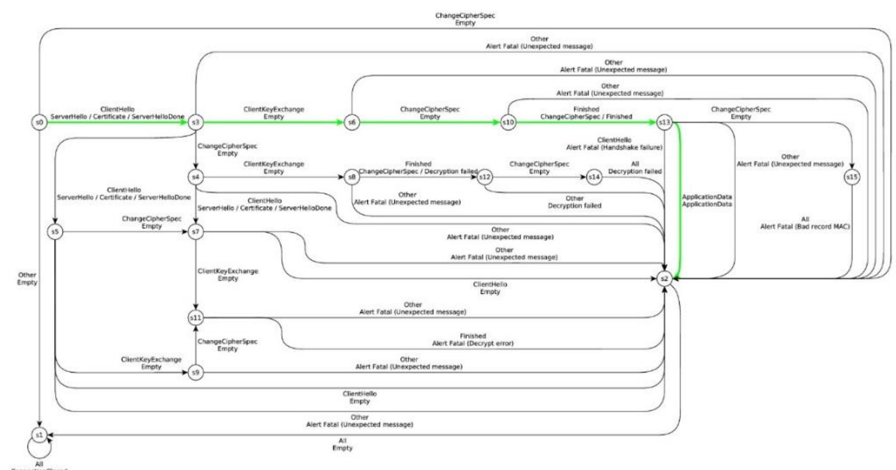
LangSec meets state machines

Erik Poll

joint work with Fabian van den Broek, Joeri de Ruiter & many others

Radboud University Nijmegen

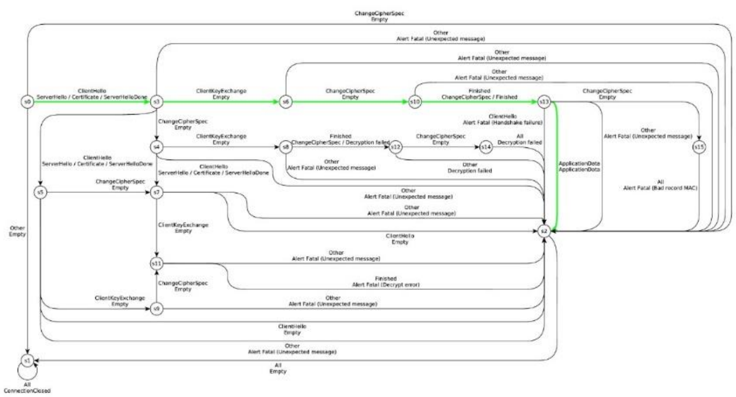


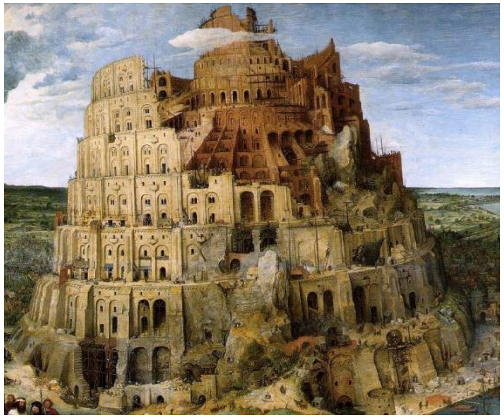
Overview

How can we tackle root causes of some classes of security vulnerabilities in a systematic way?

Two (related) ideas

- language-theoretic security (LangSec)
- state machines






LangSec
Language-theoretic
Security

LangSec (Language-theoretic Security)

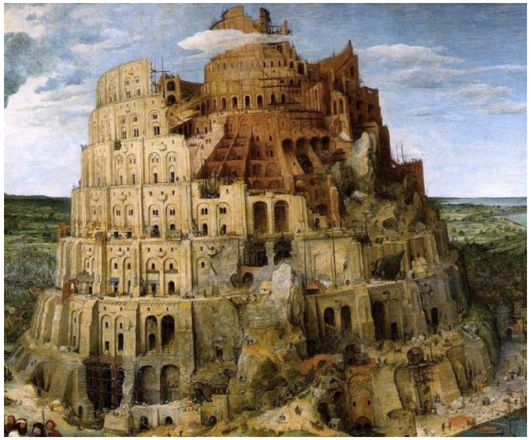
- Interesting look at root cause of large class of security problems, namely problems with **input**
- Useful suggestions for **do**s and **don't**s
- See langsec.org, esp. http://langsec.org/bof-handout.pdf

Sergey Bratus & Meredith Patterson

Tower of Babel

Web browsers and web applications involve ***many* languages**

- HTTP(S), HTML, CCS, javascript, Flash, cookies & FSOs
- Ajax & XML, ActiveX, jpeg, mpeg, mp4, png, gif, SilverLight,
- user names, email addresses, URLs/URIs, X509 certificates,
- TCP/IP (IPv4 or IPv6), file names, directories, OS commands,
- SQL, LDAP, JSP, PHP, ASCII, Unicode, UTF-8, ...



Input attacks

The common pattern in many attacks

buffer overflows, format string attacks, integer overflow, OS command injection, path traversal attacks, SQL injection, HTML injection, PHP file name injection, LDAP injection, XSS, CSRF, database command & function injection, ShellShock, HeartBleed,...

1. attacker crafts some *malicious input*
2. software goes off the rails *processing* this

Like social engineering or hypnosis as attack vector on humans?

Processing input is dangerous!

Processing involves

1) parsing/lexing
2) interpreting/executing

Eg interpreting a string as filename, URL, or email address

This relies on some language or format

1) relies on syntax

2) on semantics

Insecure processing of inputs exposes *strange functionality* that the attacker can program & abuse: a weird machine

Fallacy of classic input validation?

Classical input validation:

filter or encode harmful characters (blacklist)

or, slightly better:

only let through harmless characters (whitelist)

But:

- Which characters are harmful (or required!) depends on the language or format. You need *context* to decide which characters are dangerous.
- Not only presence of funny characters can cause problems, but als the absence of other characters, or input fields that are too long or too short, ...

Root causes (*dont's*)

Obstacles in producing code without input vulnerabilities

1. *ad-hoc and imprecise notion of input validity*
2. *parser differentials*

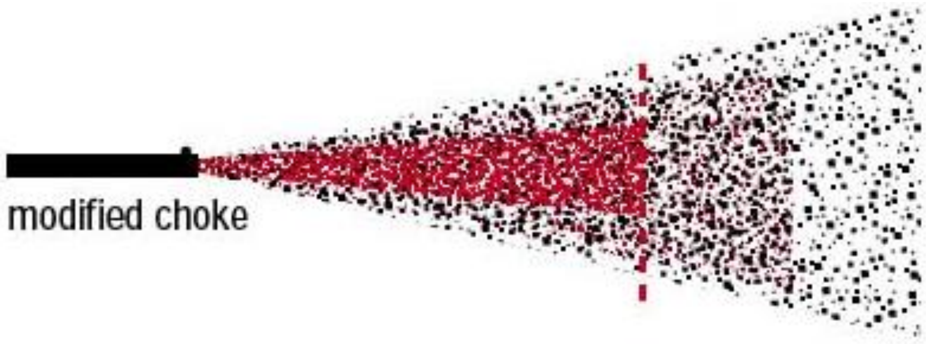
   eg web-browsers parsing same certificate in different ways
3. *mixing input recognition & processing*

   aka shotgun parsers
4. *unchecked development of input languages*

   eg ASCI text email evolving to include HTML, Javascript,...

Root cause: shotgun parsers

Handwritten code that incrementally parses & interprets input,
in a piece-meal fashion



Tell-tale signs in the code:

- use of strings or byte arrays
- code all over the place that parses and combines these

An example shotgun parser – spot the security flaw!

```
...
char buf1[MAX_SIZE], buf2[MAX_SIZE];
// make sure url is valid URL and fits in buf1 and buf2:
if (!isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// copy url up to first separator, ie. first '/', to buf1
out = buf1;
do {
  // skip spaces
  if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buf2, buf1);
...
```

loop termination flaw (for URLs without /) caused Blaster worm

LangSec principles (*do's*)

No more handwritten shotgun parsers, but

1. *precisely defined* input languages

   eg with EBNF grammar

2. *generated* parsers

3. *complete parsing before processing*

   So don't *substitute strings & then parse,*

   but *parse & then substitute in parse tree*

   Eg parameterised queries instead of dynamic SQL.

4. *keep the input language simple & clear*

   So that equivalence of various parsers is decidable.

