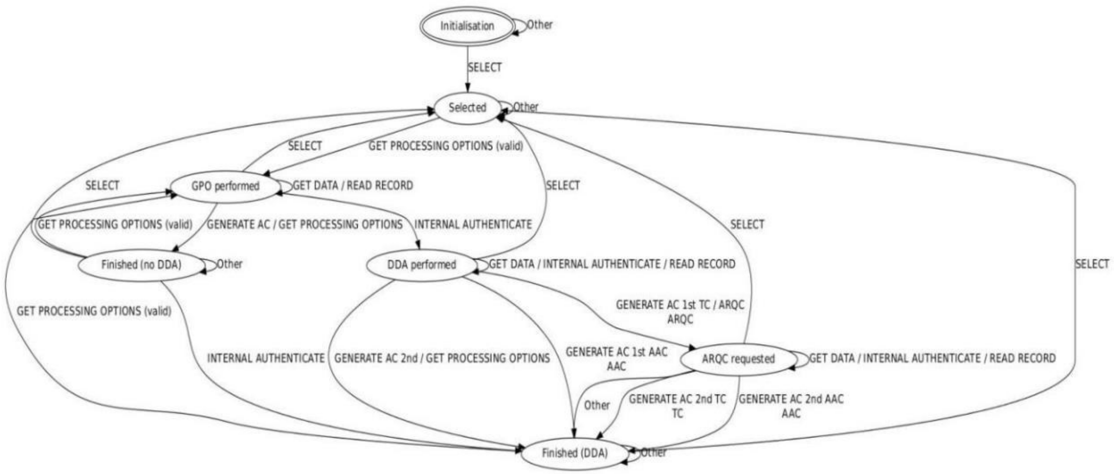
State machine learning
&
Formal Methods

Erik Poll

Digital Security

Radboud University Nijmegen

Radboud Universiteit Nijmegen



State Machine Learning

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

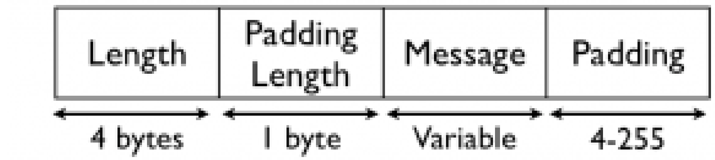
Which systems are harder to test (or fuzz): stateless or stateful systems?

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

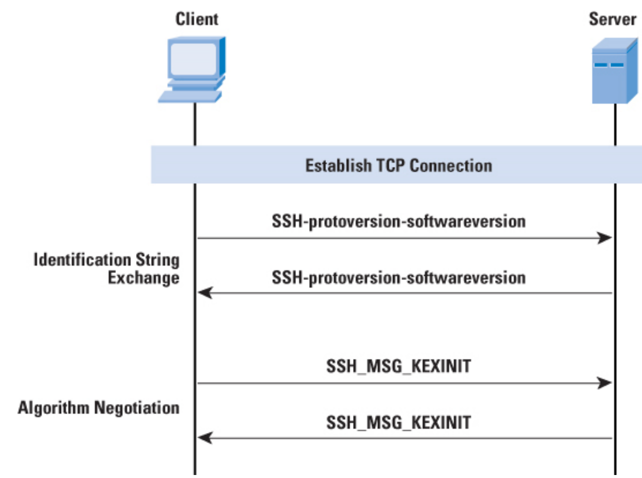
Protocols

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
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_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`	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*

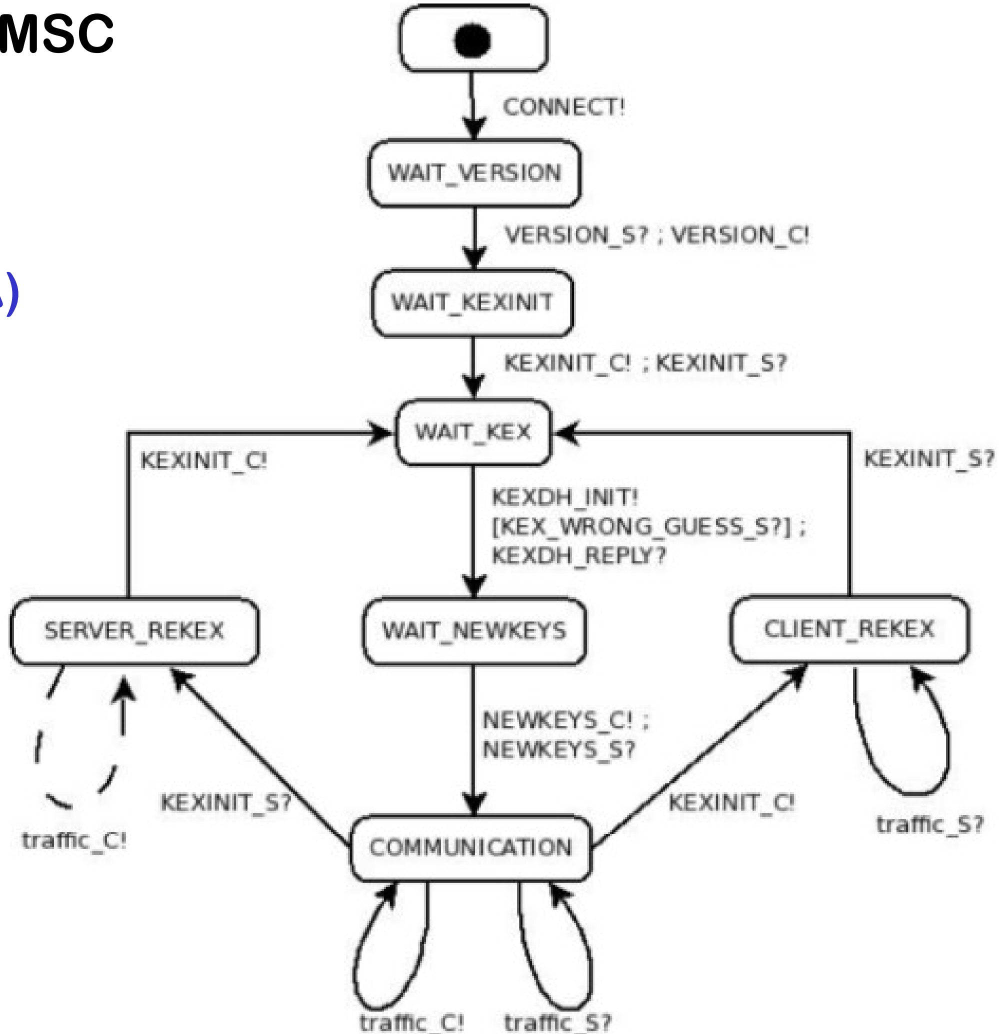
Protocol state machines

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

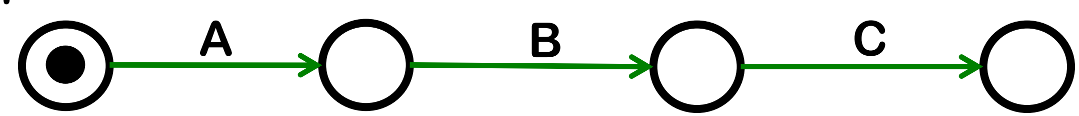
1) some error state,

2) back to the initial state, or

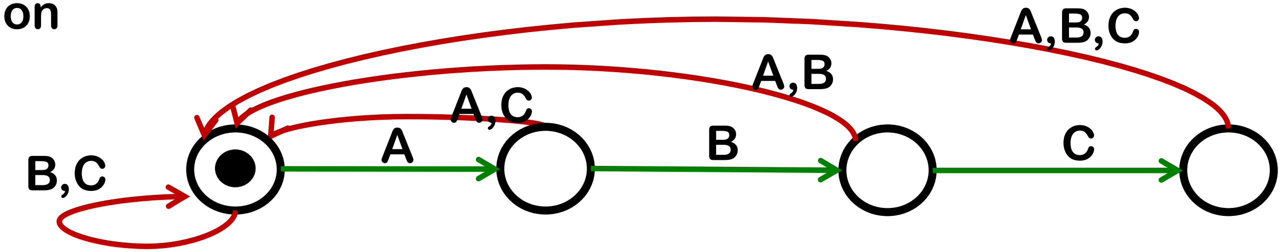
3) are ignored

input enabling

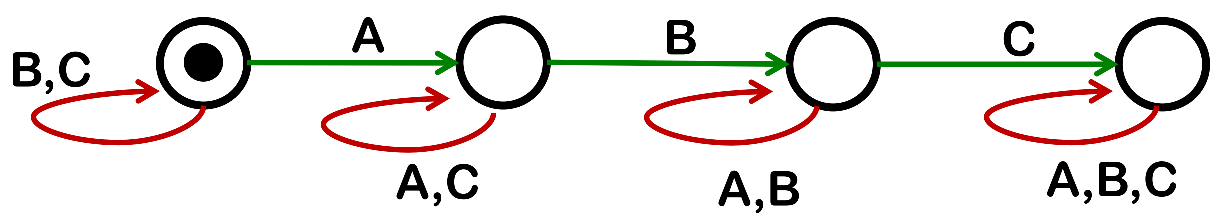
State machine that is not input-enabled



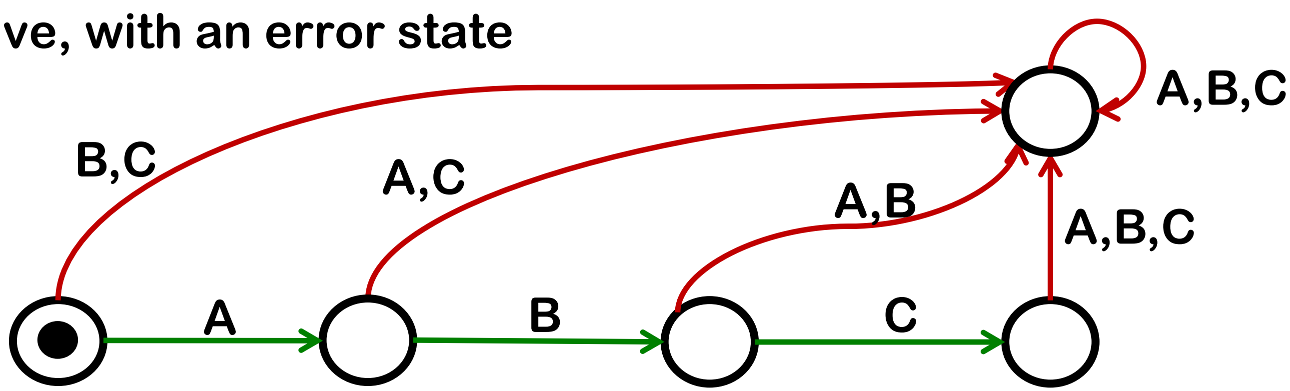
Input enabled version



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

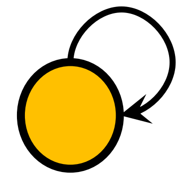
libssh versions 0.6 and above have an authentication bypass vulnerability in the server code. By presenting the server an SSH2_MSG_USERAUTH_*SUCCESS* message in place of the SSH2_MSG_USERAUTH_*REQUEST* message which the server would expect to initiate authentication, the attacker could successfully authenticate 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



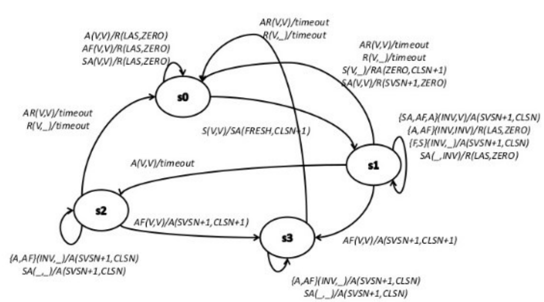
[Verifying an implementation of SSH, WIST 2007]

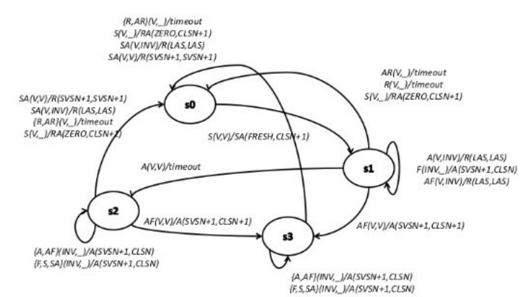
- **e.dentifier2**

  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

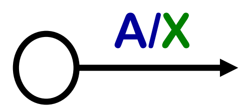
It is a great way to obtain protocol state machines

- without reading specs!
- without reading code!

