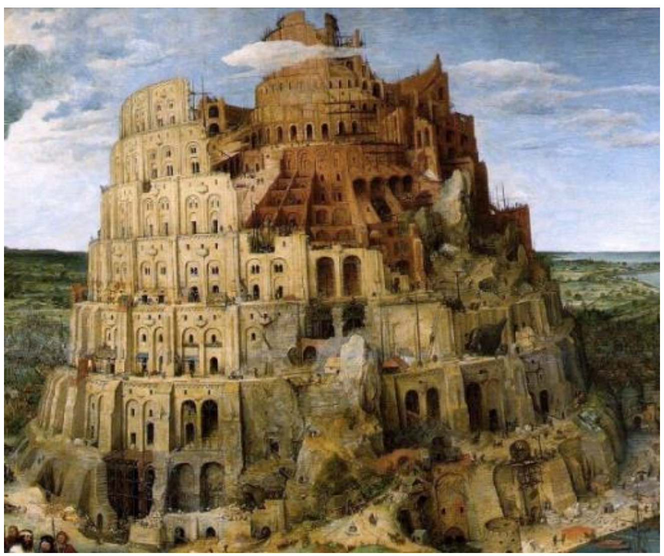
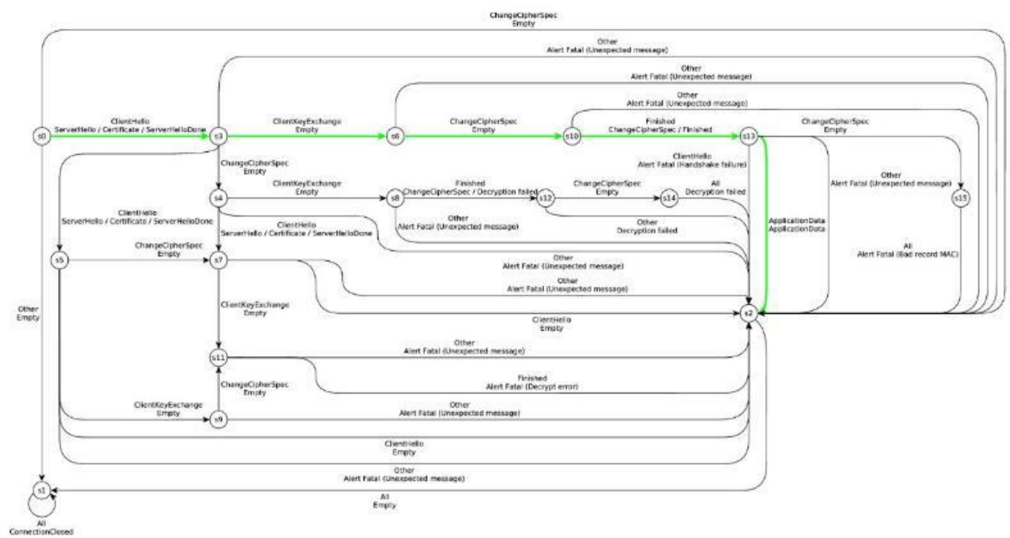
Protocol state machines & session languages

Erik Poll **Joeri de Ruiter** **Aleksy Schubert**

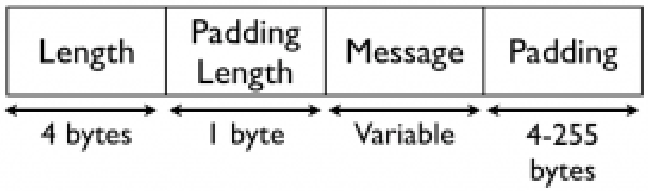




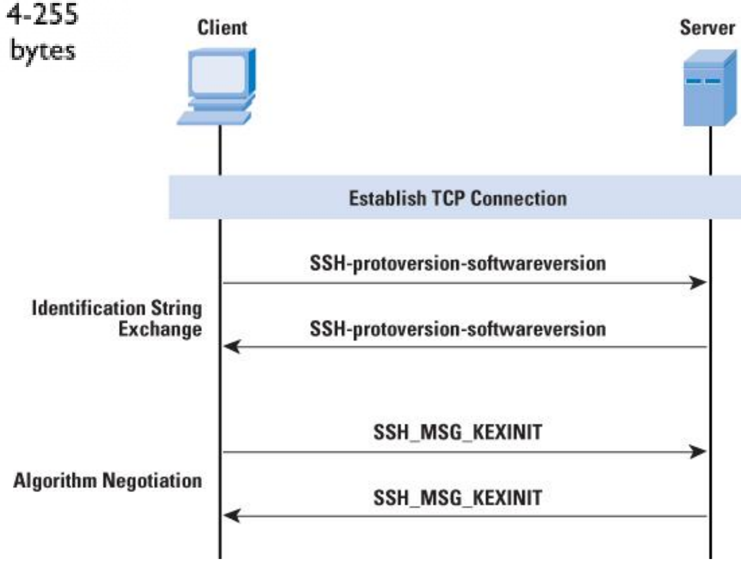
LangSec workshop @ IEEE Security & Privacy, 2015

Input languages: messages & sessions

- Handling inputs involves language of input messages



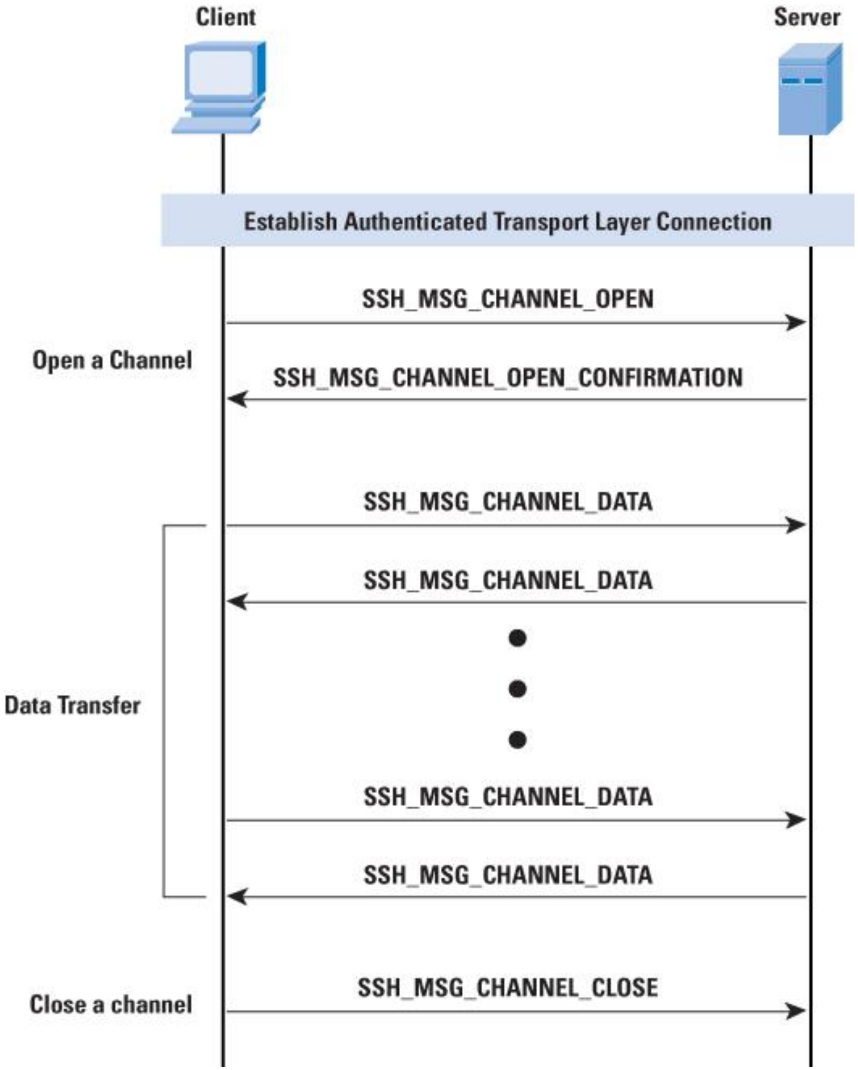
- Often it also involves language of sessions, ie. *sequences of messages*