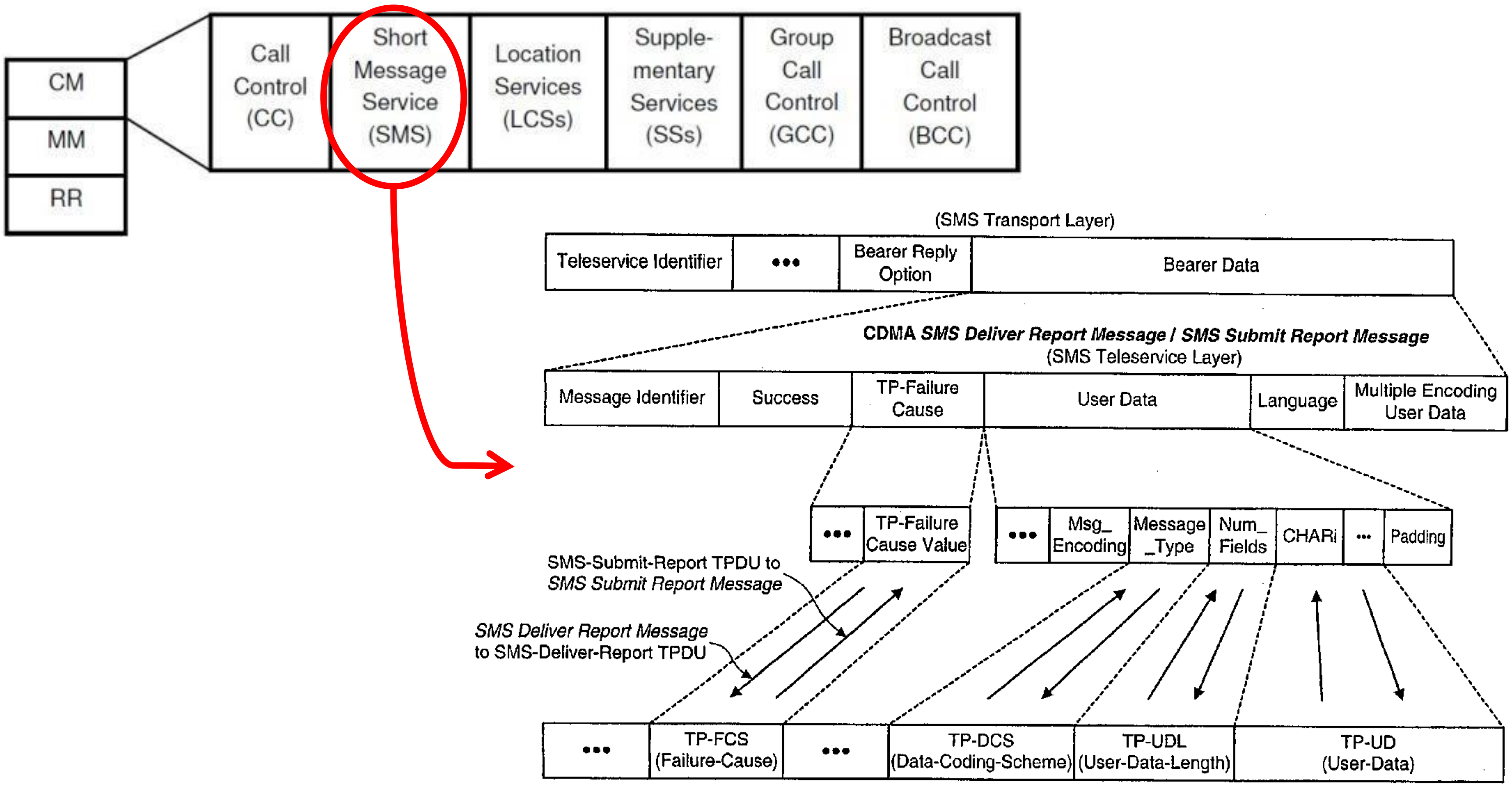
   So that you give minimal processing power to attackers.


NO MORE
Turing-
Complete
Input
Languages


FULL RECOGNITION
BEFORE PROCESSING

Example complicated input language: GSM

GSM is a extremely rich & complicated protocol



Example: GSM protocol fuzzing

Lots of stuff to fuzz!

With an USRP with OpenBTS software

we can fuzz phones

[Fabian vd Broek, Brinio Hond, Arturo Cedillo Torres, Security Testing of GSM Implementations, Essos 2014]

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird

eg possibility to send faxes (!?)

you have a fax!

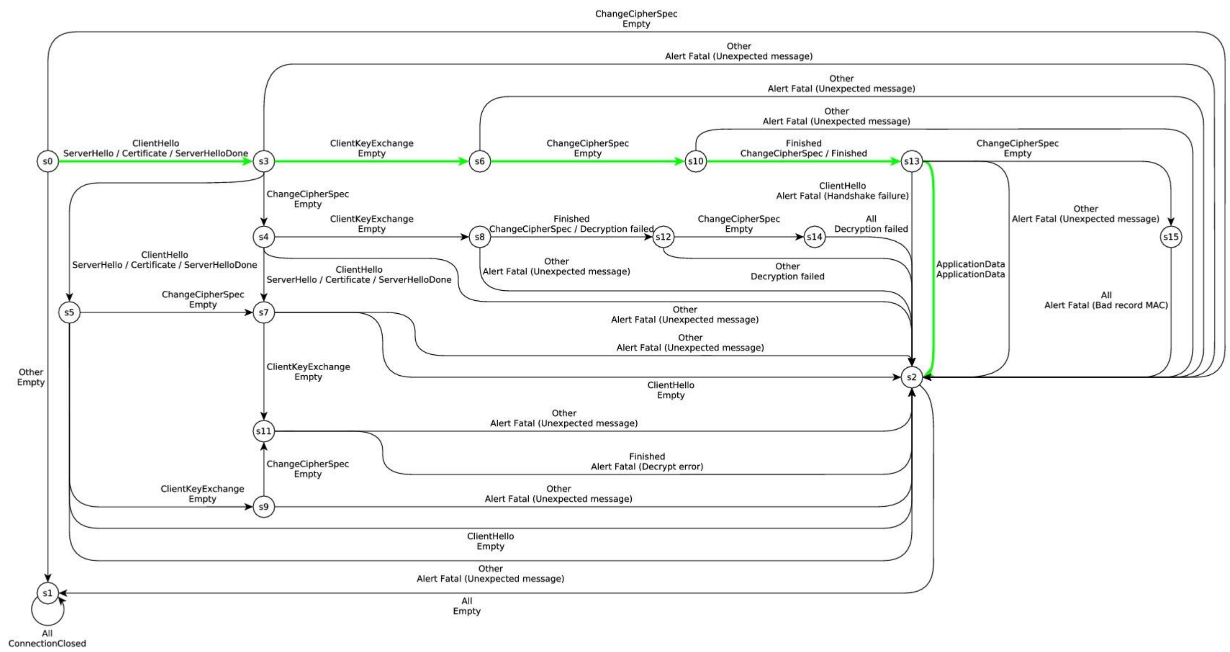
Only way to get rid if this icon; reboot the phone

Results with GSM protocol fuzzing

- Lots of success to DoS phones: phones crash, disconnect from the network, or stop accepting calls
- Little correlation between problems and phone brands & firmware versions
  - how many implementations of the GSM stack do vendors have?
- *The scary part: what would happen if we fuzz base stations?*