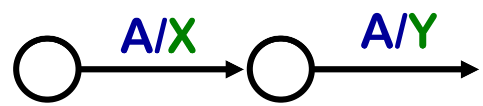
State machine inference, eg using LearnLib

Just try out many sequences of inputs, and observe outputs

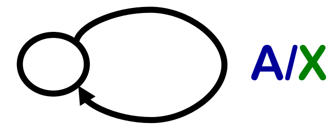
Suppose input A results in output X



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

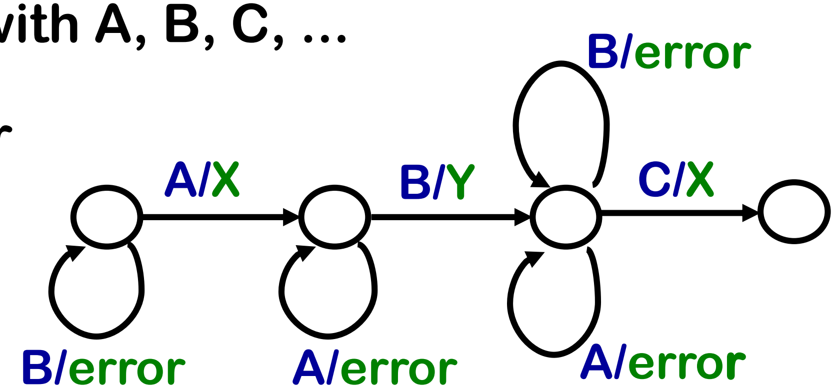


- If second input A results in the *same* output X



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

to e.g. infer



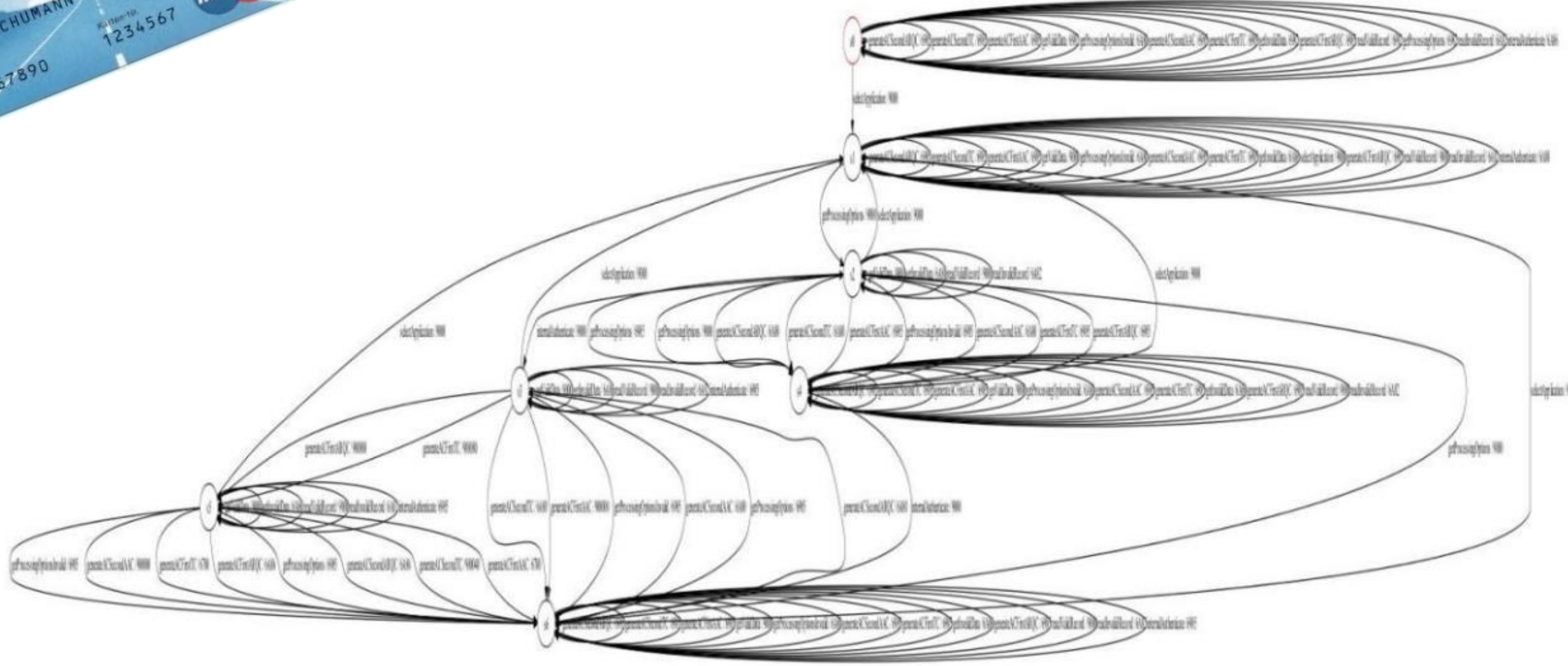
The inferred state machine is an *under*-approximation of real system

Case study 1: EMV

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

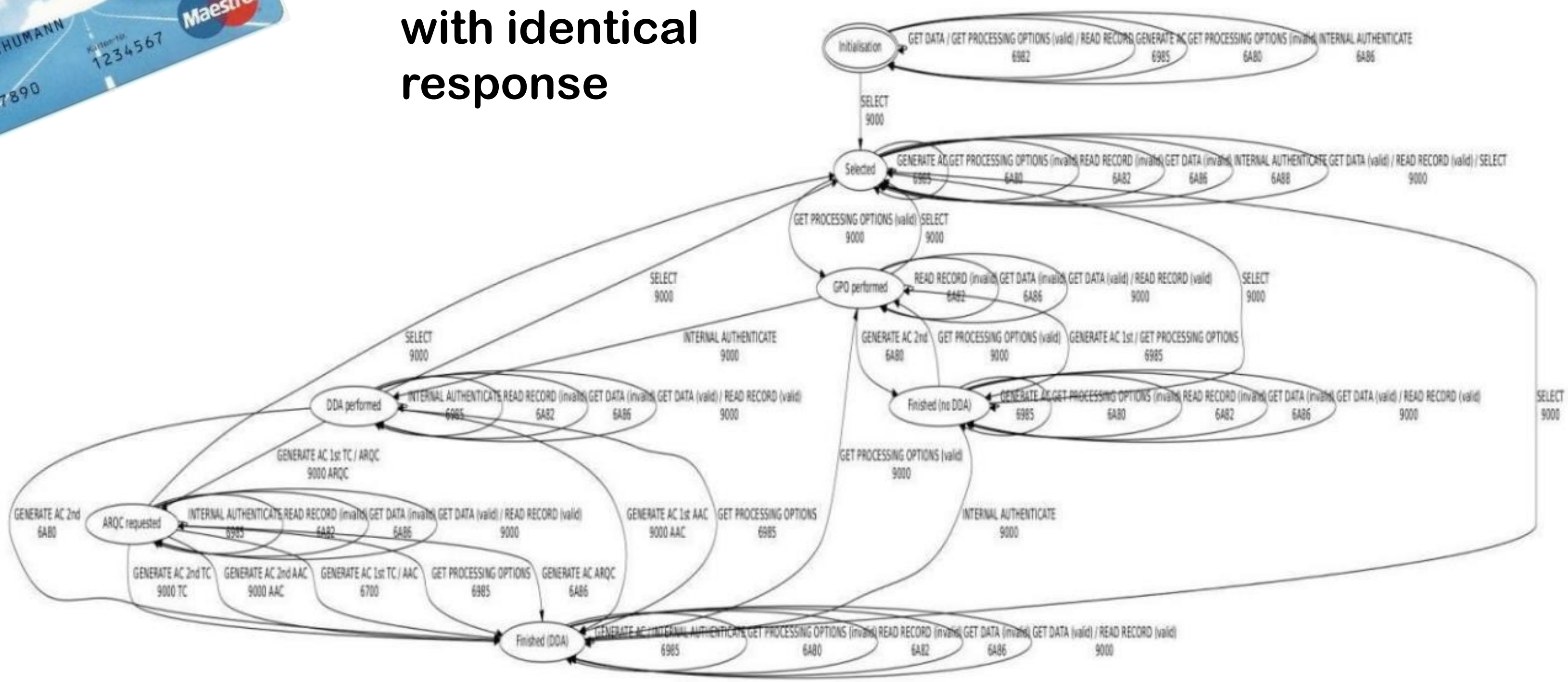
- **Specs controlled by EMVCo which is owned by MasterCard, VISA, American Express, JCB**
- **Specification in 4 books totalling > 700 pages**
- **EMV contactless specs: 10 more books, > 1500 pages**