- *Do LangSec principles also apply at this session level?*
  - when it comes to specification & implementation?

Session language as ***message sequence chart***

This *oversimplifies*
the session language
because it only specifies
one correct, happy flow



Session language as ***protocol state machine***

This *still oversimplifies*:

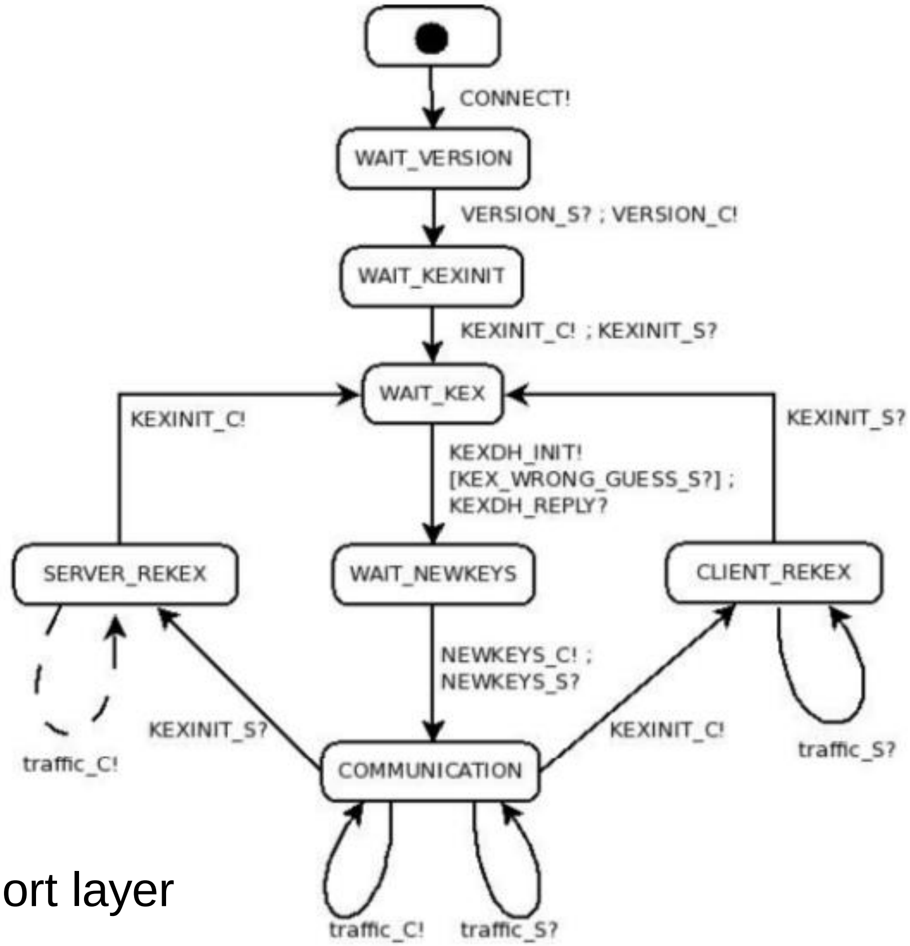
an implementation will have

to be *input-enabled,*

ie in every state

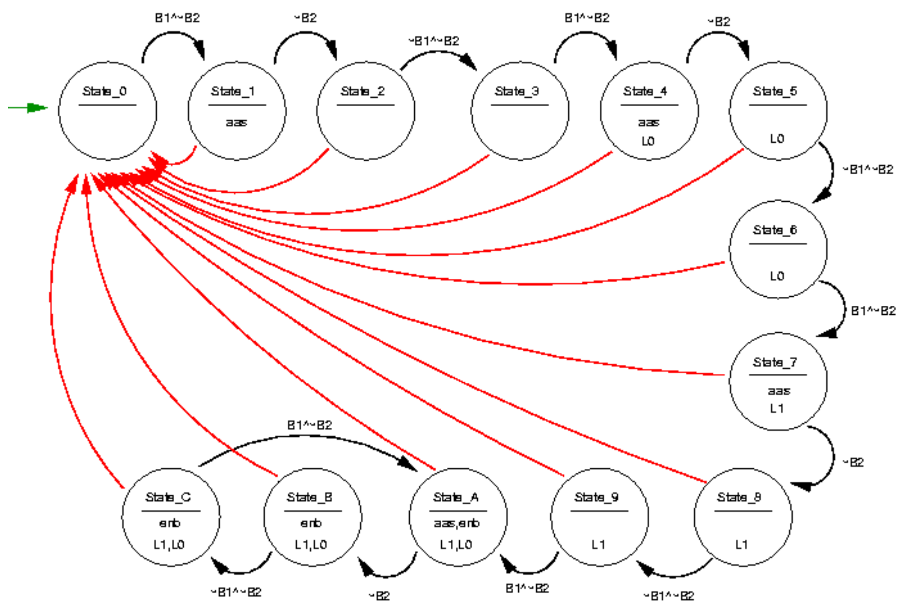
every message

may be received



SSH transport layer

typical ***input enabled*** state machine



Security flaws due to broken state machines

- MIDPSSH

  Open source Java implemention of SSH for Java feature phones

  No protocol state machine implemented at all.

  [Erik Poll at al., Verifying an implementation of SSH, WITS 2007]

- e.dentifier2

  USB-connected device for internet banking

  Strange sequence of USB commands by-passes user OK

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

- TLS

  Flawed state machines in many TLS implementations - more to come

  [Benjamin Beurdouche et al, A messy State of the union, IEEE Security & Privacy 2015]

Typical prose specifications: 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 machine from prose is hard!