Root cause: complex input language, with lots of handwritten code to parse & interpret input

protocol state machines



Messages & *sequences of messages*

Protocols not only involve messages,

but also *sequences of messages*

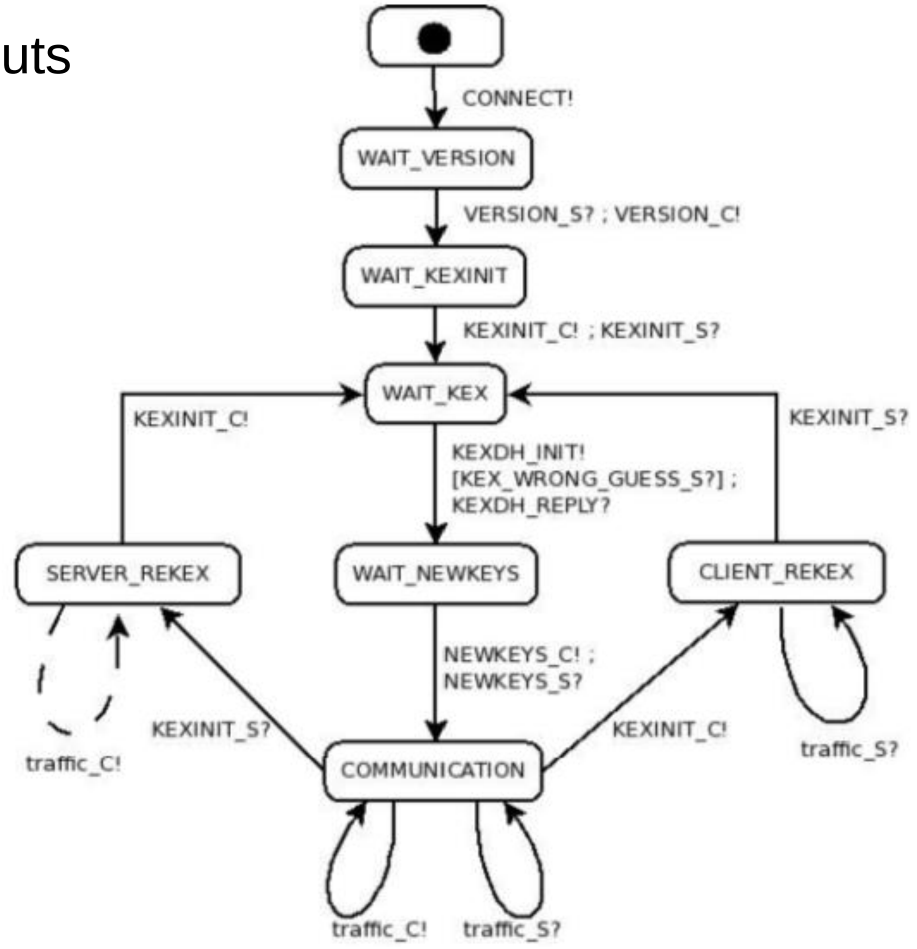
1. $C \rightarrow S$: `CONNECT`
2. $S \rightarrow C$: `VERSION_S` server version string
3. $C \rightarrow S$: `VERSION_C` client version string
4. $S \rightarrow C$: `SSH_MSG_KEXINIT` I_C
5. $C \rightarrow S$: `SSH_MSG_KEXINIT` I_S
6. $C \rightarrow S$: `SSH_MSG_KEXDH_INIT`e
   where $e = g^x$ for some 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. ...

- Steps 1–3: protocol identification
- Steps 4–5: key exchange algorithm negotiation
- Steps 6–9: key exchange
- Step 10 onwards: session, incl. SSH authentication and connection protocols

Using a protocol state machine (FSM)

Language for sequences of inputs can be specified using a finite state machine (FSM)

This state machne only describes the happy flows. The implementation will have to be input-enabled.



SSH transport layer

Typical prose specifications: RFC for SSH ☹

"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."

Understanding state machines from prose is hard!

Extracting state machines from code!

Using state machine learning we can *automatically* infer a state machine from implementation *by black box testing*.

- This is effectively a form of fuzzing.
    - not fuzzing the *content* of messages,
      but fuzzing the *order* of messages.
- Using variants of the L* algorithm,
  implemented in open source libraries such as LearnLib

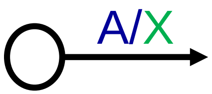
This is a great way to obtain protocol state machines

- without reading specs!
- withour reading code!

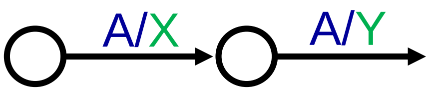
How does state machine learning work?

Just try out sequences of inputs, and observe outputs

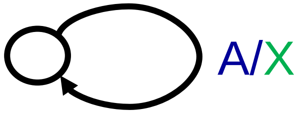
Suppose input A results in output X



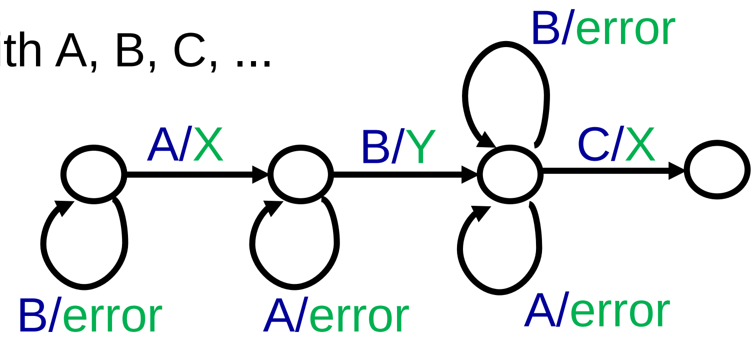
- If a second input A results in *different* output Y



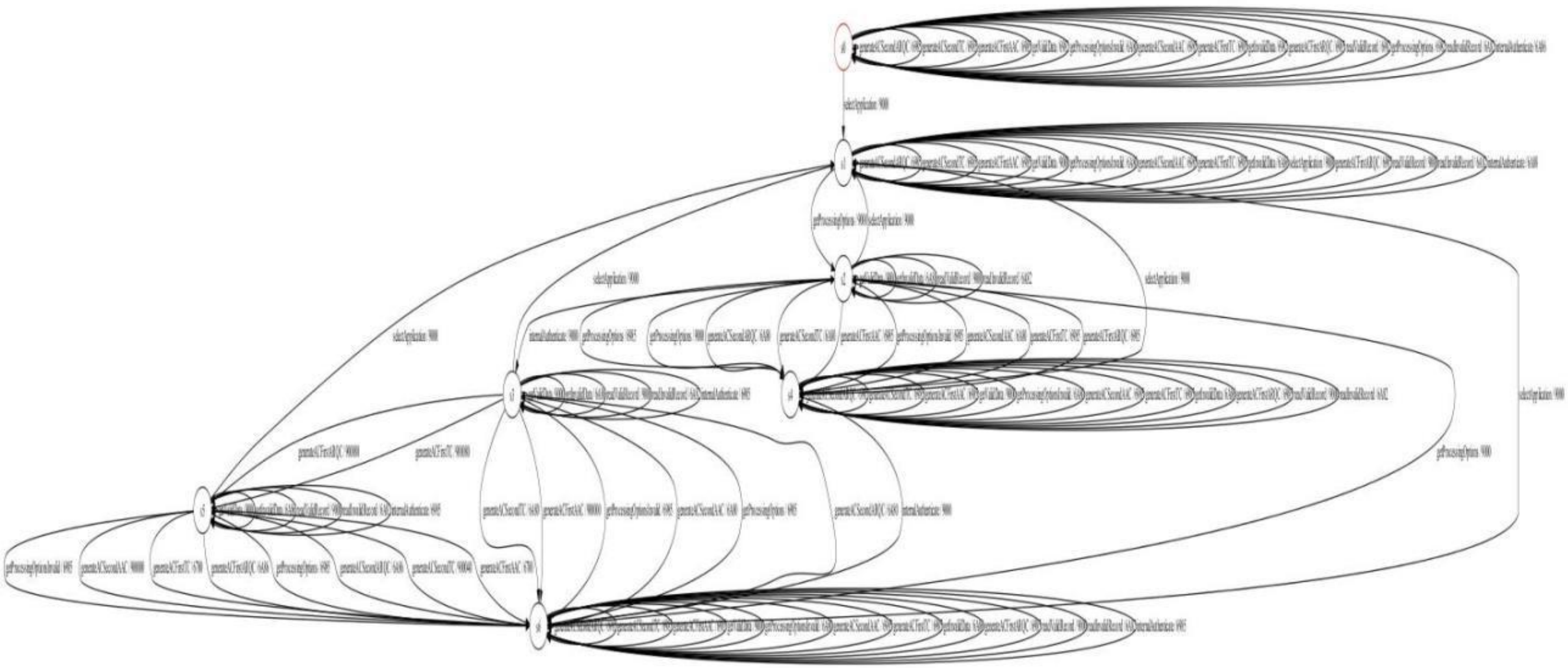
- If second input A results in the *same* output X



