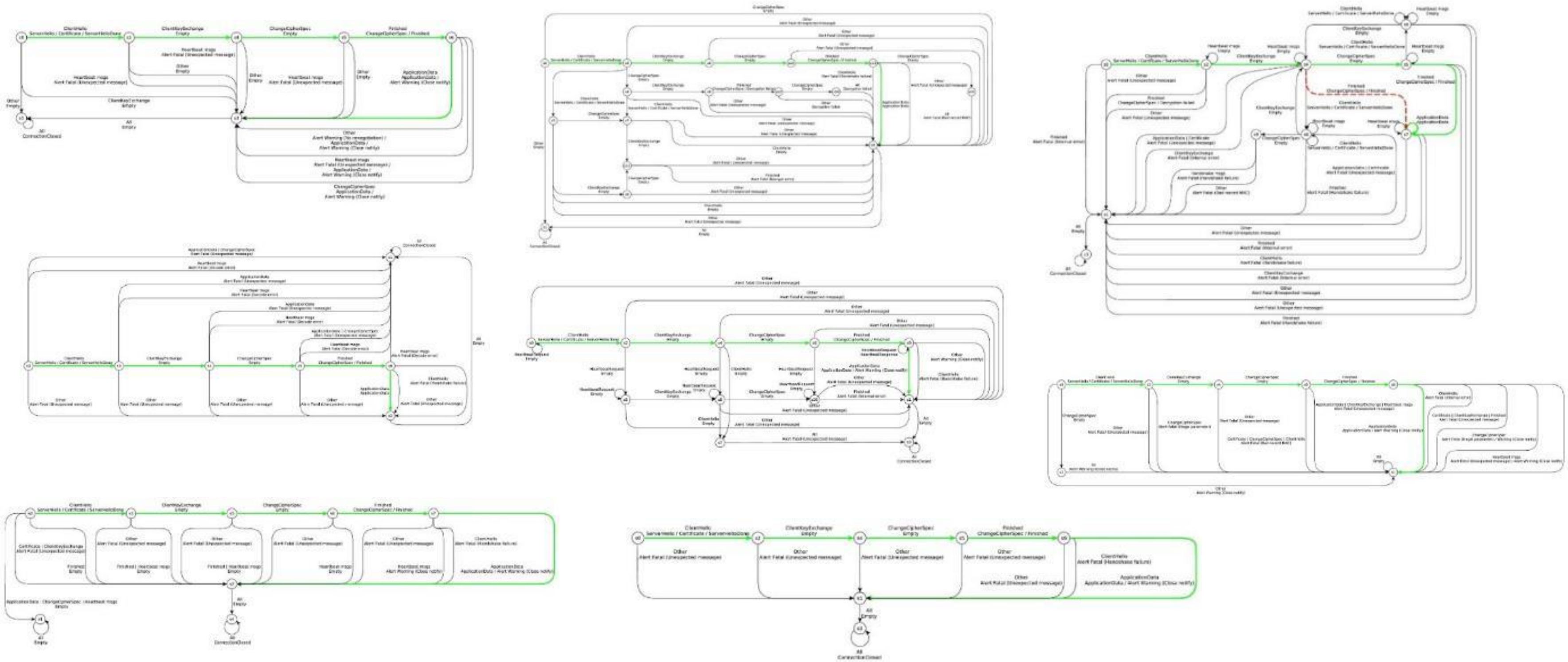
TLS state machine extracted from OpenSSL



TLS state machine extracted from JSSE



Which TLS implementations are correct? or secure?



New security flaws found in 3 out of 9 tested implementations;
recently discovered flaw in a 4th implementation could also be found.