Typical implementation: `openssh`

```
laptop:/home/erikpoll/openssh/src
erikpoll@laptop:~/.../src$ ls
aclocal.m4         auth-skey.c     dh.h             mac.c             platform.h        sftp-client.c       ssh-dss.c
acss.c             bufaux.c        dispatch.c       mac.h             progressmeter.c   sftp-client.h       ssh-gss.h
acss.h             bufbn.c         dispatch.h       Makefile          progressmeter.h   sftp-common.c       ssh.h
addrmatch.c        buffer.c        dns.c            Makefile.in       PROTOCOL          sftp-common.h       ssh-keygen
atomicio.c         buffer.h        dns.h            Makefile.inc      PROTOCOL.agent    sftp-glob.c         ssh-keygen.0
atomicio.h         buildpkg.sh.in  entropy.c        match.c           readconf.c        sftp.h              ssh-keygen.1
audit-bsm.c        canohost.c      entropy.h        match.h           readconf.h        sftp-server         ssh-keygen.c
audit.c            canohost.h      fatal.c          md5crypt.c        README            sftp-server.0       ssh-keyscan
audit.h            ChangeLog       fixpaths         md5crypt.h        README.dns        sftp-server.8       ssh-keyscan.0
auth1.c            channels.c      fixprogs         mdoc2man.awk      README.platform   sftp-server.c       ssh-keyscan.1
auth2.c            channels.h      groupaccess.c    md-sha256.c       README.privsep    sftp-server-main.c  ssh-keyscan.c
auth2-chall.c      cipher-3des1.c  groupaccess.h    misc.c            README.smartcard  ssh                 ssh-keysign
auth2-gss.c        cipher-acss.c   gss-genr.c       misc.h            README.tun        ssh.0               ssh-keysign.0
auth2-hostbased.c  cipher-aes.c    gss-serv.c       mkinstalldirs     readpass.c        ssh.1               ssh-keysign.8
auth2-jpake.c      cipher-bf1.c    gss-serv-krb5.c  moduli            regress           ssh1.h              ssh-keysign.c
auth2-kbdint.c     cipher.c        hostfile.c       moduli.c          RFC.nroff         ssh2.h              sshlogin.c
auth2-none.c       cipher-ctr.c    hostfile.h       monitor.c         rijndael.c        ssh-add             sshlogin.h
auth2-passwd.c     cipher.h        includes.h       monitor_fdpass.c  rijndael.h        ssh-add.0           ssh_prng_cmds.in
auth2-pubkey.c     cleanup.c       INSTALL          monitor_fdpass.h  rsa.c             ssh-add.1           sshpty.c
auth-bsdauth.c     clientloop.c    install-sh       monitor.h         rsa.h             ssh-add.c           sshpty.h
auth.c             clientloop.h    jpake.c          monitor_mm.c      scard             ssh-agent           ssh-rand-helper.0
auth-chall.c       compat.c        jpake.h          monitor_mm.h      scard.c           ssh-agent.0         ssh-rand-helper.8
authfd.c           compat.h        kex.c            monitor_wrap.c    scard.h           ssh-agent.1         ssh-rand-helper.c
authfd.h           compress.c      kexdh.c          monitor_wrap.h    scard-opensc.c    ssh-agent.c         ssh-rsa.c
authfile.c         compress.h      kexdhc.c         msg.c             schnorr.c         ssh.c               sshtty.c
authfile.h         config.guess    kexdhs.c         msg.h             scp               ssh_config          survey.sh.in
auth.h             config.h.in     kexgex.c         mux.c             scp.0             ssh_config.0        TODO
auth-krb5.c        config.sub      kexgexc.c        myproposal.h      scp.1             ssh_config.5        ttymodes.c
auth-options.c     configure       kexgexs.c        nchan2.ms         scp.c             sshconnect1.c       ttymodes.h
auth-options.h     configure.ac    kex.h            nchan.c           servconf.c        sshconnect2.c       uidswap.c
auth-pam.c         contrib         key.c            nchan.ms          servconf.h        sshconnect.c        uidswap.h
auth-pam.h         crc32.c         key.h            openbsd-compat    serverloop.c      sshconnect.h        umac.c
auth-passwd.c      crc32.h         lib              opensshd.init.in  serverloop.h      sshd                umac.h
auth-rhosts.c      CREDITS         LICENCE          openssh.xml.in    session.c         sshd.0              uuencode.c
auth-rh-rsa.c      deattack.c      log.c            OVERVIEW          session.h         sshd.8              uuencode.h
auth-rsa.c         deattack.h      log.h            packet.c          sftp              sshd.c              version.h
auth-shadow.c      Debug           loginrec.c       packet.h          sftp.0            sshd_config         WARNING.RNG
auth-sia.c         defines.h       loginrec.h       pathnames.h       sftp.1            sshd_config.0       xmalloc.c
auth-sia.h         dh.c            logintest.c      platform.c        sftp.c            sshd_config.5       xmalloc.h
erikpoll@laptop:~/.../src$
```

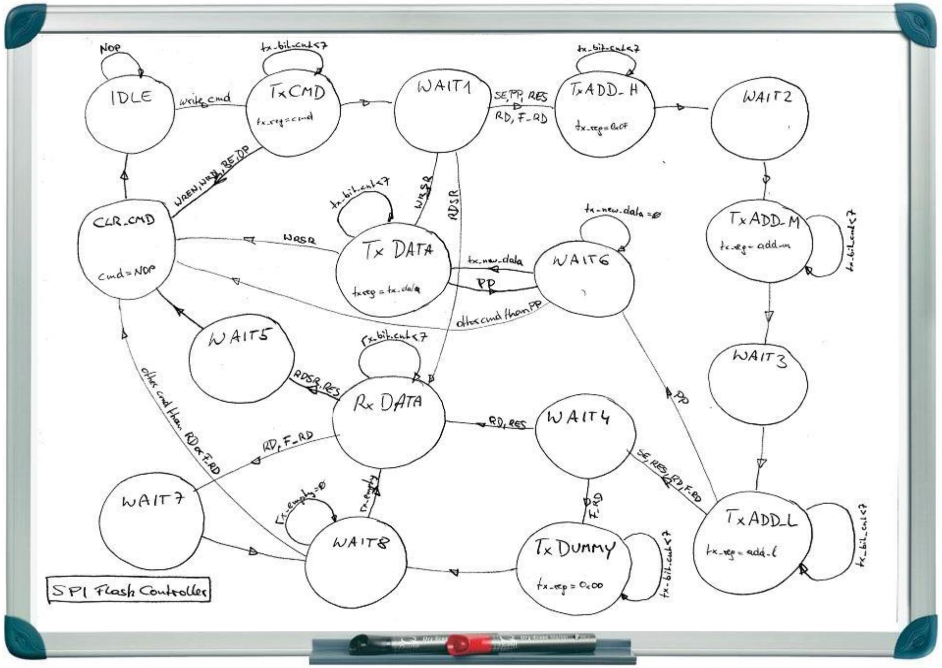
Typical implementation: openssh ☹

```
/** This array contains functions to handle protocol messages.
  * The type of the message is an index in this array. */
dispatch_fn *dispatch[255];
....
server_init_dispatch_20(void){
 dispatch_init(&dispatch_protocol_error);
 dispatch_set(SSH_MSG_CHANNEL_CLOSE, &channel_input_oclose);
 dispatch_set(SSH_MSG_CHANNEL_DATA, &channel_input_data);
 dispatch_set(SSH_MSG_CHANNEL_EOF, &channel_input_ieof);
 dispatch_set(SSH_MSG_CHANNEL_EXTENDED_DATA, &channel_input_extended_
 dispatch_set(SSH_MSG_CHANNEL_OPEN, &server_input_channel_open);
 dispatch_set(SSH_MSG_CHANNEL_OPEN_FAILURE, &channel_input_open_failu
 dispatch_set(SSH_MSG_CHANNEL_REQUEST, &server_input_channel_req);
 dispatch_set(SSH_MSG_GLOBAL_REQUEST, &server_input_global_request);
 dispatch_set(SSH_MSG_KEXINIT, &kex_input_kexinit);
```

Understanding protocol state machine from code is hard!