State machine inference of Maestro card

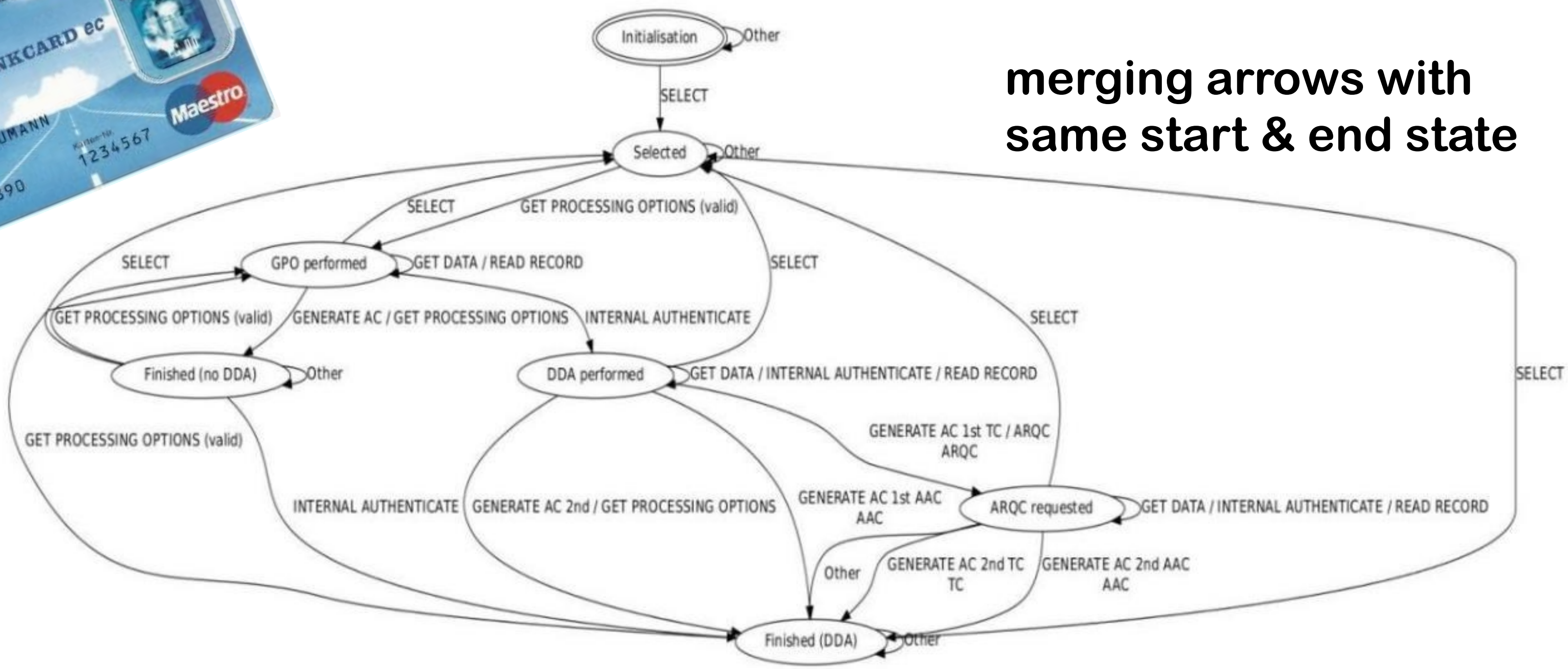


State machine inference of Maestro card

merging arrows
with identical
response



State machine inference of Maestro card



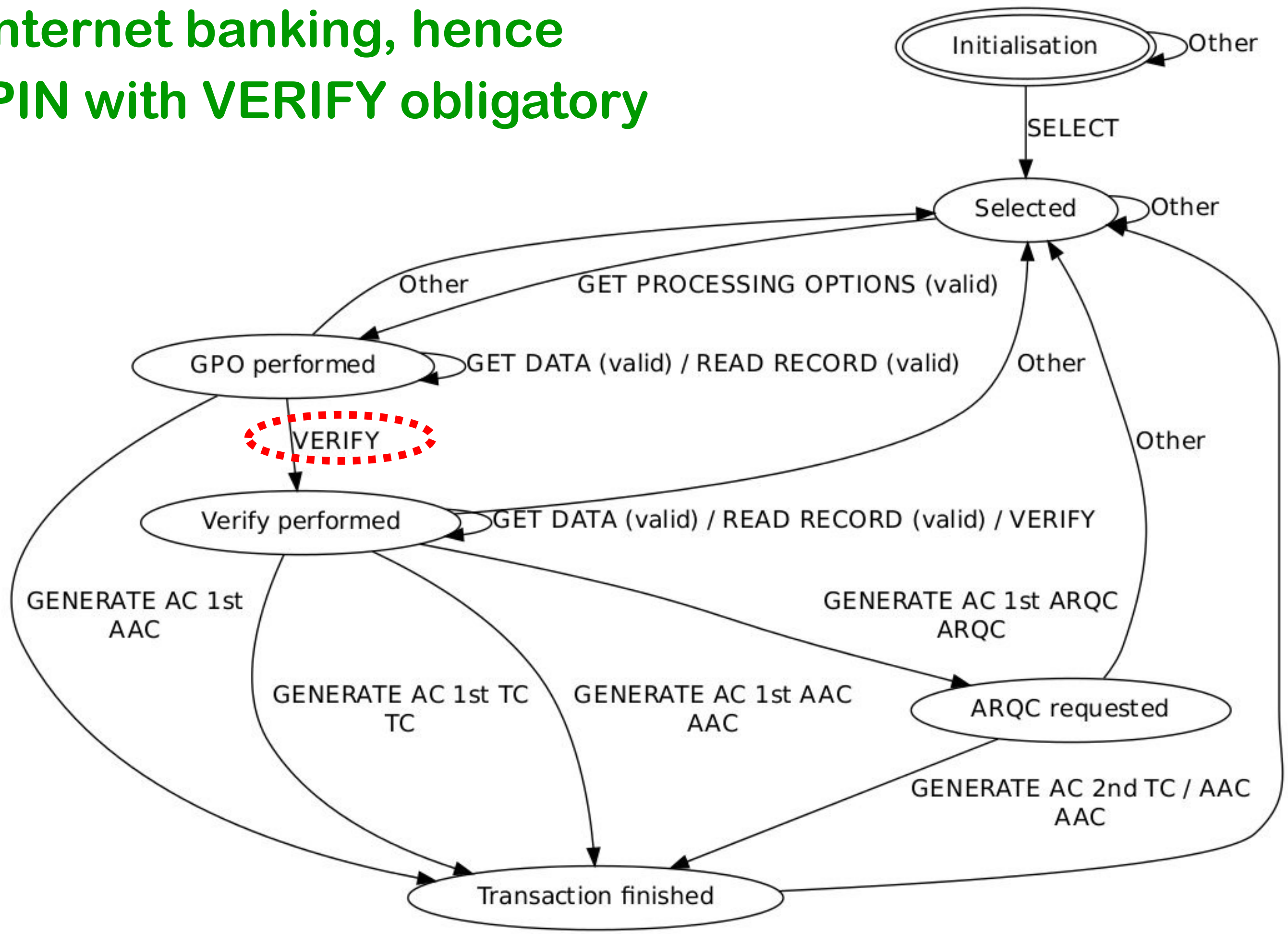
merging arrows with same start & end state

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

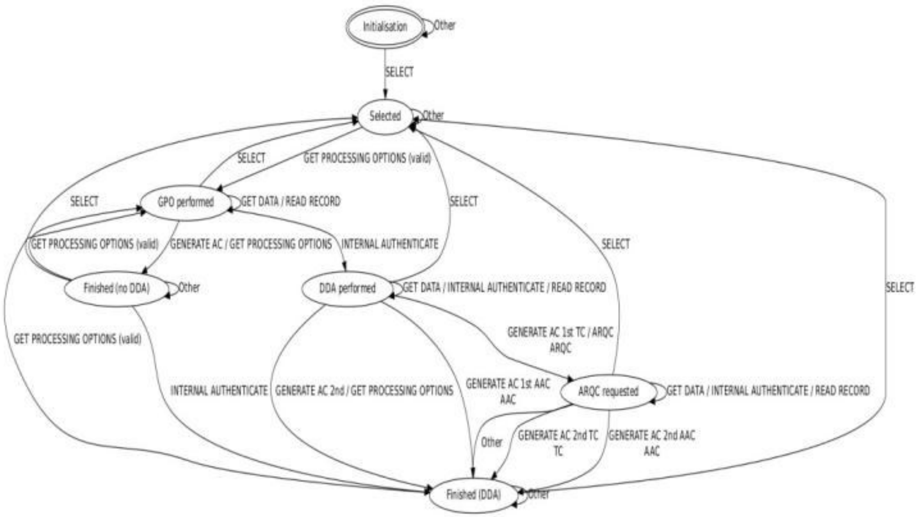
[Fides Aarts et al., Formal models of bank cards for free, SECTEST 2013]

SecureCode application on Rabobank card

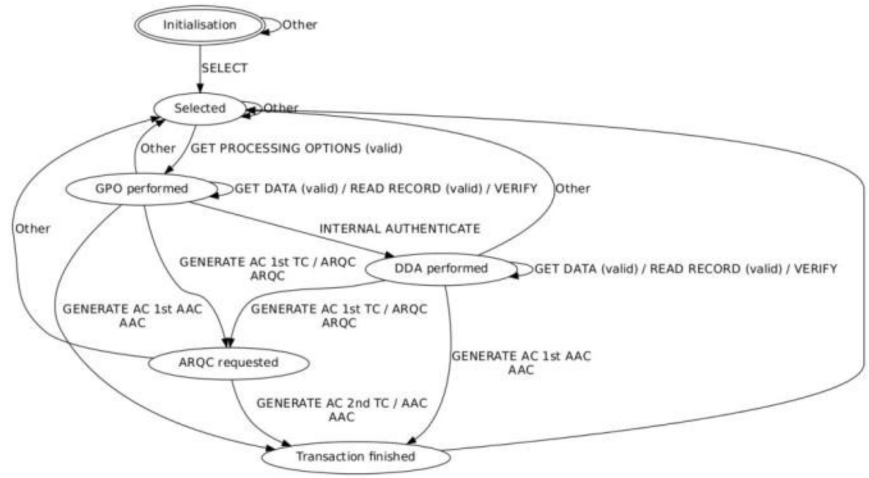
used for internet banking, hence entering PIN with VERIFY obligatory



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?

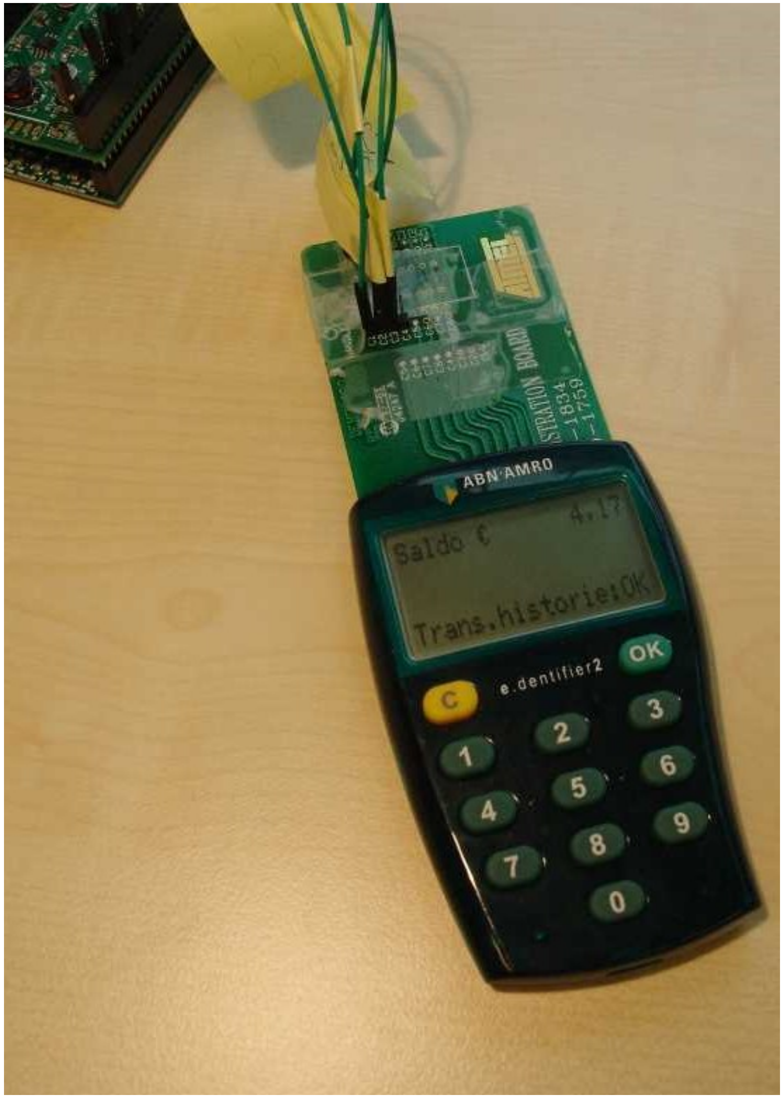
Case study 2: the USB-connected e.dentifier

Can we use state machine learning with

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

to obtain the state machine

of the ABN-AMRO e.dentifier2?