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

Case study: internet banking

Internet banking token

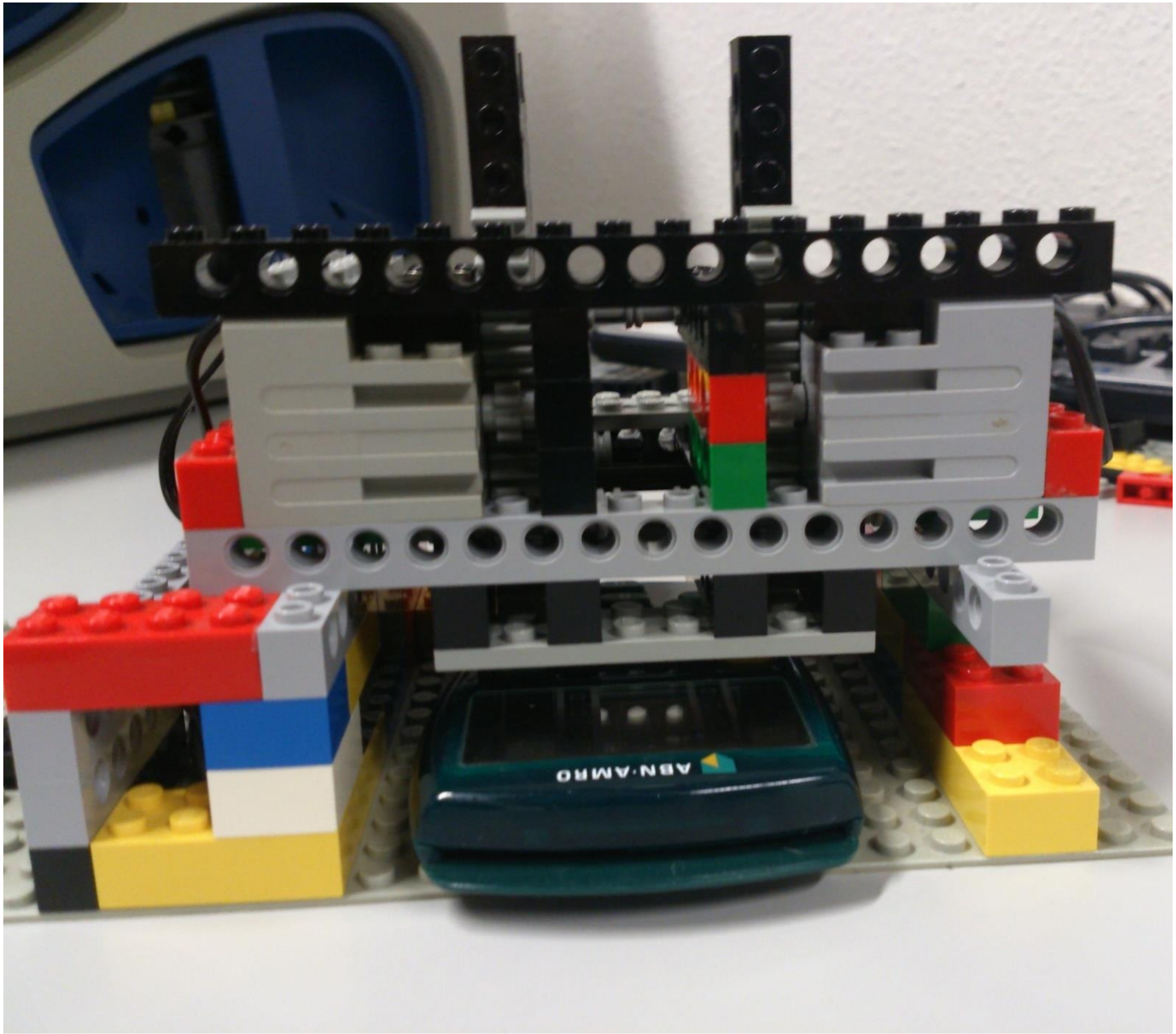
- smartcard reader for authenticating internet banking transactions