LangSec also for session languages!

Protocol state machines deserve to be explicitly specified



Extracting protocol state machine from code

We can infer a finite state machine from implementation

by black box testing using state machine 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

This is a great way to obtain protocol state machine

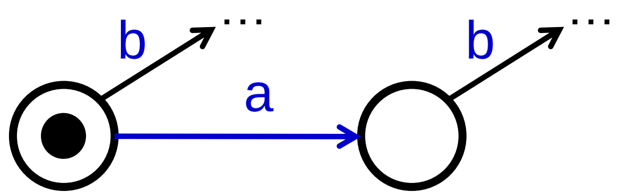
- without reading specs!
- without reading code!

State machine learning with L*

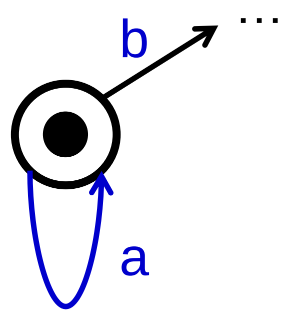
Basic idea: compare response of a deterministic system to different input sequences, eg.

1. b
2. a ; b

If response is different, then



otherwise

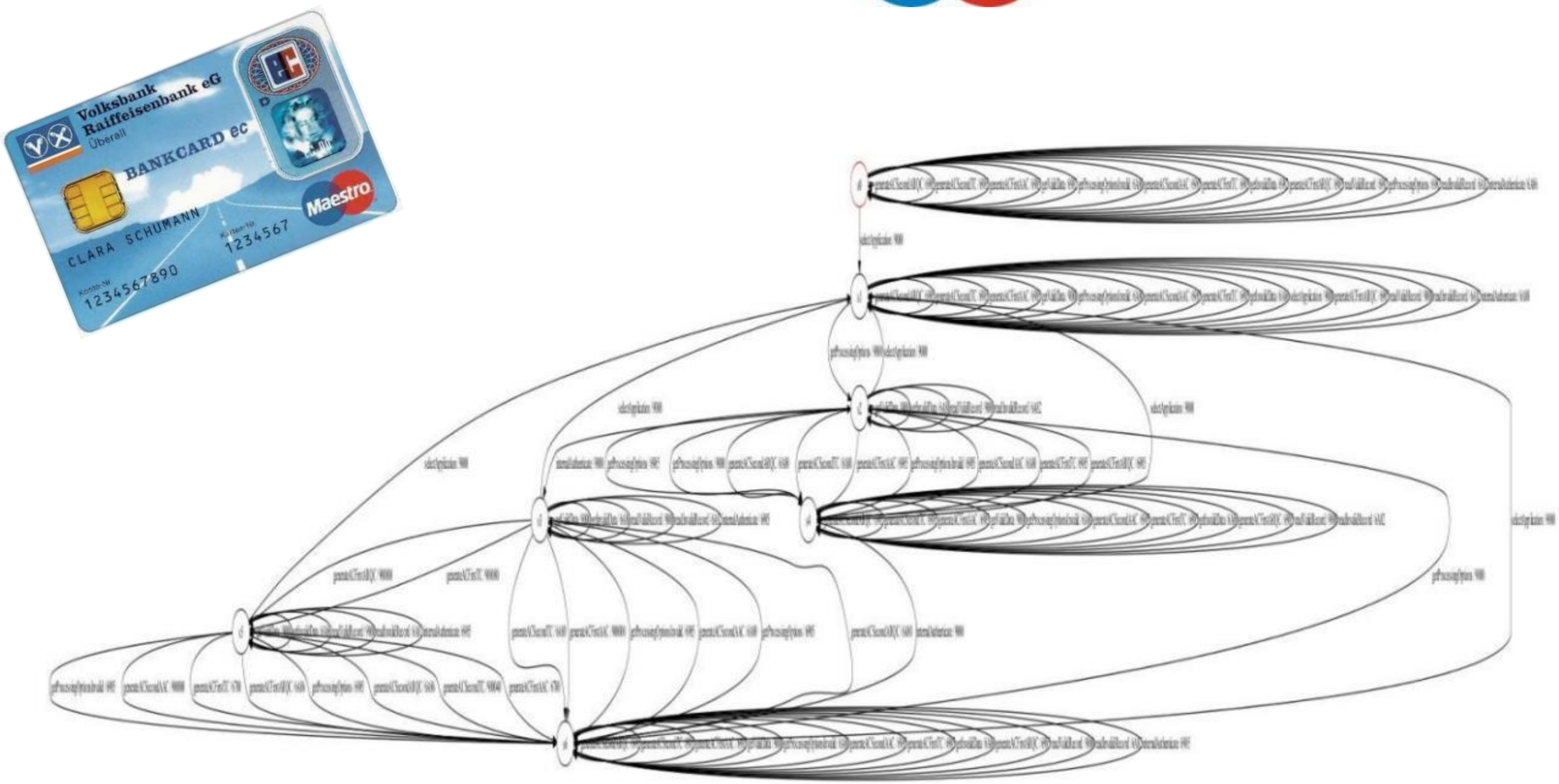


The state machine inferred is only an approximation of the system, and only as good as your set of test messages.