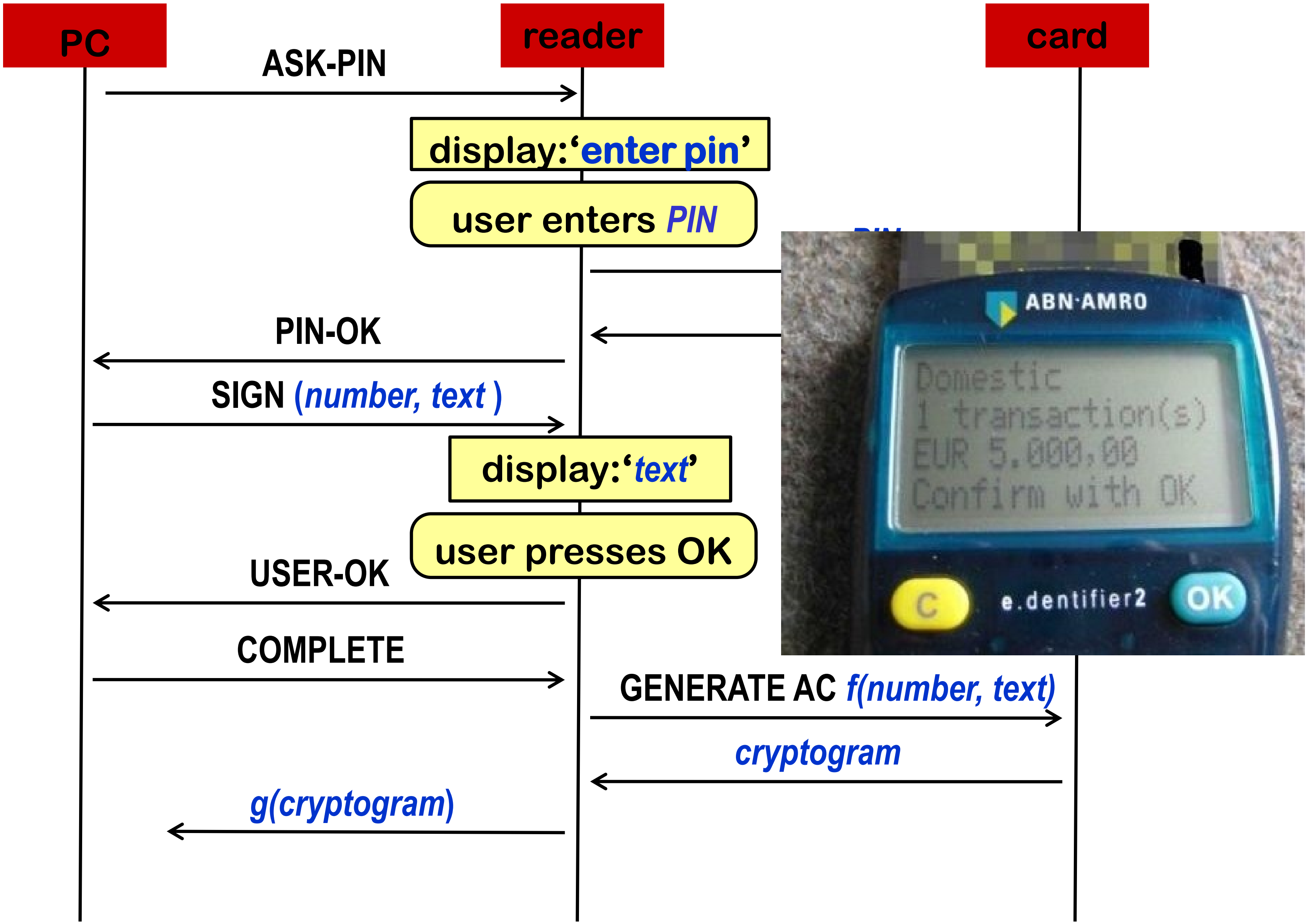


Earlier manual analysis

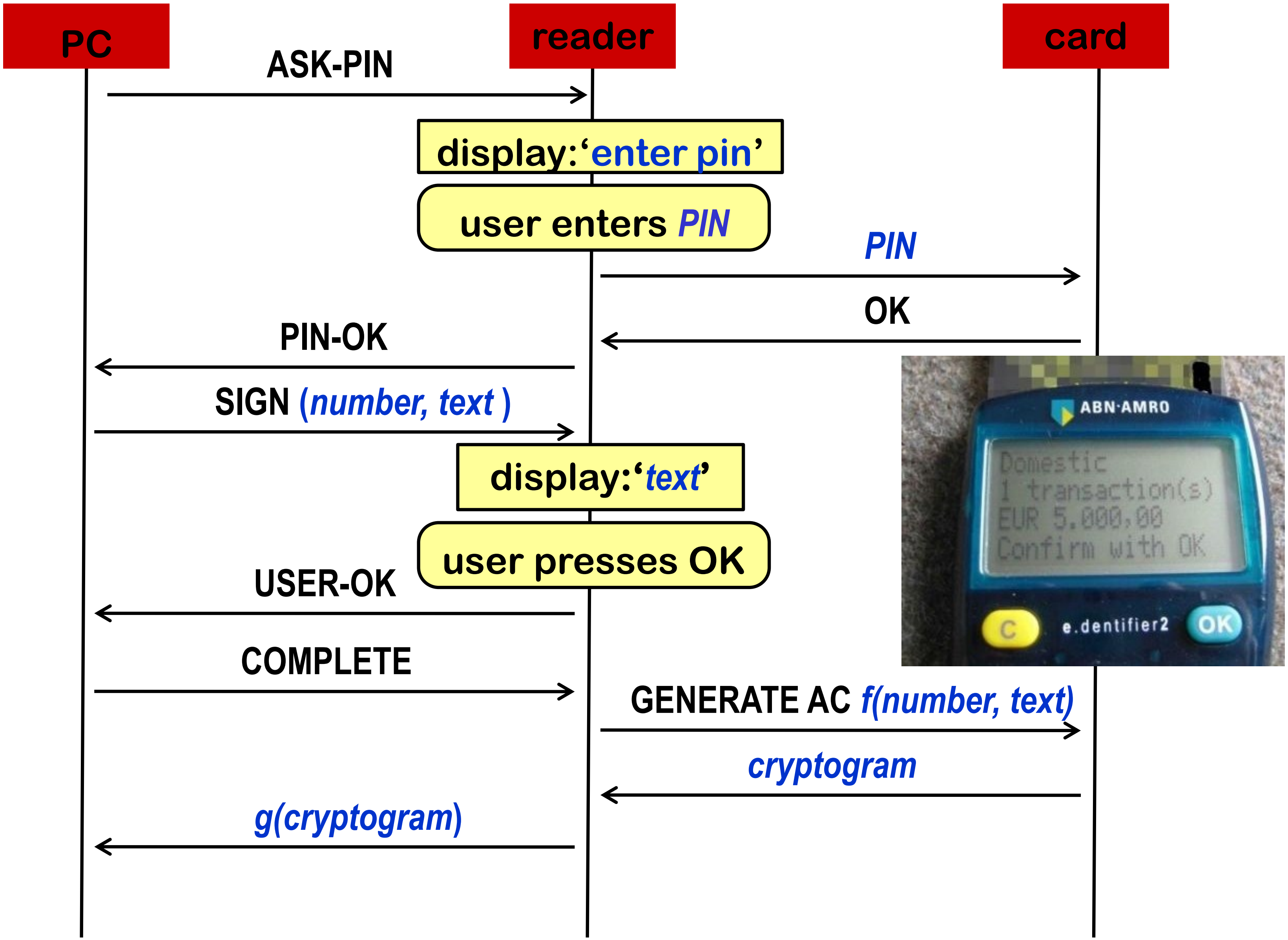
revealed the USB connection

has a flaw

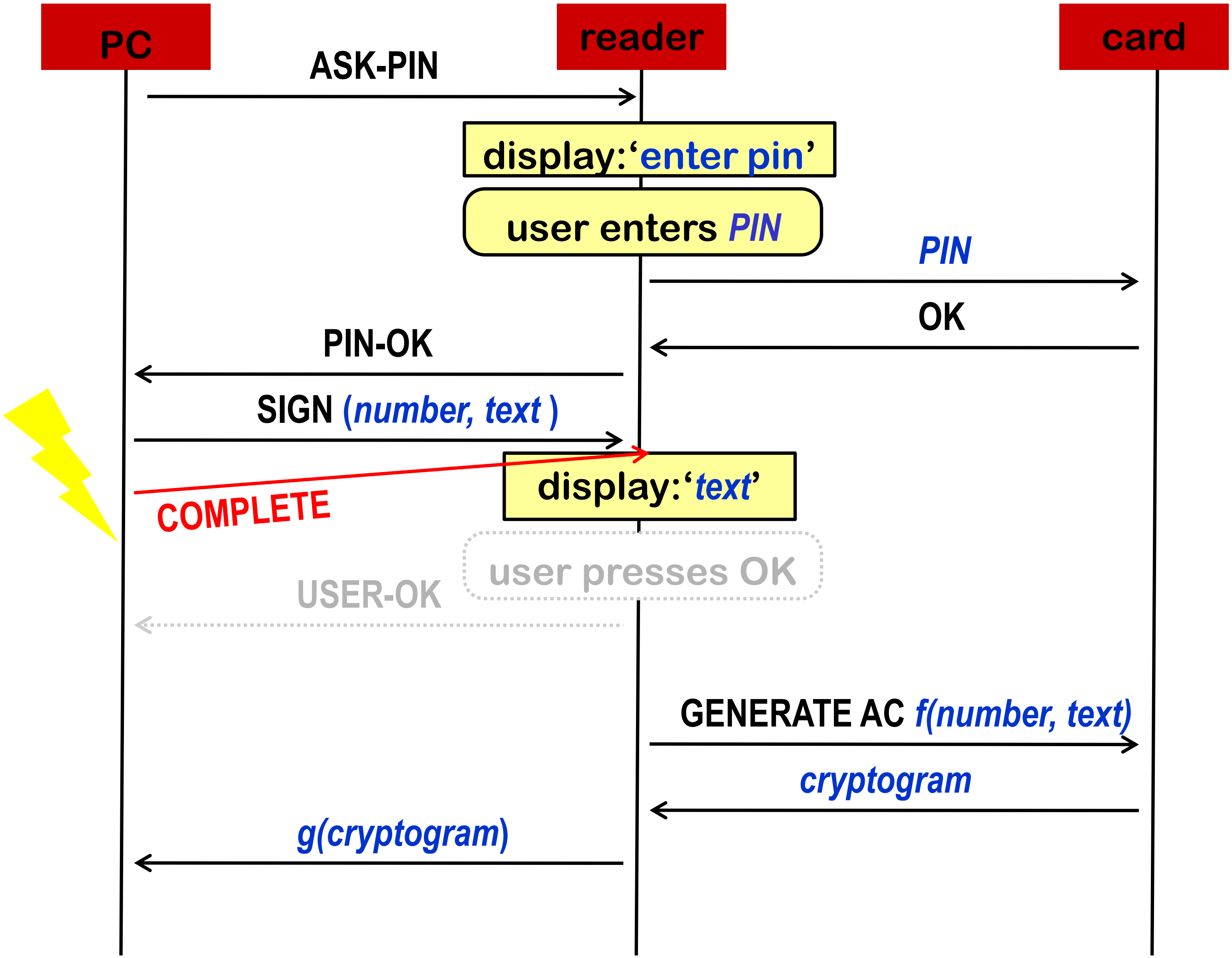