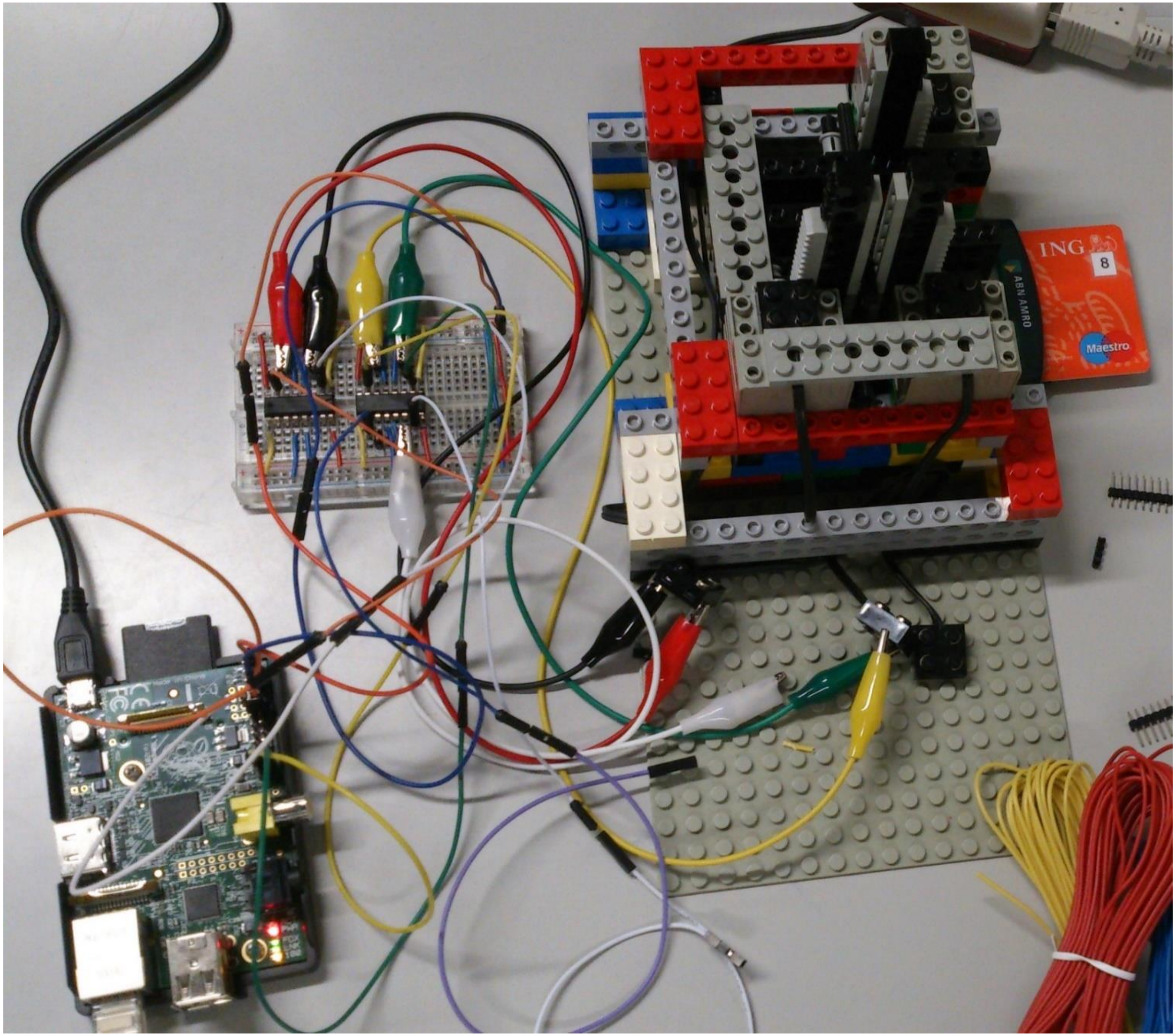
- USB-connected reader can be more user-friendly and more secure against Man-in-the-Browser attacks

- Security flaw in one such device issued by major Dutch bank: USB commands in a strange order would by-pass security check
  - NB bizarre that this device passed security evaluations!