Case study: EMV

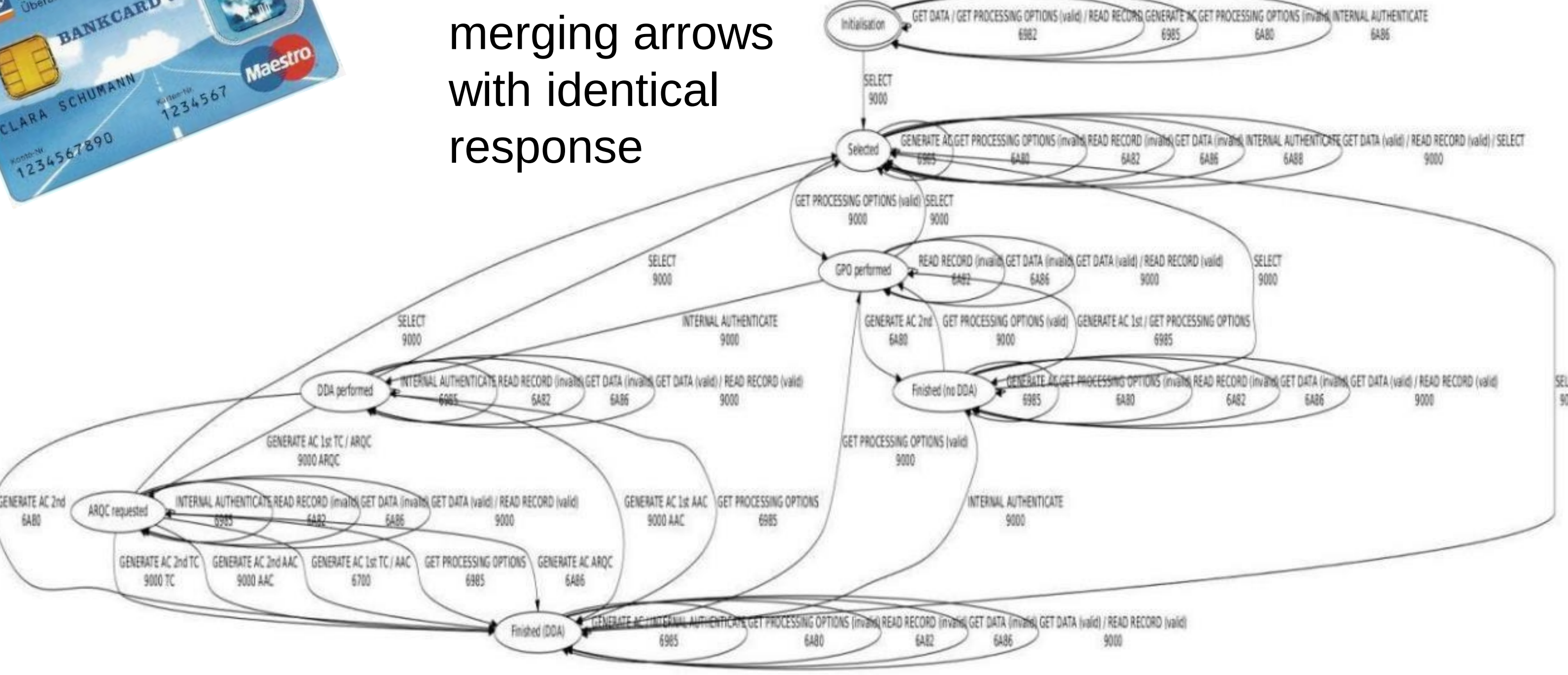
- Most banking smartcards implement a variant of EMV
- EMV (Europay-Mastercard-Visa) defines set of protocols with *lots* of variants
- Specification in 4 books totalling > 700 pages