(Manually) reverse-engineered Protocol



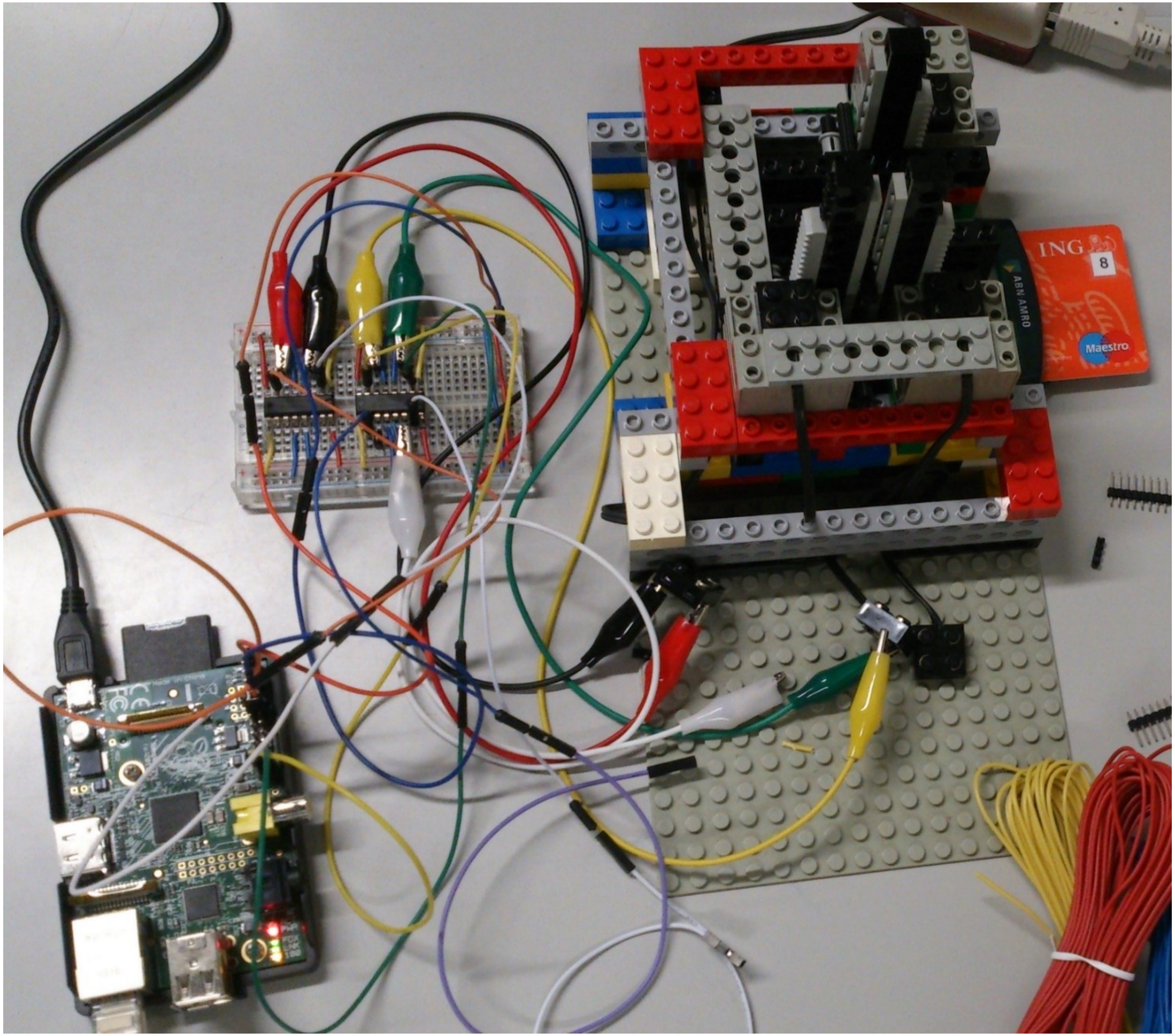
Spot the defect!

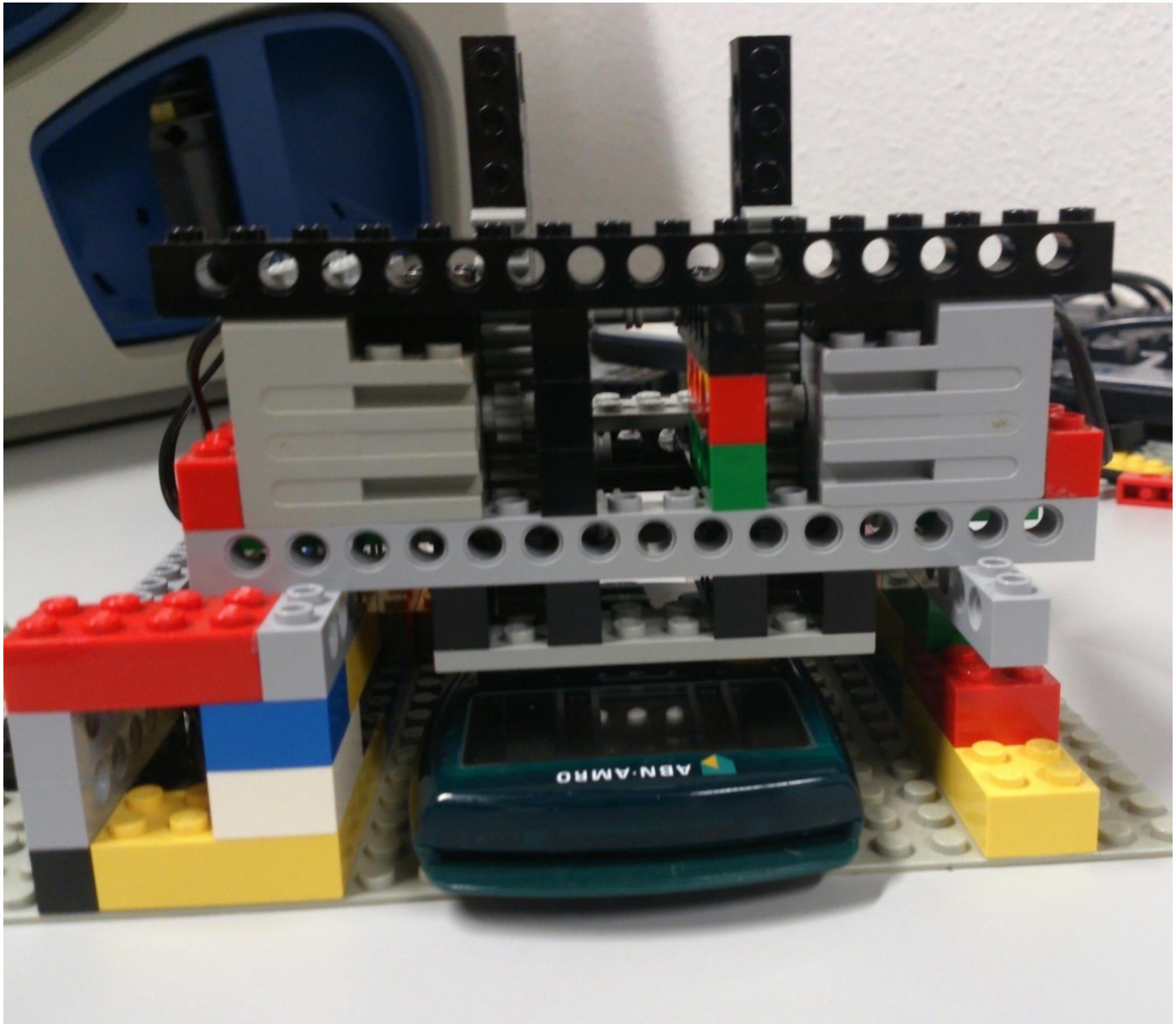


Attack!


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)