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

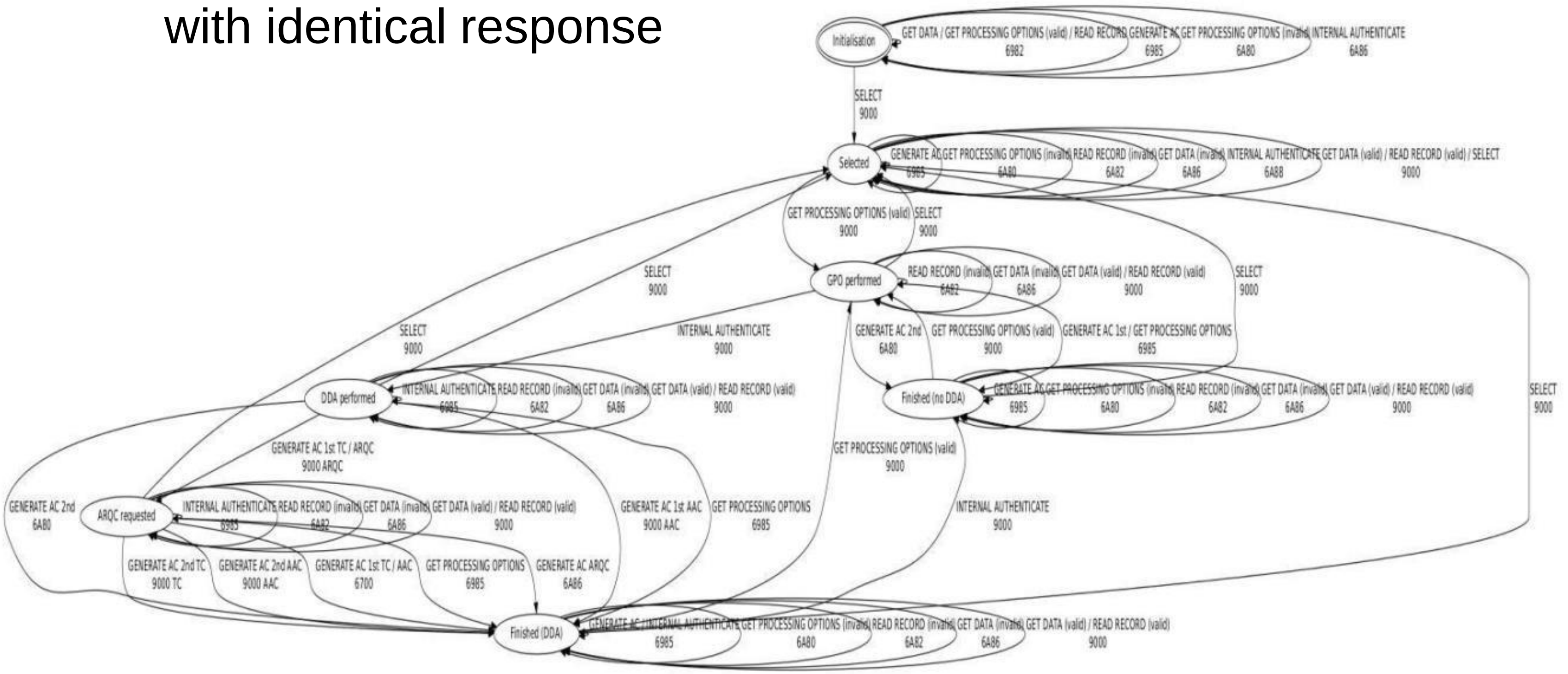


Example: state machine learning for



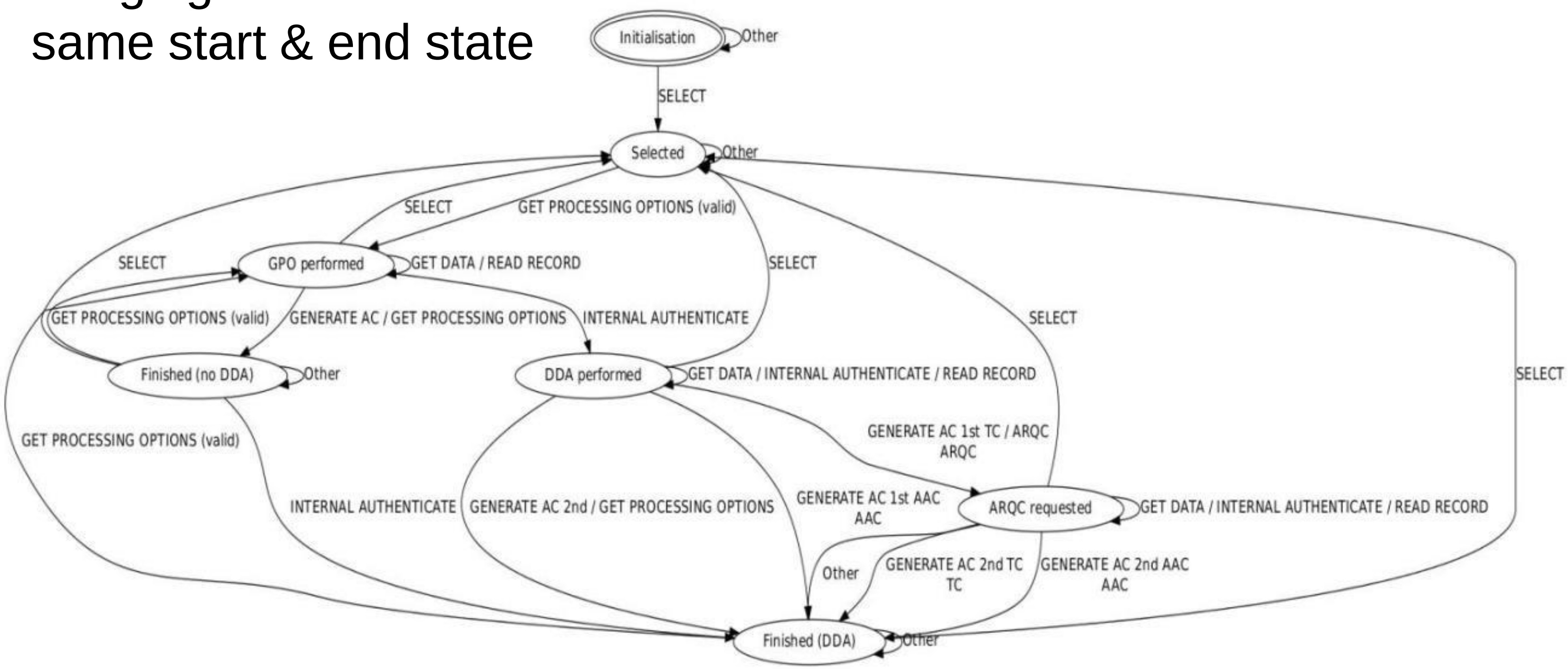
Example: state machine learning for

merging arrows
with identical response

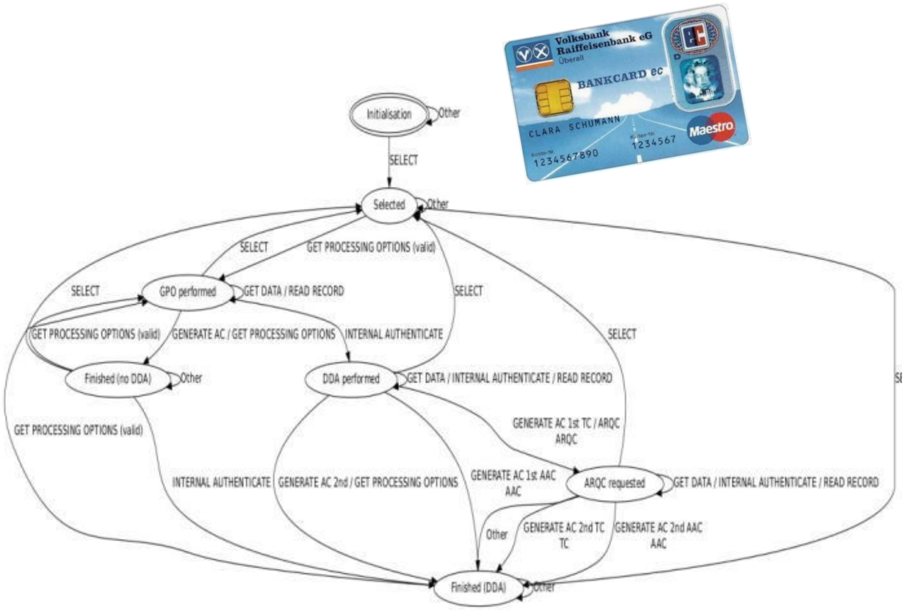


Example: state machine learning for

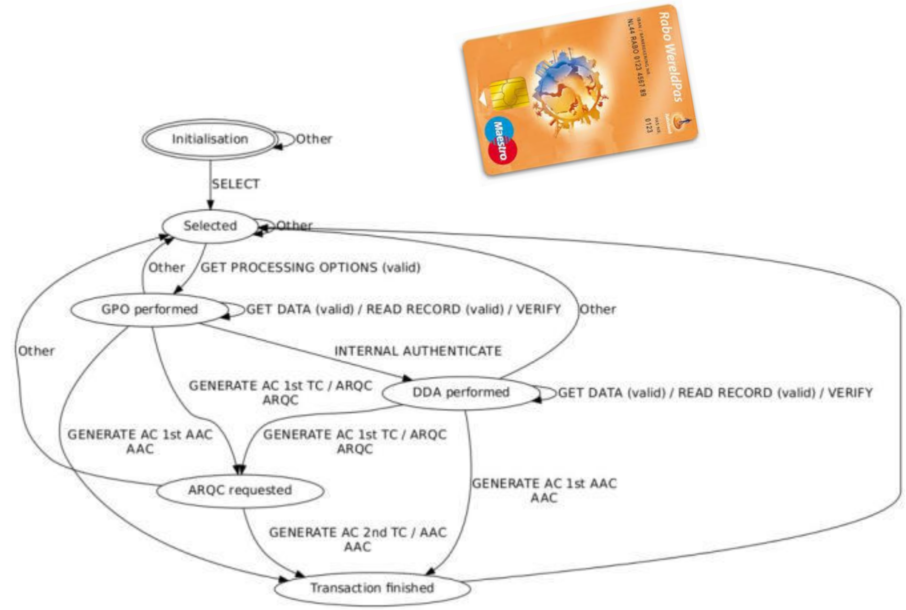
merging arrows with
same start & end state



Understanding & comparing implementations