State machine learning of Maestro card

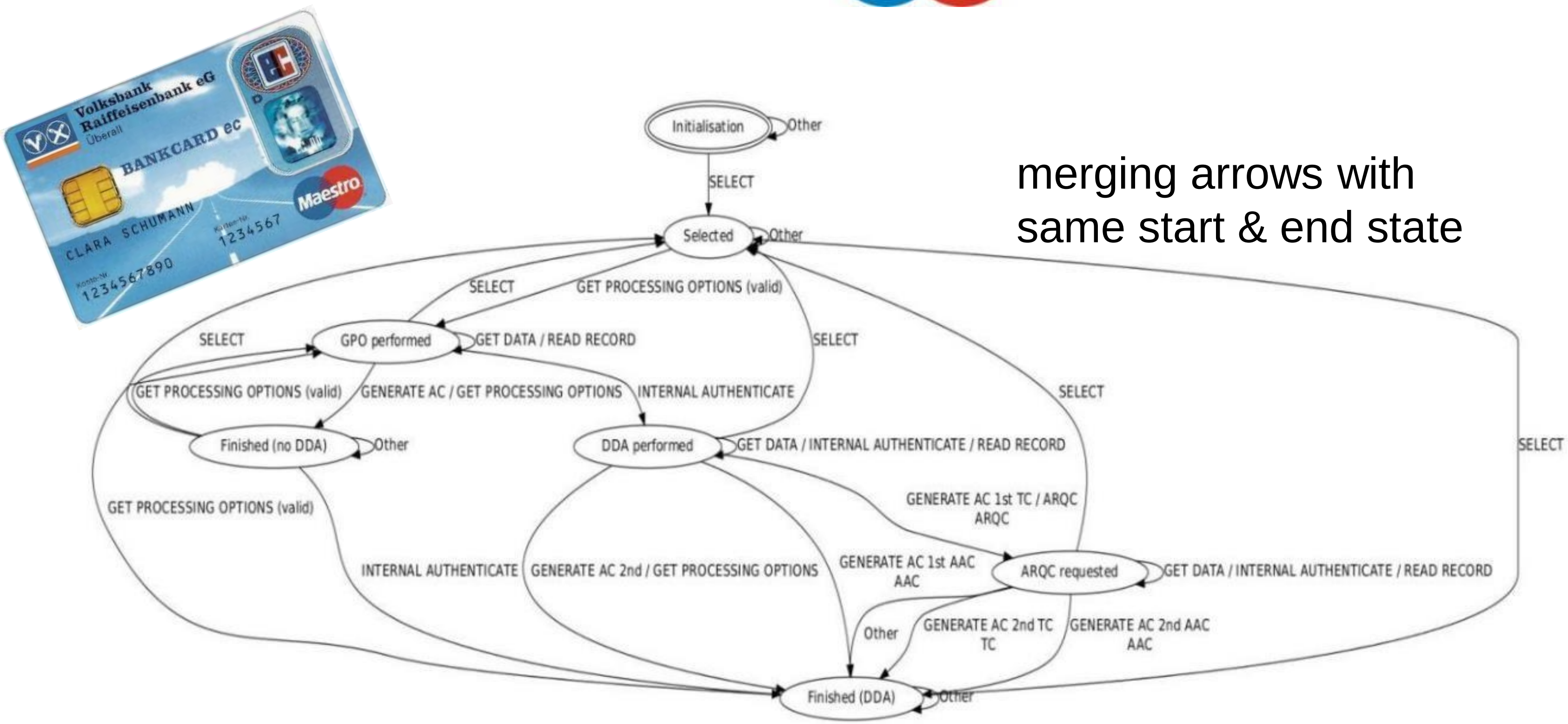


State machine learning of Maestro card

merging arrows
with identical
response



State machine learning of Maestro card



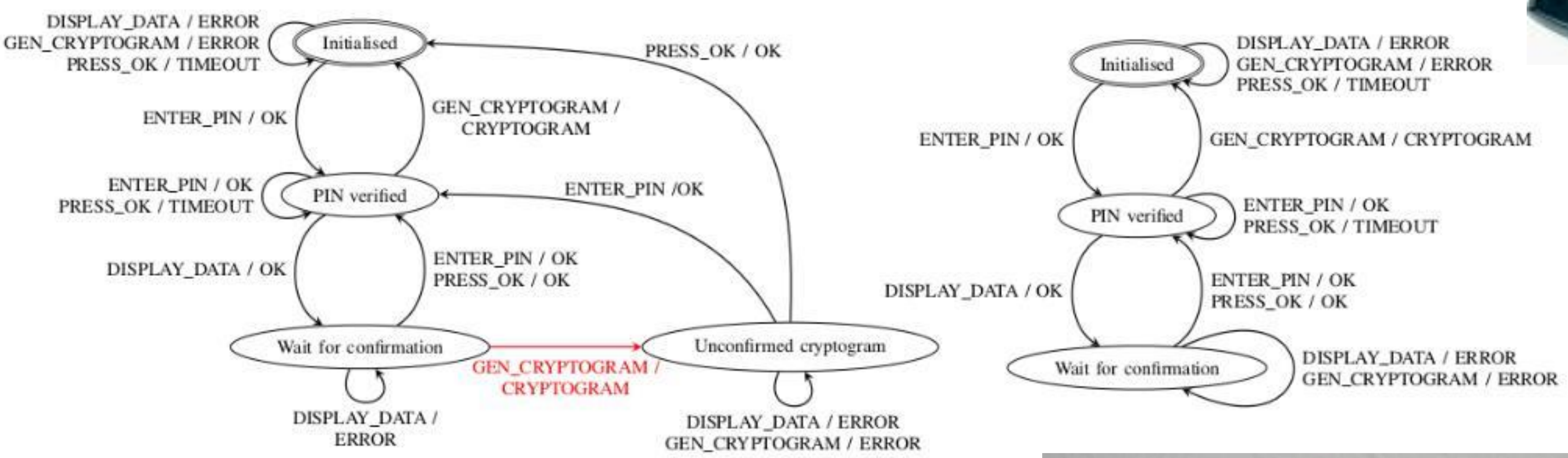
merging arrows with
same start & end state

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

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

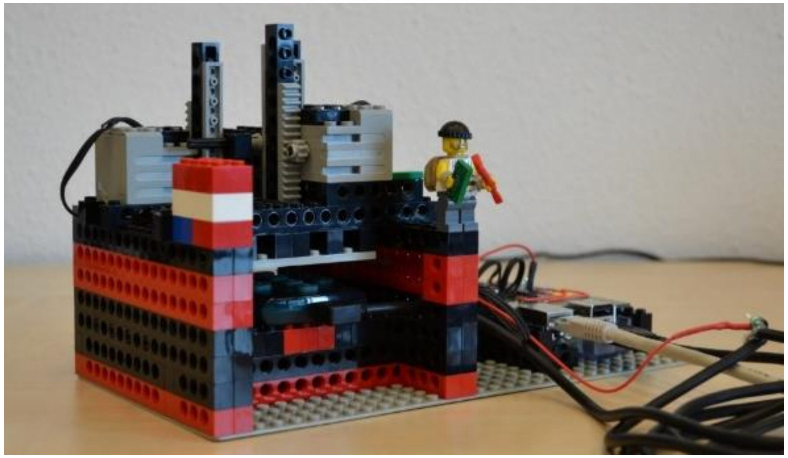
State machine learning of internet banking device

State machines inferred for flawed & patched device



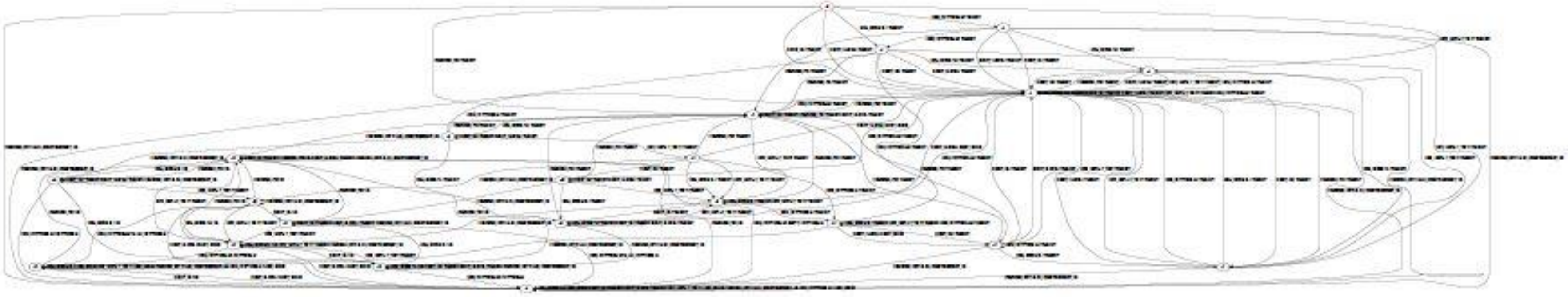
[Georg Chalupar et al.,
Automated reverse engineering using Lego,
WOOT 2014]