- Can we use state machine learning to extract a model?

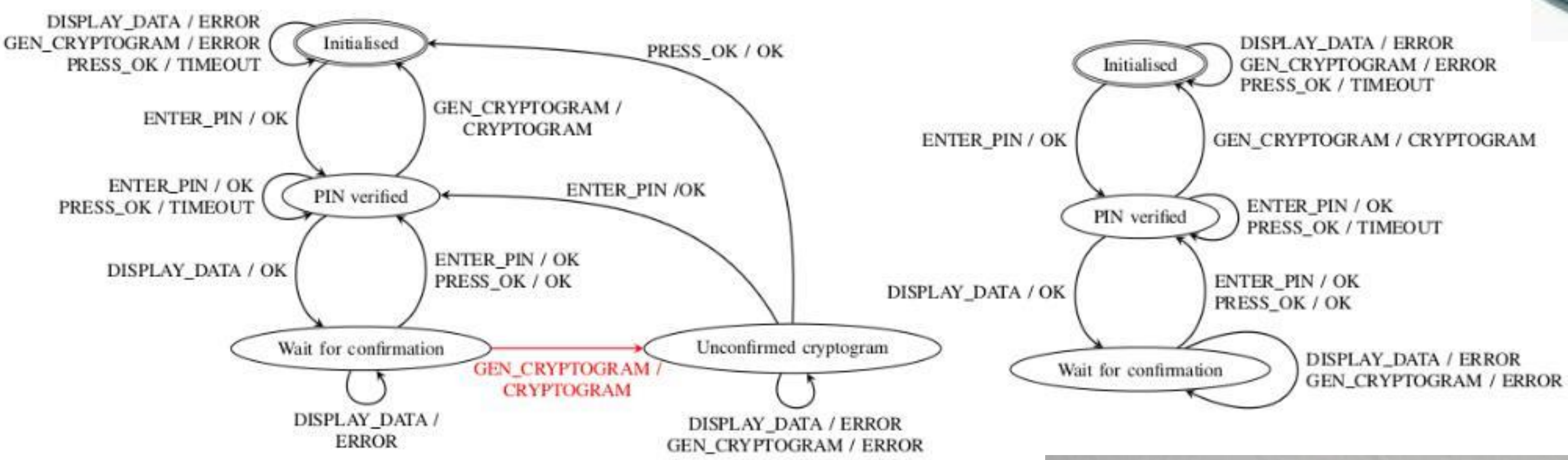
Operating the keyboard using LEGO

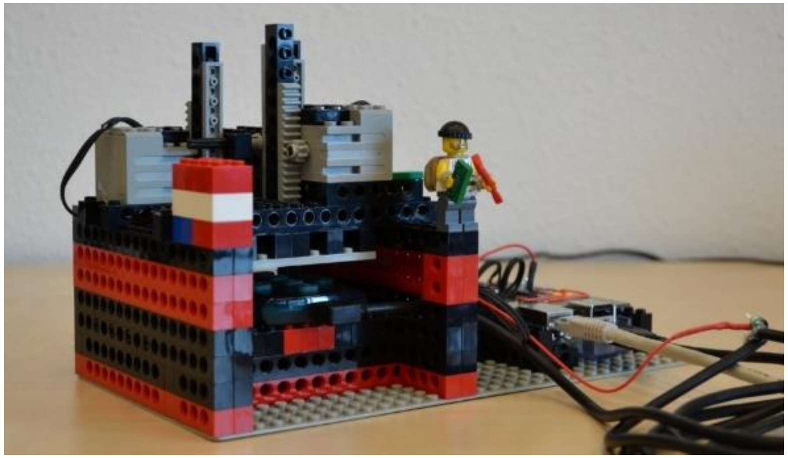

ABN·AMRO


ING
8
ABN AMRO
Maestro

State machine learning using LEGO

State machines inferred for flawed & patched device



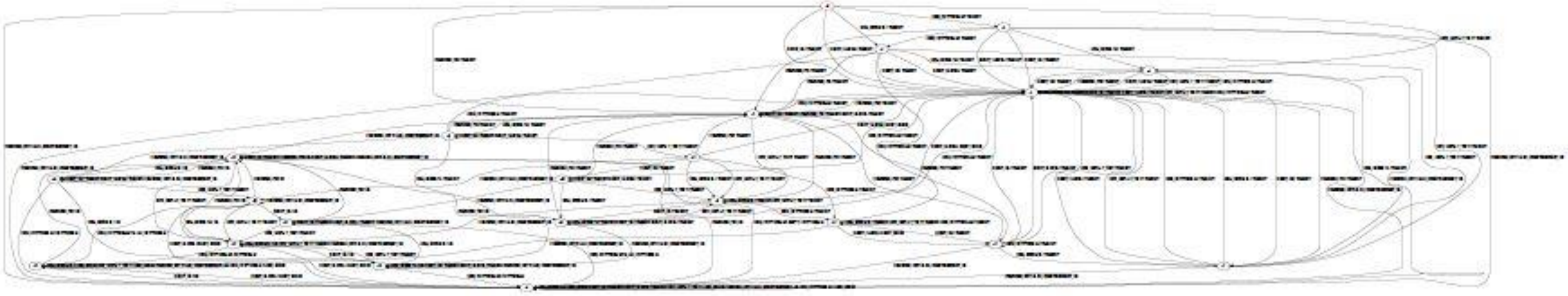


[Georg Chalupar et al.,
Automated reverse engineering using Lego,
WOOT 2014]

Movie at http://tinyurl/legolearn

Scary complexity

More complete state machine of the patched device, using a *richer* input alphabet



Aaargh!

We found no security flaws (using a model-checker), but were the developers confident that this behaviour is secure? Or necessary?

Conclusions

- State machines are a great specification formalism
  - easy to draw on white boards, typically omitted in official specs
- You can extract them for free from implementations
  - using very standard, off-the-shelf, learning techniques
  - “for free”, but you do have to implement a test harness
- Extracting state machine can reveal a certain class of security flaws
- Also useful to obtain
  - a formal spec to use in formal verification
  - legacy formal specs for existing code & protocols.
- Paying attention to protocol state machines can be regarded as a form of language-theoretic security (see langsec.org)

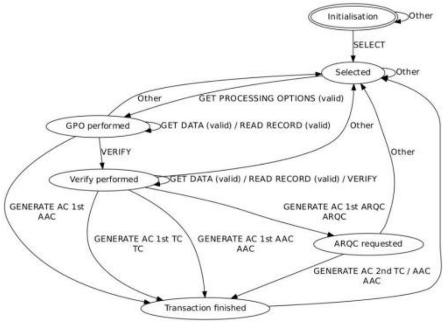
  [E. Poll et al. Protocol state machines and session languages, LangSec 2015]

specs

implementing

code

or via?



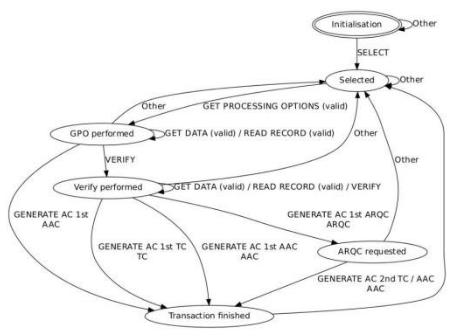
model

specs

code

model-based testing
or
verification

state machine
learning



model

Open issue & future work

- Can this technique discover security flaws in implementations of more protocols ?
- What is convenient way to present the complex state machines of real protocols?