Volksbank Maestro implementation



Rabobank Maestro implementation

Are both implementations correct & secure? And compatible?

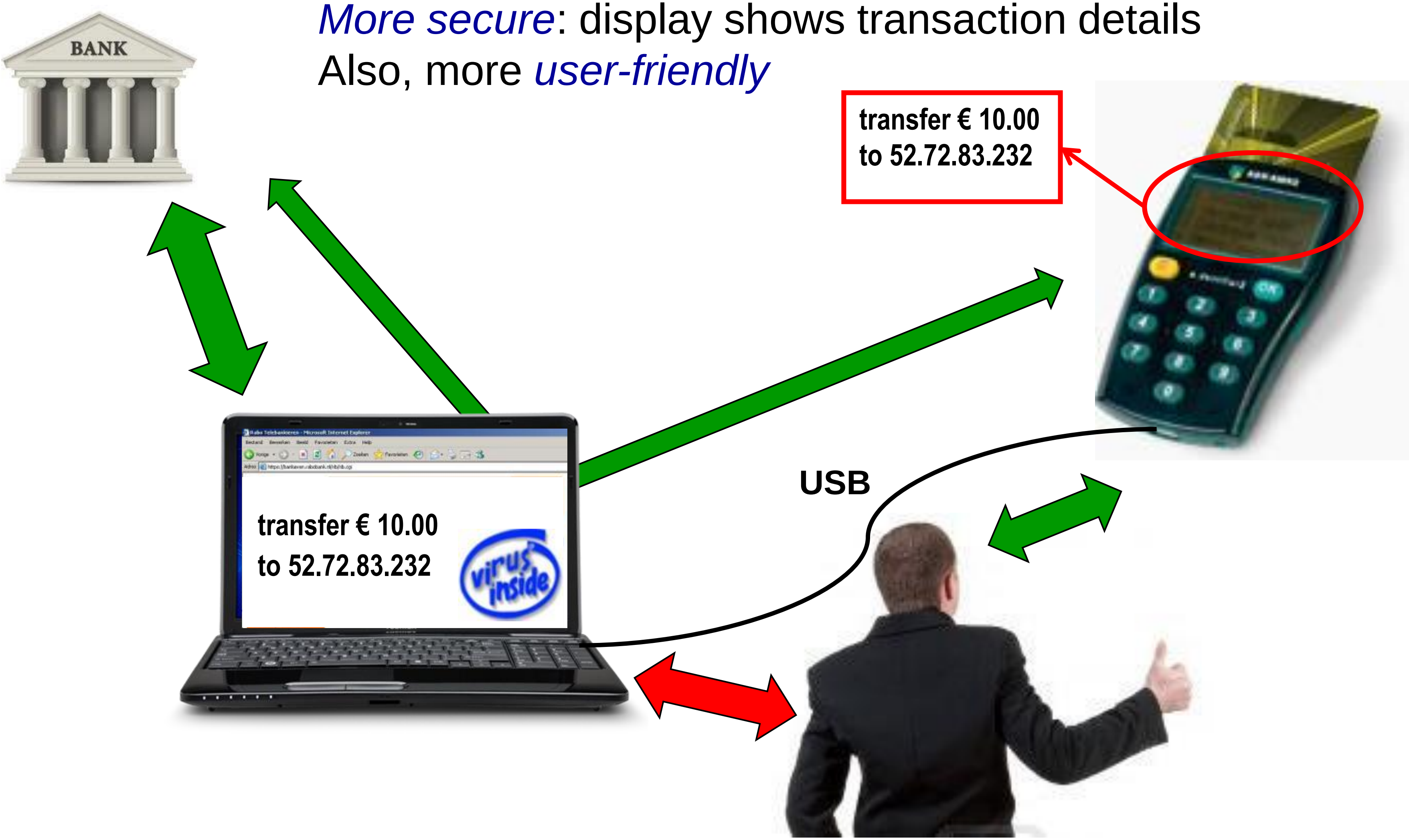
State machine inference for this device?

Internet banking with 

Internet banking with *USB-connected*

More secure: display shows transaction details

Also, more *user-friendly*



Security flaw in state machine

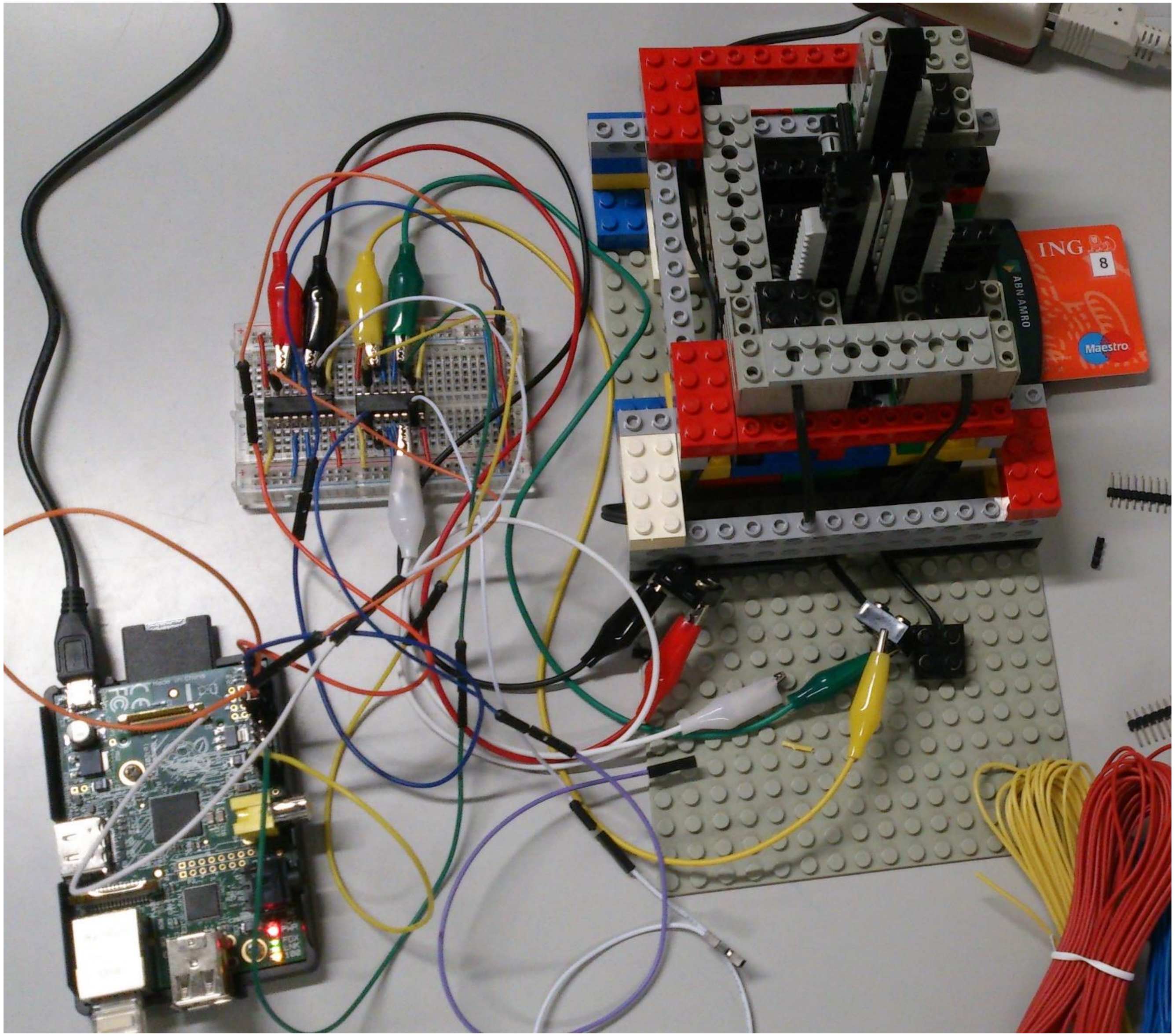
Embarrasing security flaw:

attacker can press the OK key via the USB cable

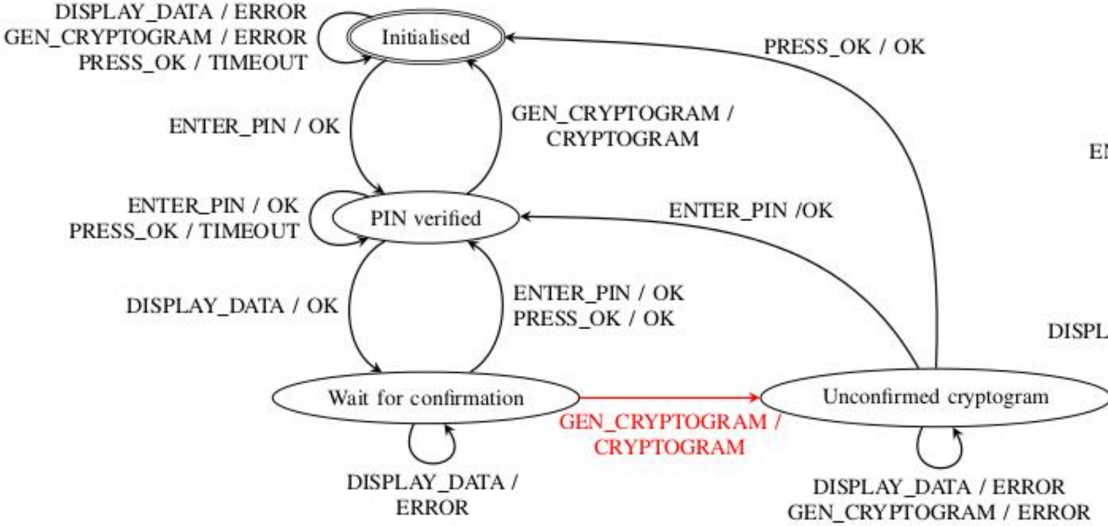
Could we detect such flaws automatically?

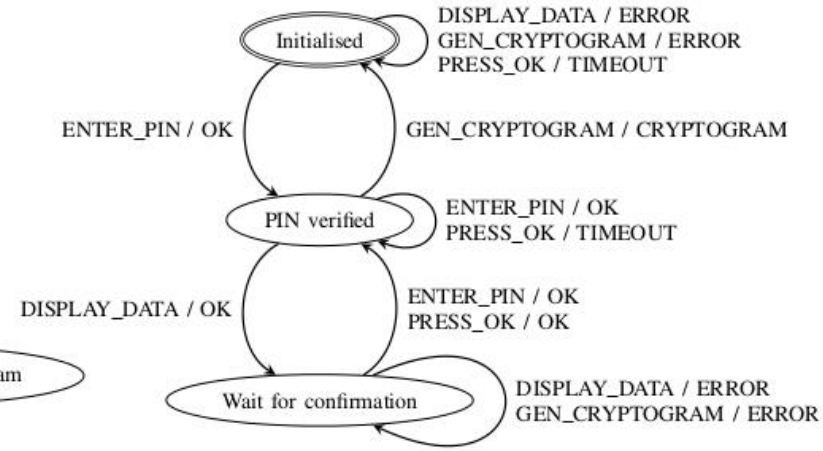
[Arjan Blom et al., Designed to fail, NordSec 2012]

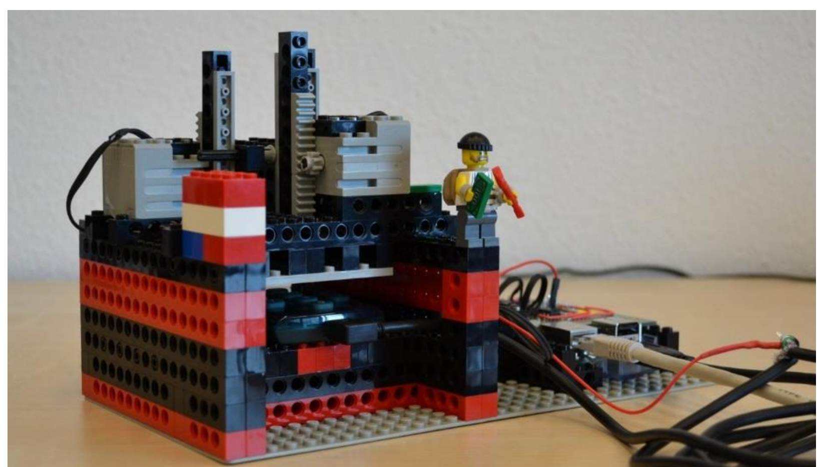
State machine learning using LEGO


ING
8
ABN AMRO
Maestro

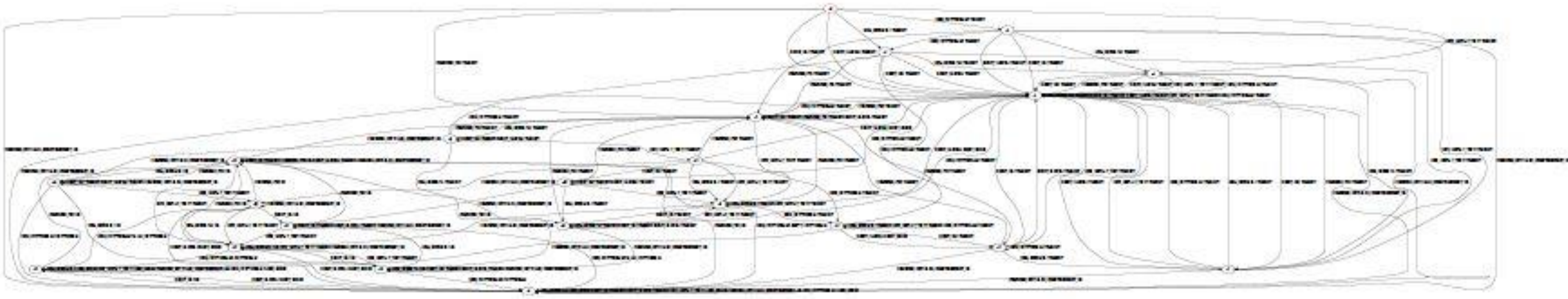
State machine of old vs new device







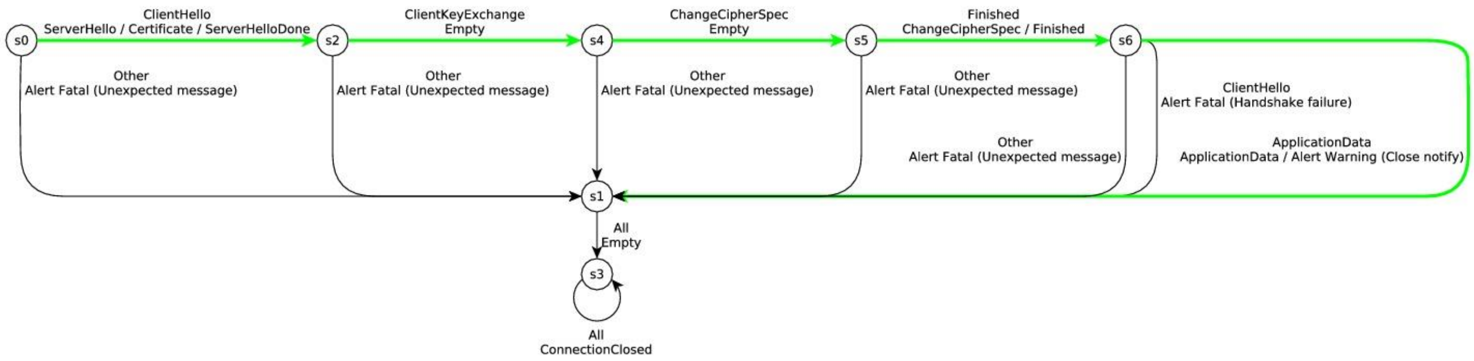
Would you trust this to be secure?



Complete state machine of new device, using richer alphabet of USB commands

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

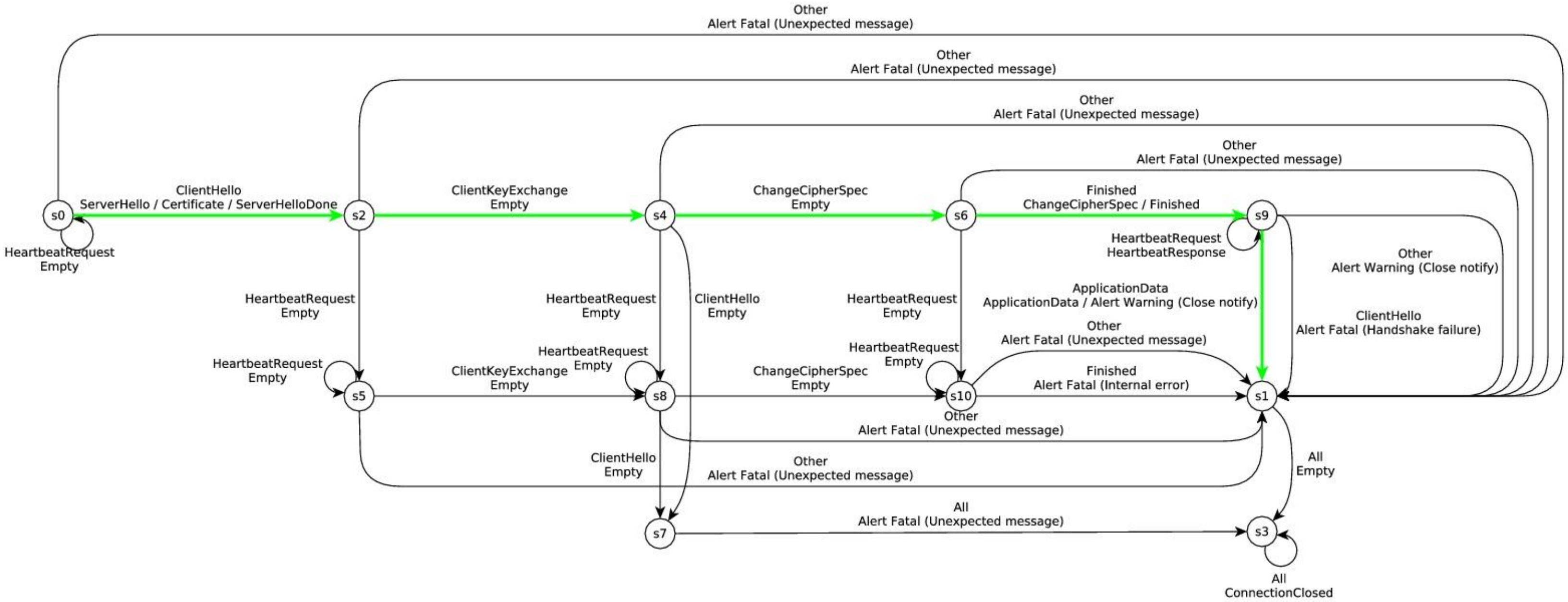
State machine learning for TLS



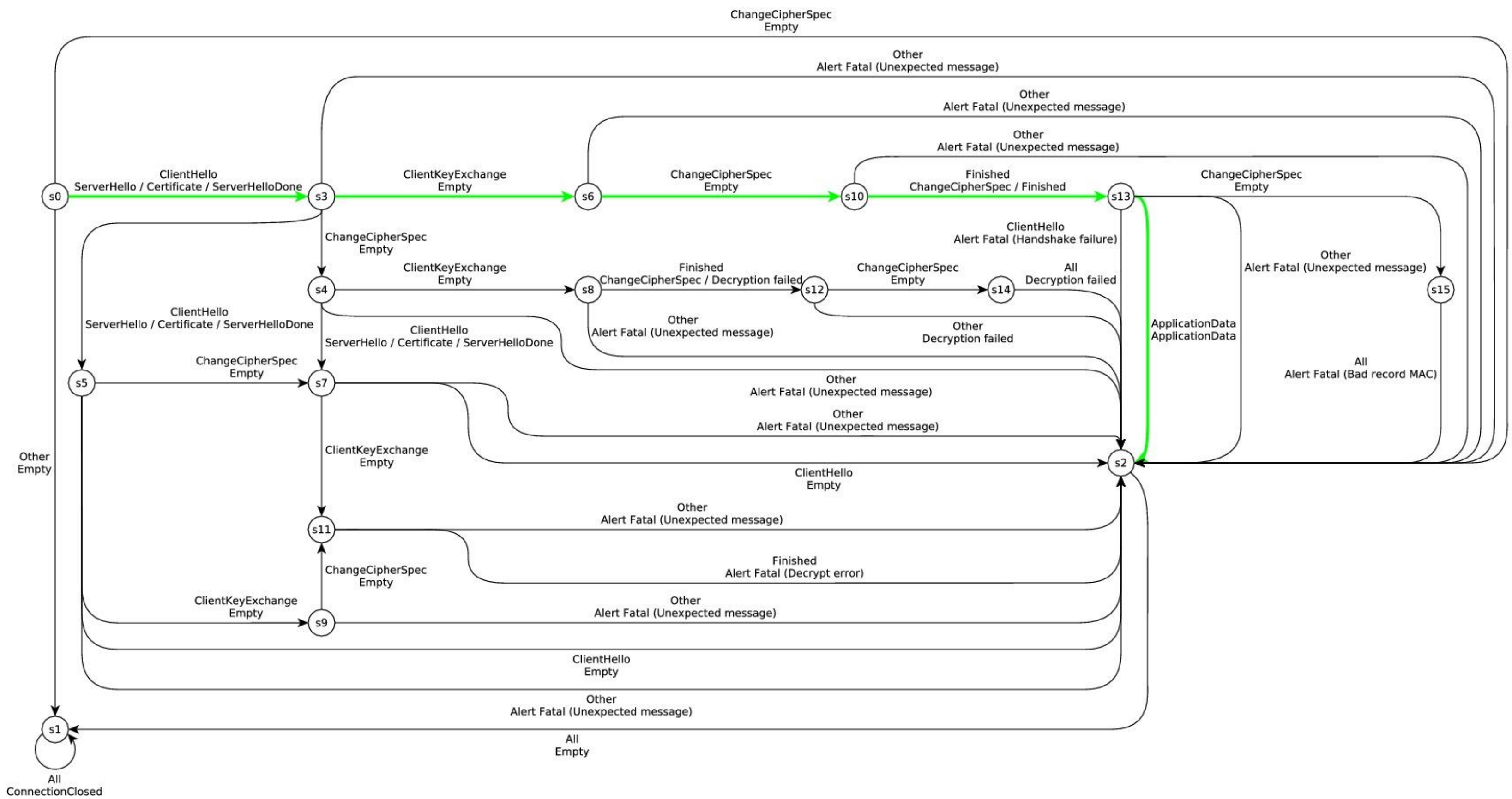
Model learned for the NSS implementation

Comforting to see this is so simple!

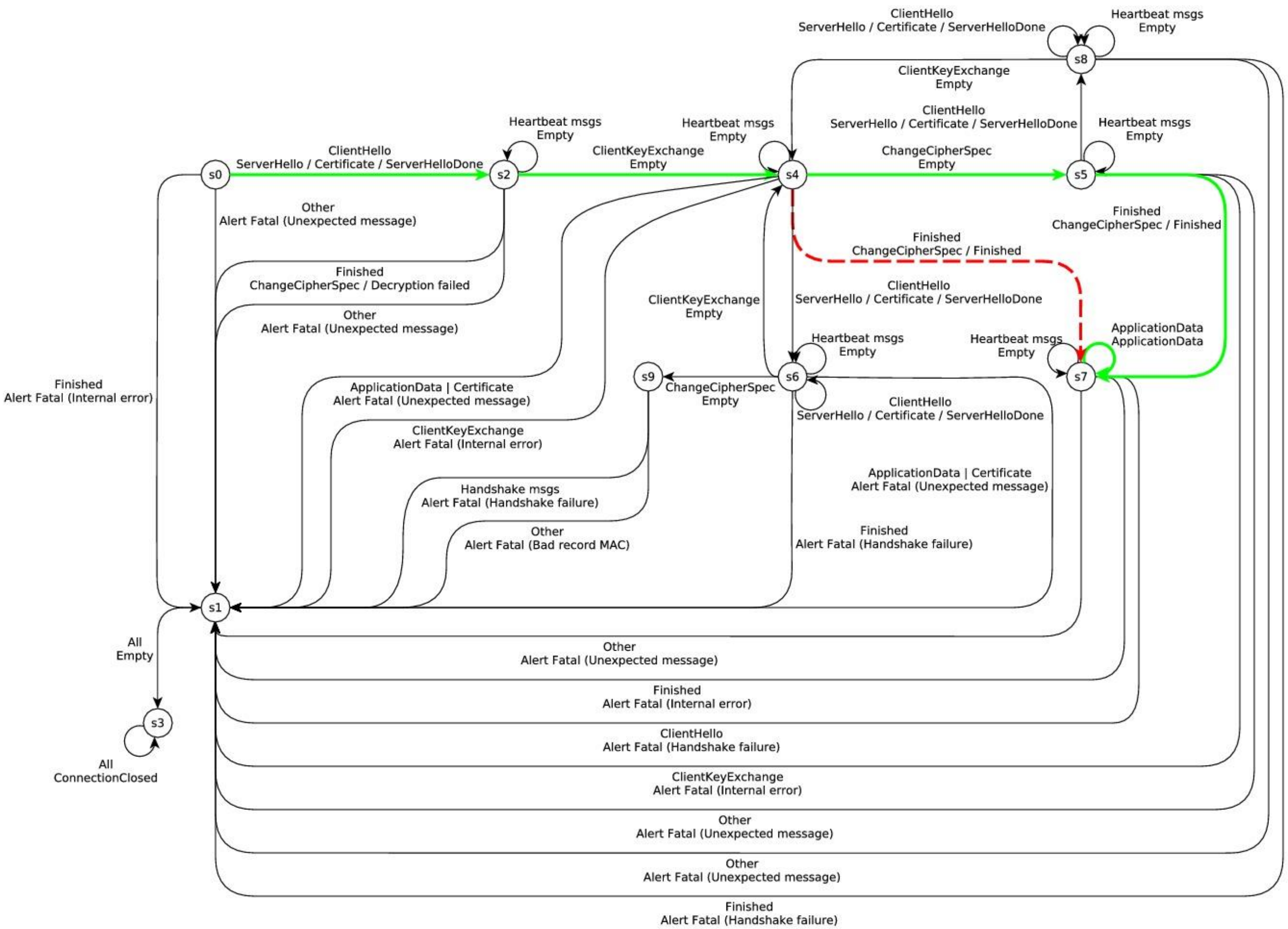
TLS... according to GnuTLS



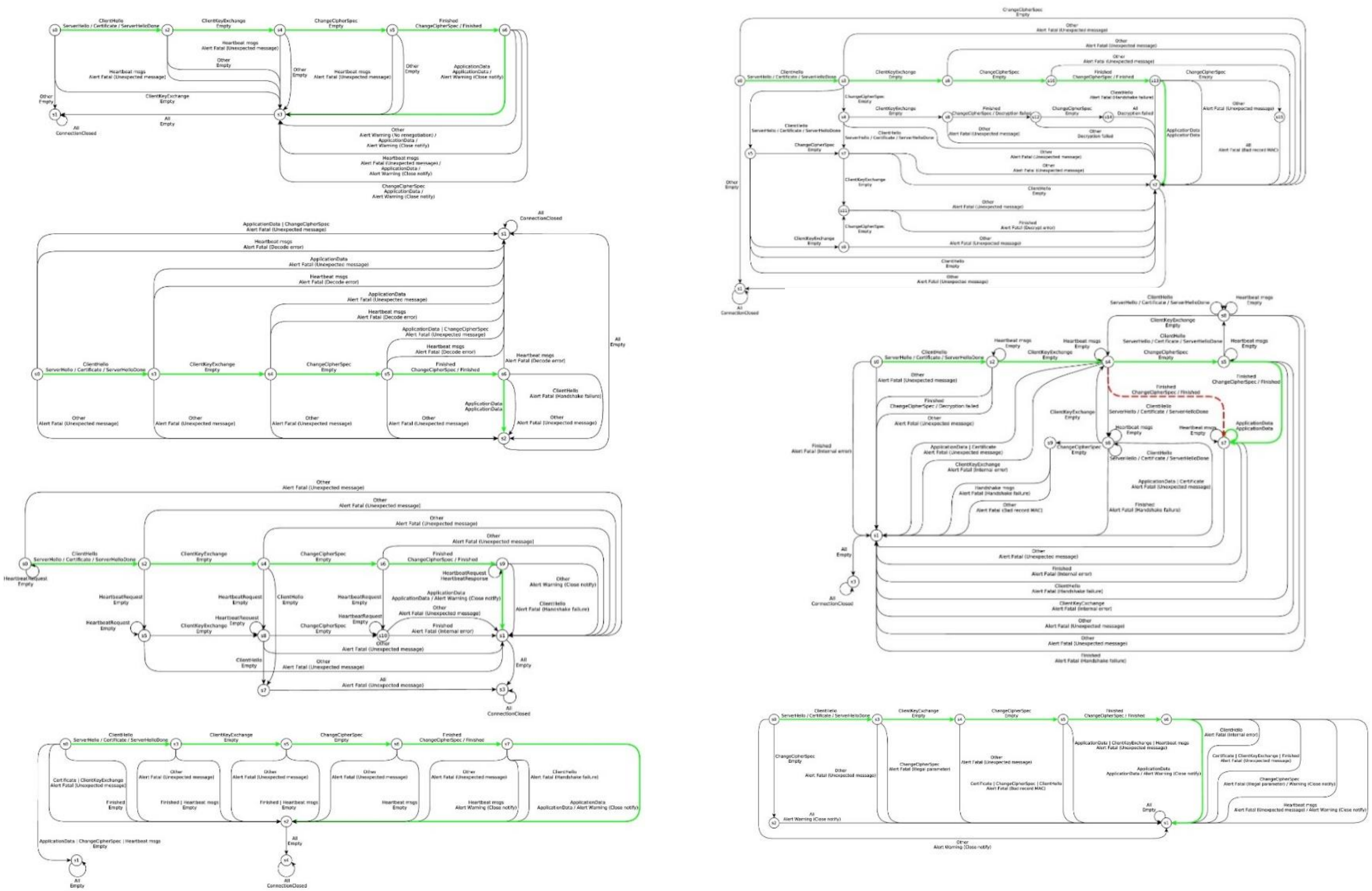
TLS... according to OpenSSL



TLS... according to Java Secure Socket Exension



Which TLS state machines are secure?



[Joeri de Ruiter and Erik Poll, Protocol State Fuzzing of TLS implementations, Usenix 2015]

Conclusions

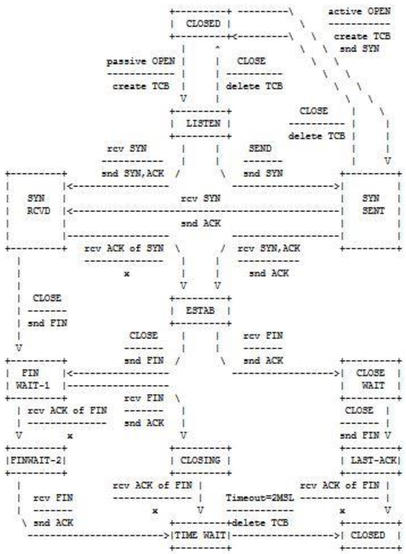
LangSec provides an interesting look at input problems

- explains root causes & a way to avoid theseS

State machines are great specification formalism

- to avoid ambiguities
- to help the programmer
- special case of LangSec,

  using state machine to express input language



TCP Connection State Diagram

Extracting state machines from code is great tool!

- analysis of existing implementations
- obtaining reference state machines for existing protocols

Thanks for you attention!