Movie at http://tinyurl/legolearn



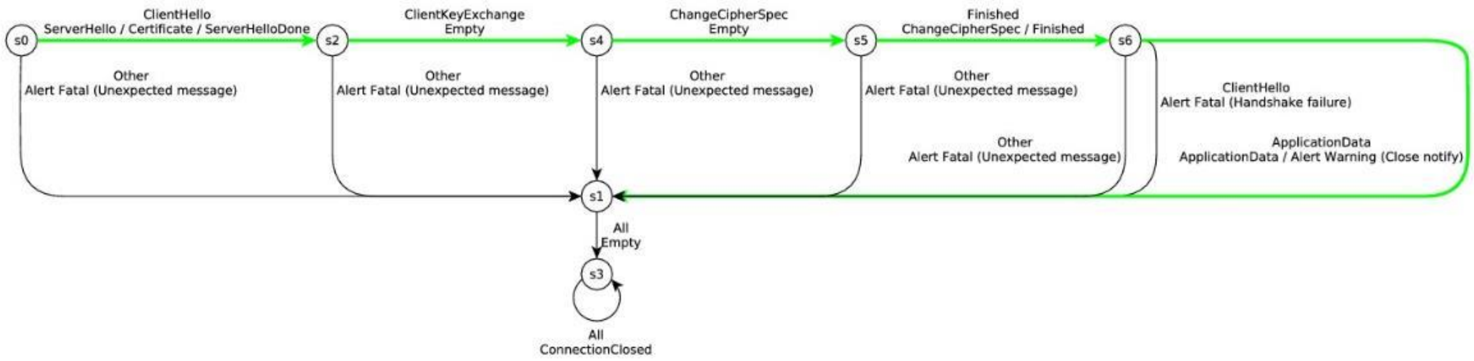
Scary state machine complexity

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



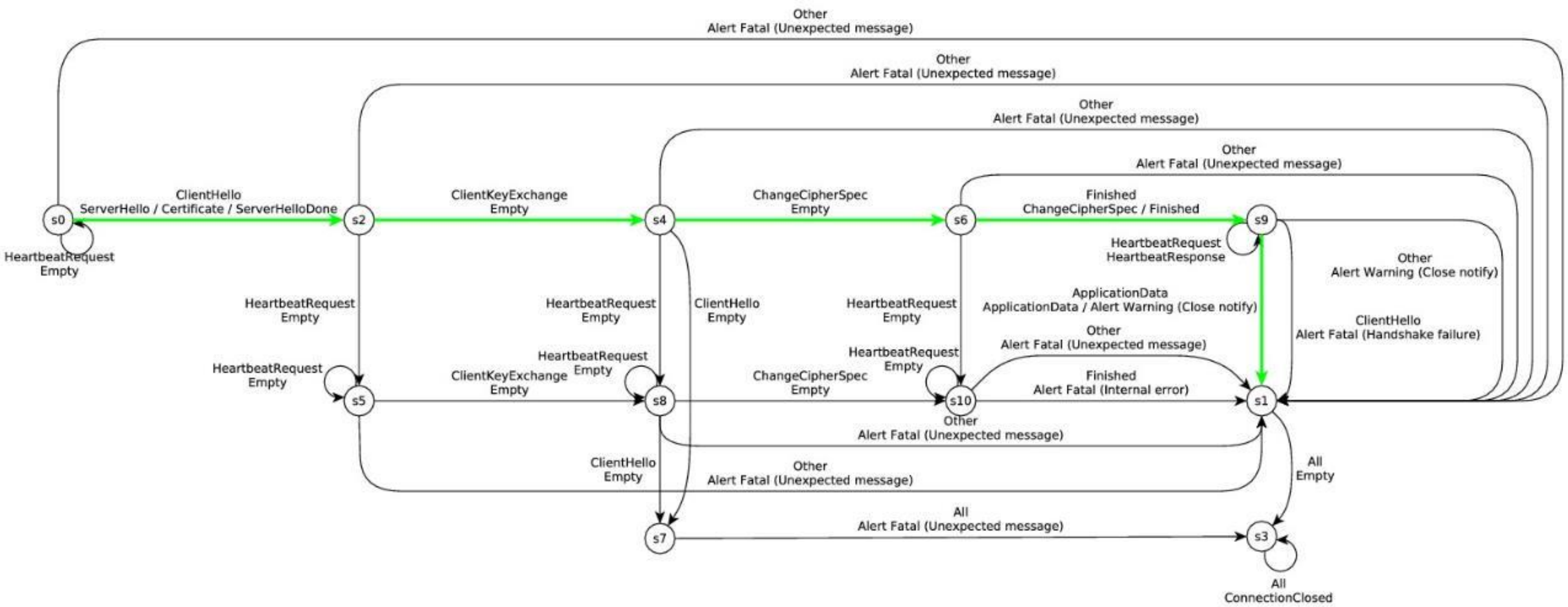
No flaws found in patched device, but were the developers really confident that this complex behaviour is secure?
Or necessary?

TLS state machine extracted from NSS

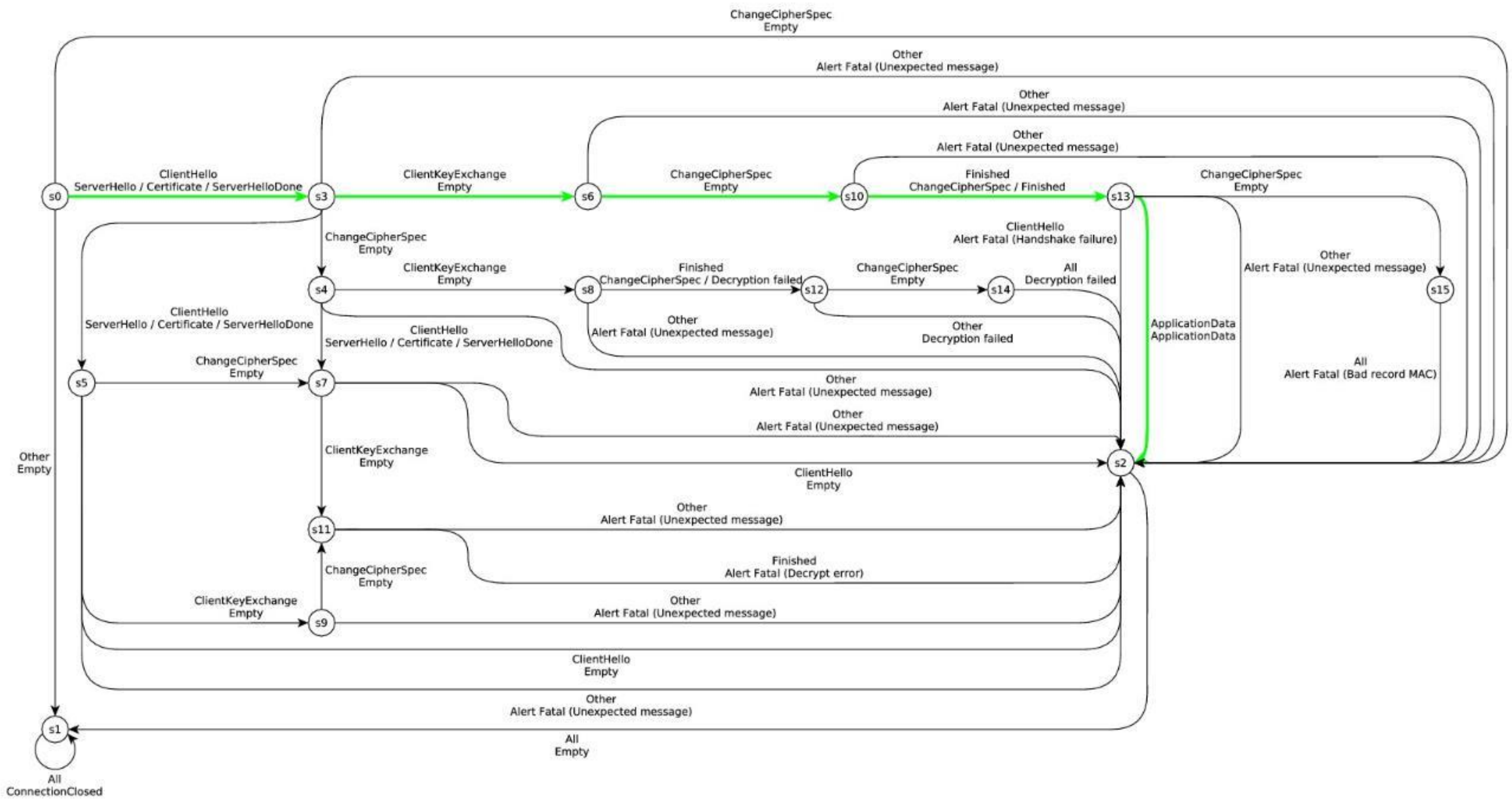


Comforting to see this is so simple!