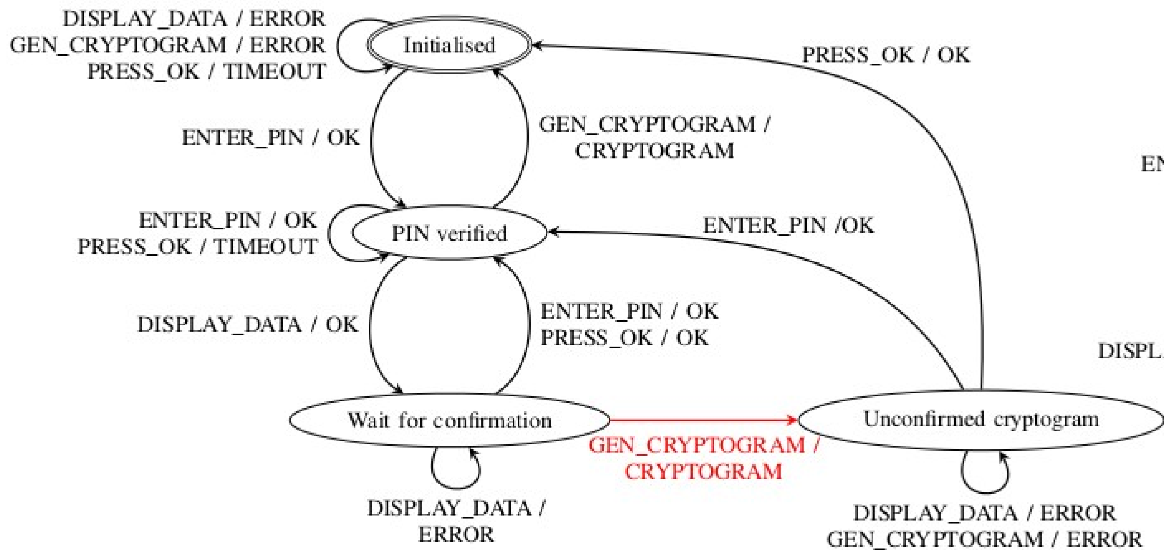
Operating the keyboard using LEGO

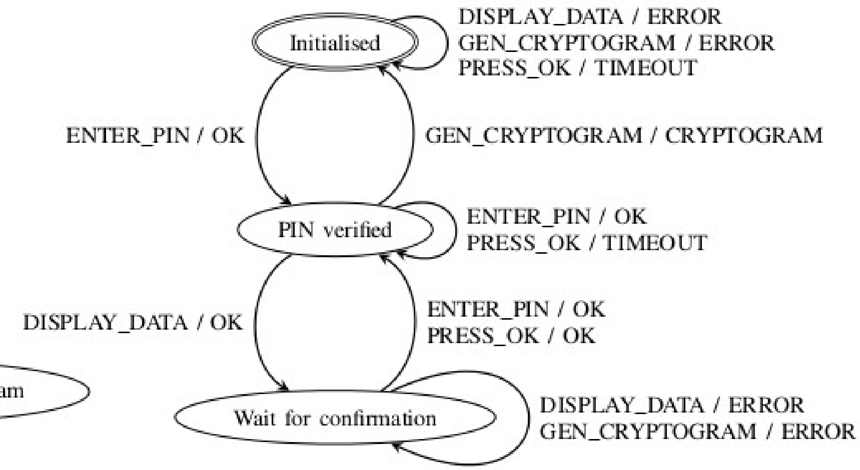

ING
8
ABN AMRO
Maestro

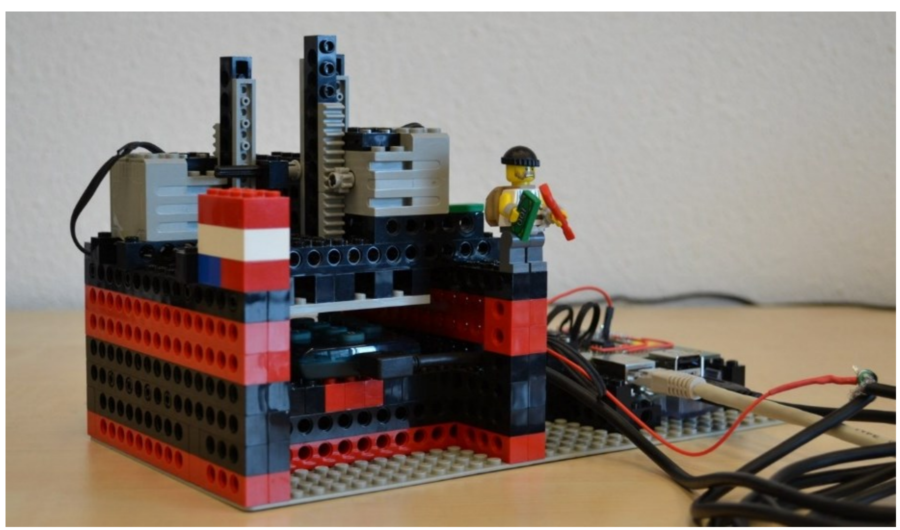


https://www.youtube.com/watch?v=hyQubPvAyq4

State machines of old vs new e.dentifier2

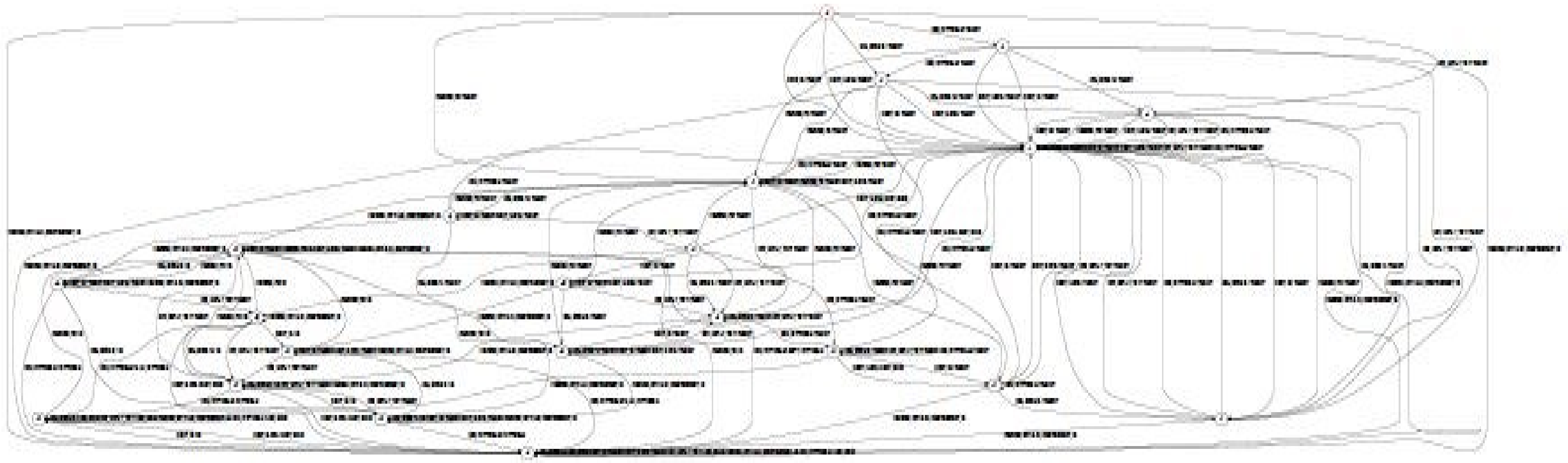






https://www.youtube.com/watch?v=hyQubPvAyq4

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?

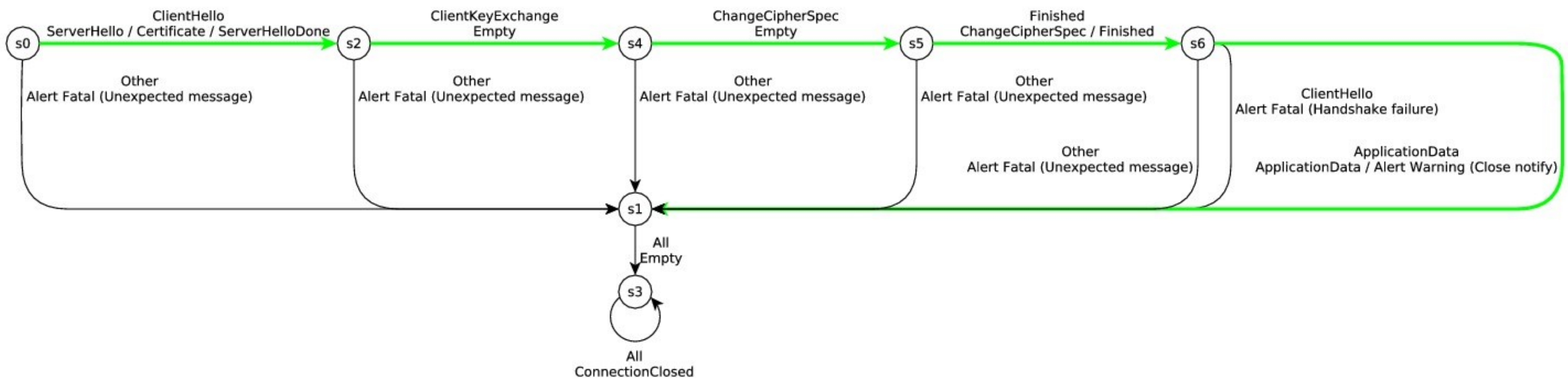
Results with learning state machines for e.dentifier2

- 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

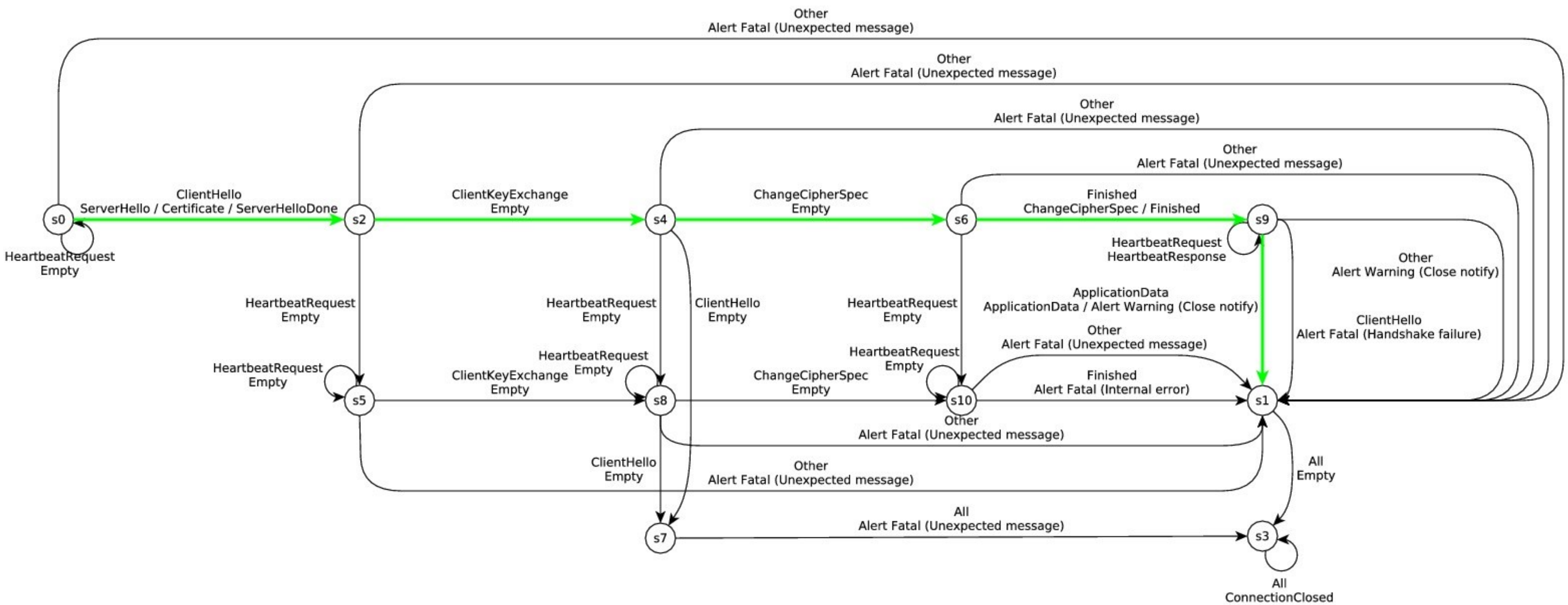
Case study 3: TLS



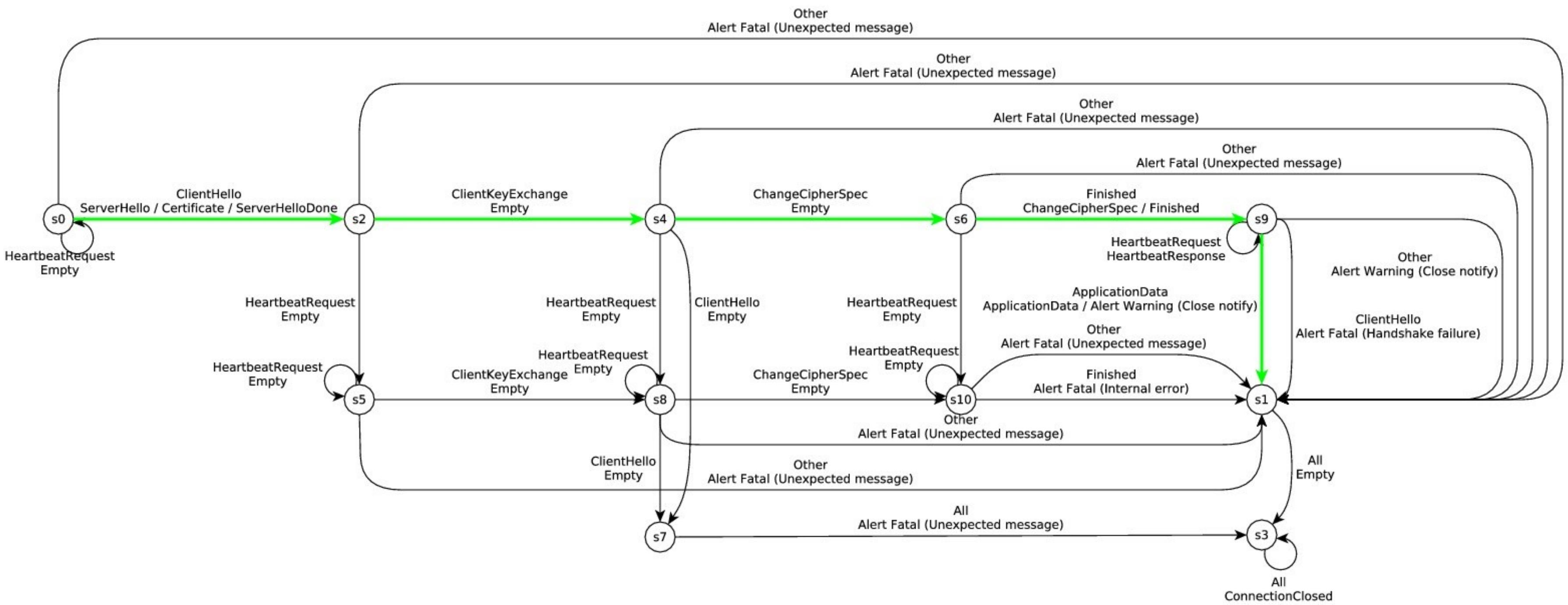
State machine inferred from NSS implementation

Comforting to see this is so simple!

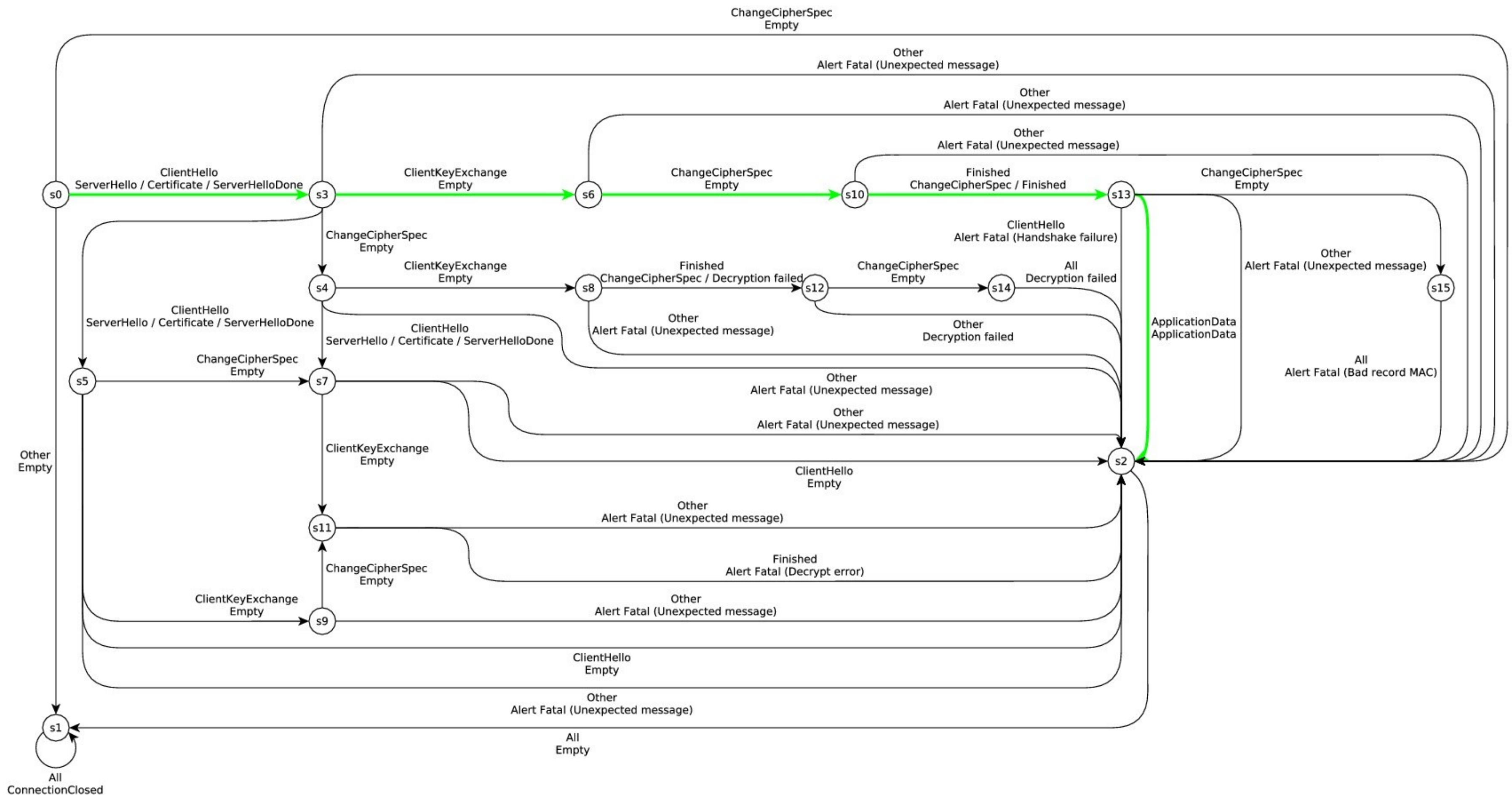
TLS... according to GnuTLS



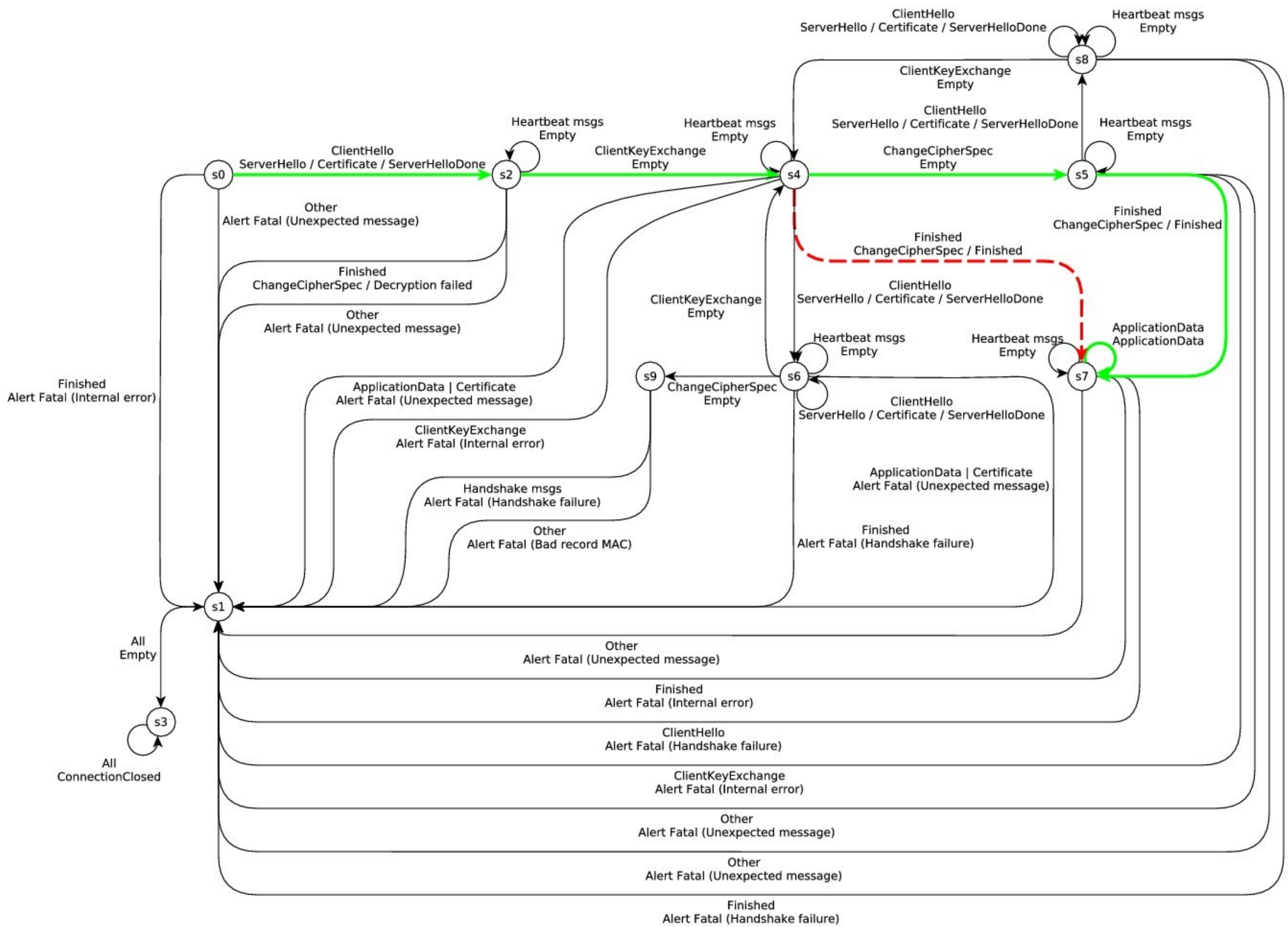
TLS... according to GnuTLS



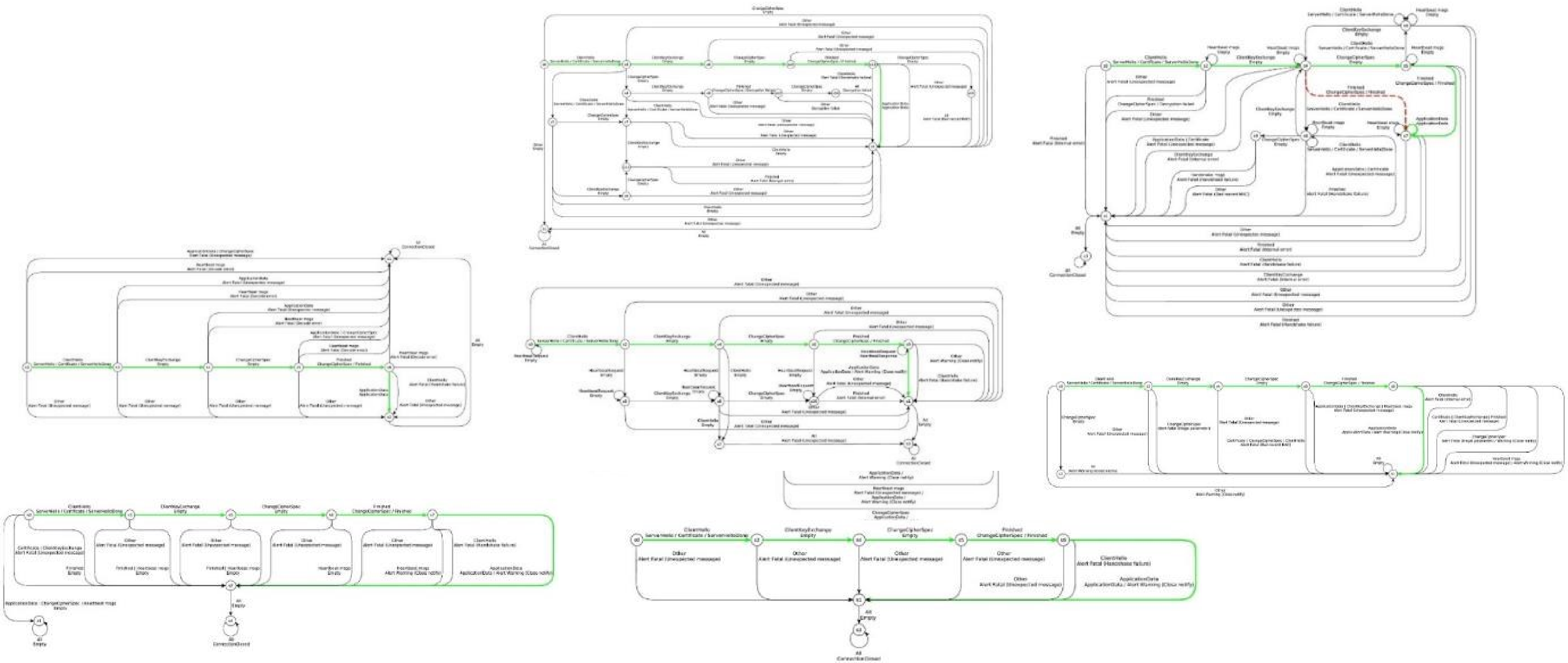
TLS... according to OpenSSL



TLS... according to Java Secure Socket Exension



Which TLS implementations are correct? or secure?



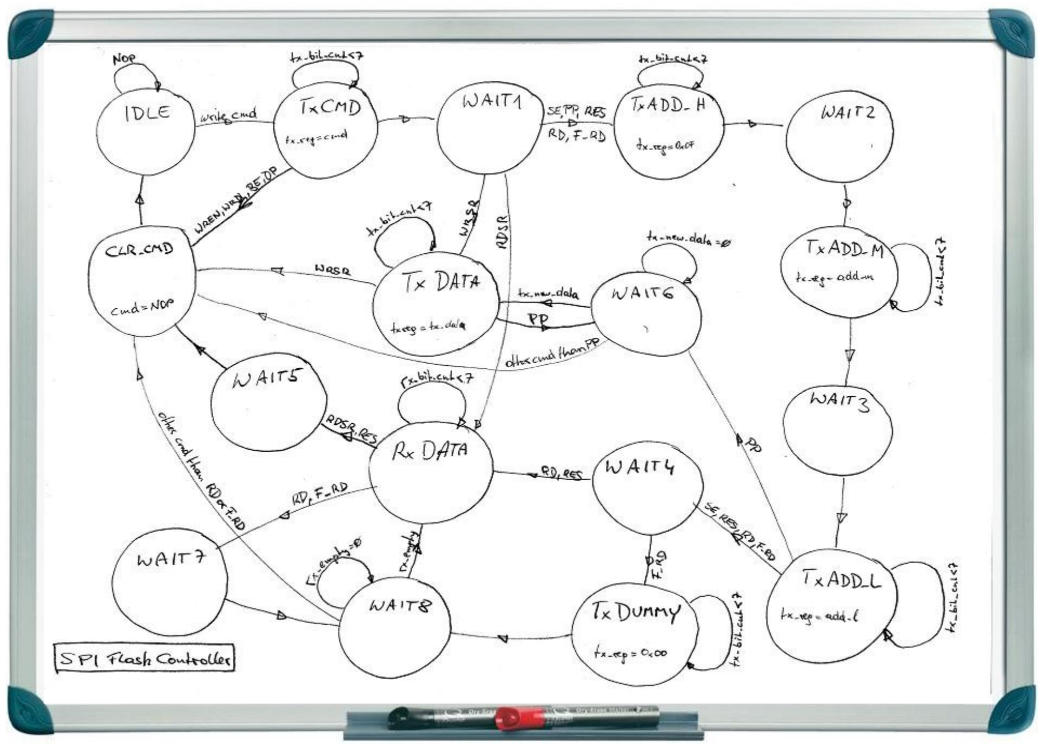
[Joeri de Ruiter et al., Protocol state fuzzing of TLS implementations, Usenix Security 2015]

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

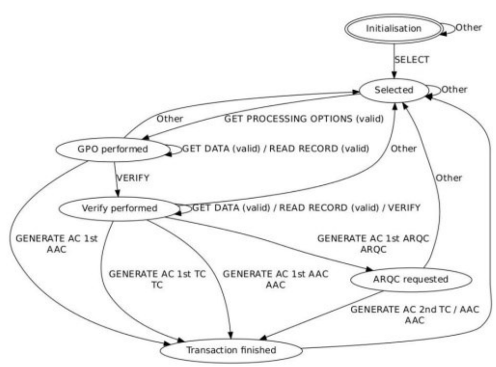
The road we followed

specs

implementing

code

state machine learning



model

Ideally specs would include a state machine!

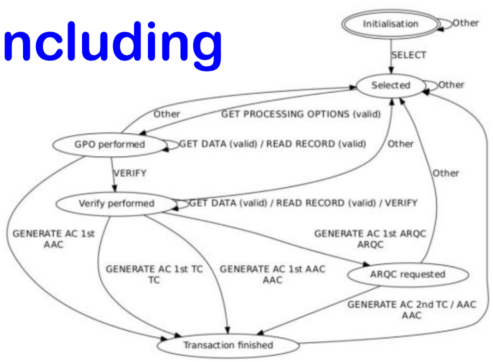
specs

implementing

code

including

model-based testing



model

Or maybe we could generate code?


NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES

Formal methods

What are formal methods?

A formal method consists of

- mathematical formalism to describe (aspects of) systems
- associated mathematical or logical techniques to reason about these models

Example formalisms:

regular expressions, finite state machines (FSMs),

context-free grammars, possibly in ABNF or EBNF notation,

proposition/predicate/temporal logic,

term rewriting systems,

process algebra,

....

Different ways to use formal methods

1. **Make a formal model of (some aspect of) a system, and then prove security/correctness properties of that model**
   - eg. model system components as FSMs and using a model checker to prove absence of deadlock
   - eg. model a security protocol in applied pi-calculus and prove that the protocol is secure
2. **Make a formal model of the programming language, so that you can reason about arbitrary programs**

In principle, formal methods can **prove** correctness and/or security of programs, but… in practice two questions remain:

A. Does the model accurately describe reality?

B. What does it mean for the system to be correct or secure?

Security protocol verification

Security protocols, such as SSH, TLS, Signal/WhatsApp, EMV, can be modelled & verified in tools such as Tamarind and ProVerif

- **Crypto primitives for hashing and decryption are assumed to be secure**
- **Tools can prove that private keys do not leak and the impossibility of replay attacks, Man-in-the-Middle attacks, ..**

The 'Verification of Security Protocols' course at TUE treats this in detail.

Formalisation of EMV in F#

750 pages of prose specification in 370 lines of F#

(Known) security flaws can be found automatically by FS2PV & Proverif

Part of EMV model

```
// Perform DDA Authentication if requested, otherwise do nothing
let card_dda (channel, atc, (sIC,pIC), nonceC) dda_enabled =
  let data = Net.recv channel in
  if APDU.get_command data = INTERNAL_AUTHENTICATE then
    if dda_enabled then
      begin   let nonceT = APDU.parse_internal_authenticate data in
              let signature = rsa_sign sIC (nonceC, nonceT) in
             Net.send channel (APDU.internal_authenticate_response nonceC signature);
            Net.recv channel
      end
    else  failwith "DDA not supported by card"
  else  data
```

Properties checked with ProVerif

1. Sanity checks to ensure absence of deadlock
2. Secrecy of private keys
3. Highest supported card authentication method is used
   - eg no fallback to weaker method can be forced
4. 'transaction security': if a transaction is completed, then everyone agrees on the parameters (eg with/without pin, off/online, amount,…)

```
query   evinj:TerminalTransactionFinish(sda,dda,cda,pan,amount,…)
            ==>   evinj:CardTransactionInit(sda,dda,cda,pan,amount,…)
```

No new attacks found, but most existing attacks inevitably (re)discovered

Security protocol verification

Remaining worries?

- **Are implementations of the protocol (in C, C++, Java, hardware, ...) identical to this model?**
  - **Doing state machine learning of these implementations can provide some confidence!**
- **Are the cryptographic primitives indeed secure?**
- **Are the *implementations* of these crypto primitives (in hardware and/or software secure, esp. against physical side channel attacks?**

Topic of 'Physical attacks on secure systems' by Lejla Batina & Ileana Buhan next semester

Verifying properties of SSH implementations

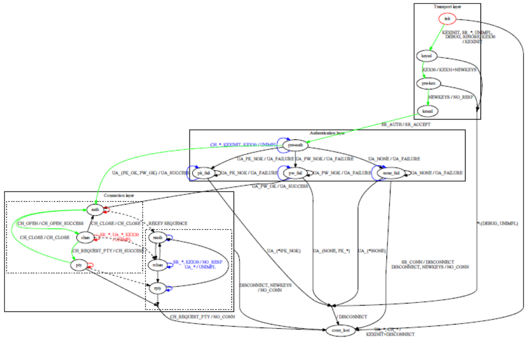
Do SSH implementations conform to requirements stated in the RFC?

To verify this, we

1) **learned state machine models for SSH implementations using LearnLib**

2) **expressed the requirements in temporal logic (LTL)**

3) **used model-checker NuSMV to verify these LTL properties for these state machines**



[Paul Fiterau-Brostean et al., Model learning and Model Cheking of SSH Implementations, SPIN 2017]

Verifying properties of SSH implementations

Example LTL (Linear Temporal Logic) property of SSH

- **RFC 4254 states that "after sending a KEXINIT message, a party MUST not send another KEXINIT or SR_ACCEPT message, until it has sent a ReceivedNewKeys message.**
- **In LTL:** G (out=KEXINIT —> X ((out ! = SR_ACCEPT & out ! = KEXINIT) W receivedNewKeys))

LTL notation: G = Globally, X = neXt W = Weak until

Model checking results of all inferred models

	Property	Key word	OpenSSH	Bitvise	DropBear
Security	Trans.		✓	✓	✓
	Auth.		✓	✓	✓
Rekey	Pre-auth.		X	✓	✓
	Auth.		✓	X	✓
Funct.	Prop. 6	MUST	✓	✓	✓
	Prop. 7	MUST	✓	✓	✓
	Prop. 8	MUST	X*	X	✓
	Prop. 9	MUST	✓	✓	✓
	Prop. 10	MUST	✓	✓	✓
	Prop. 11	SHOULD	X*	X*	✓
	Prop. 12	MUST	✓	✓	X

Program verification

- Formally proving (in the mathematical/logical sense) that a program satisfies some property
  - eg that it does not crash, always terminates, never terminates, meets some functional specification, meets some security requirement, etc
  - *for all possible executions*: ie all possible inputs and all possible scheduling of parallel threads.
- NB in industry, the term verification is used for testing
  but testing provides only *weaker* guarantees
  - because testing will only try some executions
  - except in rare case where you can do exhaustive testing
- Formal verification provides the highest level of assurance that code is correct & secure
  - provided you can specify what it means for the code to be correct & secure

What do we need for program verification?

1. a formal semantics of the programming language
2. a specification language to express properties
3. a logic to reason about programs and specifications
   - aka a program logic
4. a verification tool to support all this

Verification of Java programs using JML

JML is a formal specification language for Java

- to specify behaviour of Java classes
- to record detailed design decisions

- Used by adding annotations to Java source code for pre/postconditions and invariants
  - aka Design-By-Contract style

- Design goal: meant to be usable by any Java programmer

JML formal specification example

```
public class ePurse{
    private int balance;
    //@ invariant 0 < balance && balance < 500;


    //@ requires amount >= 0;
    //@ ensures balance <= \old(balance);
    //@ signals (BankException) balance == \old(balance);
    public debit(int amount) {
    if (amount > balance) {
          throw (new BankException("No way"));}
     balance = balance - amount;
}
```

JML formal specification example


```
public class ePurse{

   private int balance;

   //@ invariant 0 <= balance && balance < 500;
```

This is the kind of flaw program verification will spot!

```
   //@ requires amount >= 0;

   //@ ensures balance <= \old(balance);

   //@ signals (BankException) balance == \old(balance);

   public debit(int amount) {

   if (amount > balance) {

         throw (new BankException("No way"));}

    balance = balance - amount;

}
```

Program verification: the good & the bad

- Program verification doable for 1000s of lines of code ☺, too labour-intensive for millions of lines of code ☹
- Writing specifications of the APIs used can be the bottleneck ☹
- Formally specifying what it means for a system to be correct or correct can be hard ☹
  - But: simply annotating code with the object/class invariants needed to prevent runtime exceptions is a great start

- Annotations also useful for testing: runtime assertion checking.

  Eg JML compiler can translate JML annotations to runtime checks

  Advantages: this provides

  - a very thorough test oracle for 'free'
  - very precise feedback in case of test failures

Program Verification of Hyper-V [2008]

- A hypervisor is a minimal software layer below the OS that turns a physical processor into multiple, isolated virtual processors
- Microsoft's Hyper-V hypervisor is 100 Kloc of C and 5 Kloc of assembly
- Hyper-V was verified using VCC tool for concurrent C, that turns code & specifications into verification conditions for theorem prover Z3

seL4 microkernel [2009]

- *micro*kernel – OS kernel that is *kept to minimum code size, in effort to reduce TCB*
- seL4 is 8,700 lines of C code and 600 lines of assembly
- Verified using interactive theorem prover Isabelle/HOL in L4.verified project at NICTA
- Steps in the verification process
  - Developing abstract, executable specification in Haskell
  - Proving that C & machine code implementation behaves identical to (technically – *simulates*) this Haskell prototype
- Proof size 200,000 lines of proof scripts
  Verification effort 11 person-year

https://trustworthy.systems/projects/seL4-verification/

CompCert [2016]

Dilemma: ***is it better to verify the source code of some program***

or the compiled binary?

- **Advantage of verifying source code: easier to verify ☺**
- **Disadvantage: we have to trust the compiler ☹**

CompCert is C compiler that has been formally verified

- **Using the Coq theorem prover**

miTLS & HACL* [2016 & 2017]

- miTLS is fully verified TLS 1.3 implementation
  - Implementations in
    - functional language F#
    - ML-like functional language F*

https://mitls.org

- HACL* is a formally verified cryptographic library in F*
  - can be compiled down to C

https://github.com/project-everest/hacl-star

Want to know more about formal methods?

- Robbert Krebbers teaches new course in spring semester:

  Program verification with types and logic

  https://www.sws.cs.ru.nl/Teaching/ProgramVerification

  https://robbertkrebbers.nl

- Freek Verbeek works on verification of binaries

  at Open University & VirginiaTech

  and has thesis projects in this area

  https://www.cs.ru.nl/~freekver