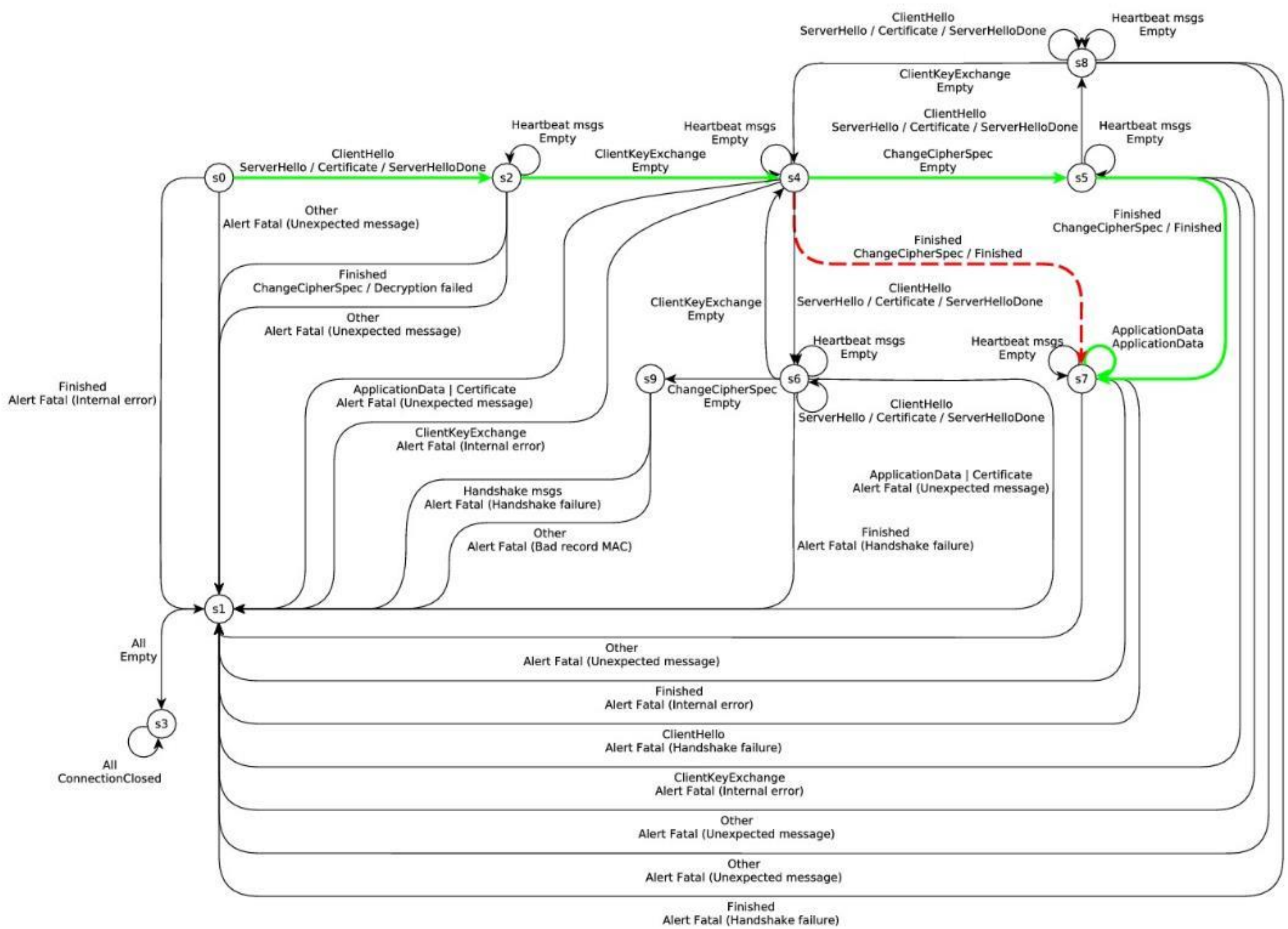
TLS state machine extracted from GnuTLS



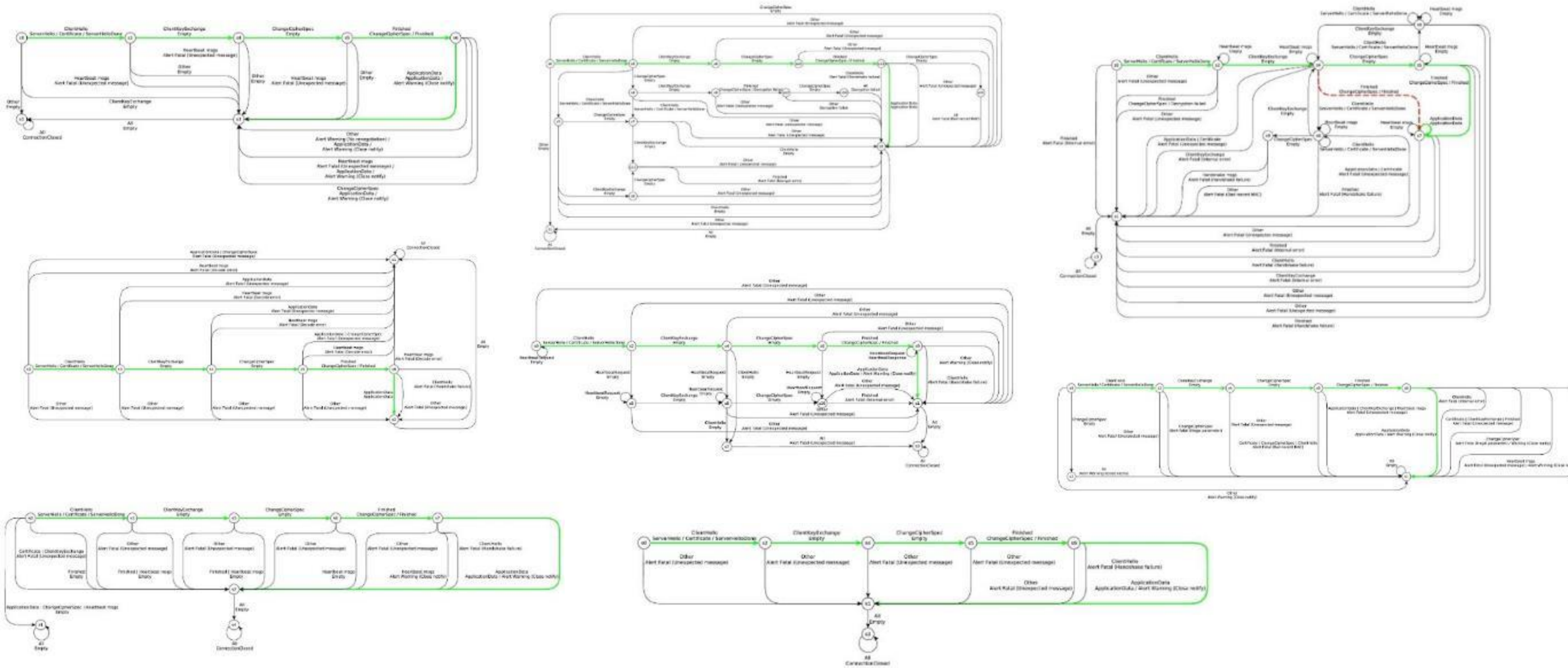
TLS state machine extracted from OpenSSL



TLS state machine extracted from JSSE



Which TLS implementations are correct? or secure?



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

Conclusions

LangSec principles not only apply to language of *input messages*
but also for language of *protocol sessions*
because in practice we see

- *unclear specifications* of session languages

  without explicit state machines
- *messy & flawed implementations* of session languages
- *security flaws* as a result of this

Open question: How common is this category of security flaws?

Comparing session languages to message formats

Bad news

1. even less likely to be rigorously specified
   - many specs provide EBNF but no protocol state machine
2. complete specification of state machine is tricky
   - input-enabled state machine becomes messy
3. generating code from spec is harder
   - handling state has to be interpersed with other functionality (cf. aspect)

Good news

1. we can extract state machines from code!

   to find flaws in program logic, but not malicious backdoors
2. bugs in state machine can cause security problems, but no weird machines?


NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES