Automated reverse engineering of security protocols

Learning to fuzz, fuzzing to learn

Fides Aarts Erik Poll

Joeri de Ruiter Sicco Verwer

Radboud University Nijmegen

Fuzzing

1. Plain fuzzing, with long inputs to trigger seg-faults and find hence find buffer overflows

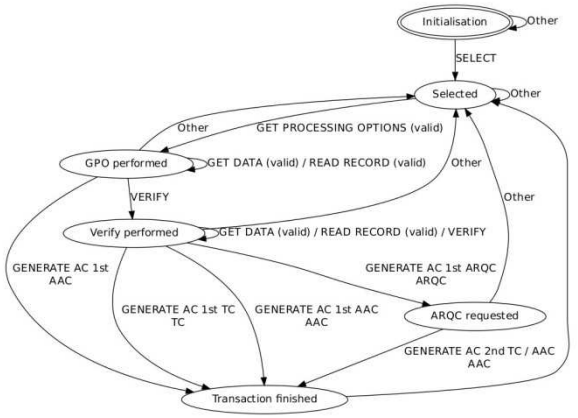
2. Protocol fuzzing based on known *protocol format*, fuzzing interesting fields [eg SNOOZE]

Command APDU						
Header (required)				Body (optional)		
CLA	INS	P1	P2	Lc	Data Field	Le

3. State-based fuzzing to reach interesting states in the *protocol state-machine*, and then fuzz there

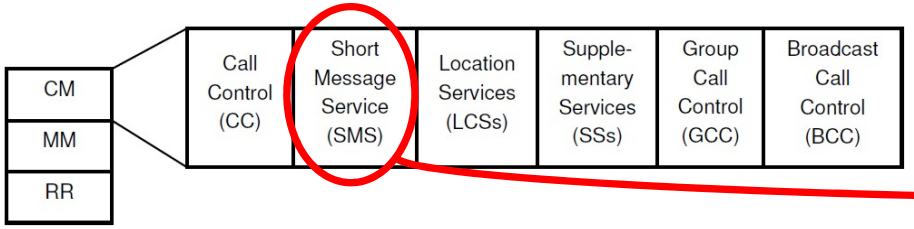
   essentially model-based testing

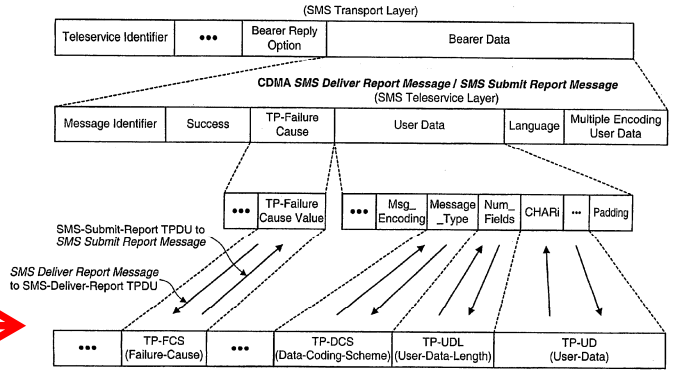
   [eg Peach, jTor]



Example: protocol fuzzing to crash things

- GSM is a very "rich" protocol





- Fuzzing protocol fields quickly reveals weird behaviour
  - using USRP as GSM cell tower

  - no SMS-of-death found,
    but lots of phones crashing in weird ways

Example: protocol fuzzing for information leakage

- e-passport implements protocol to prevent "skimming"
  - correct protocols runs don't leak info to an eavesdropper

Command APDU						
Header (required)				Body (optional)		
CLA	INS	P1	P2	Lc	Data Field	Le

- But fuzzing "incorrect" instructions leaks a fingerprint, unique per implementation and hence (almost) unique per country
  - for Australian, Belgian, Dutch, French, German, Greek, Italian, Polish, Spanish, Swedish passports

In the other direction:

Instead of using protocol knowledge when testing,
we can also use testing to gain protocol knowledge

or to gain knowledge about protocol *implementation*

In order

- to analyse your own code and hunt for bugs, or
- to reverse-engineer someone else's unknown protocol,
  eg a botnet, to fingerprint or analyse (and attack) it

What to reverse engineer?

Different aspects that can be learned:

- timing/traffic analyis
- protocol formats [eg Discoverer, Dispatcher, Tupni,....]
- protocol state-machine [eg LearnLib]

or both protocol format & state-machine [eg Prospex]

How to reverse engineer?

- ***passive*** vs ***active*** learning

  ie passive observing or active testing

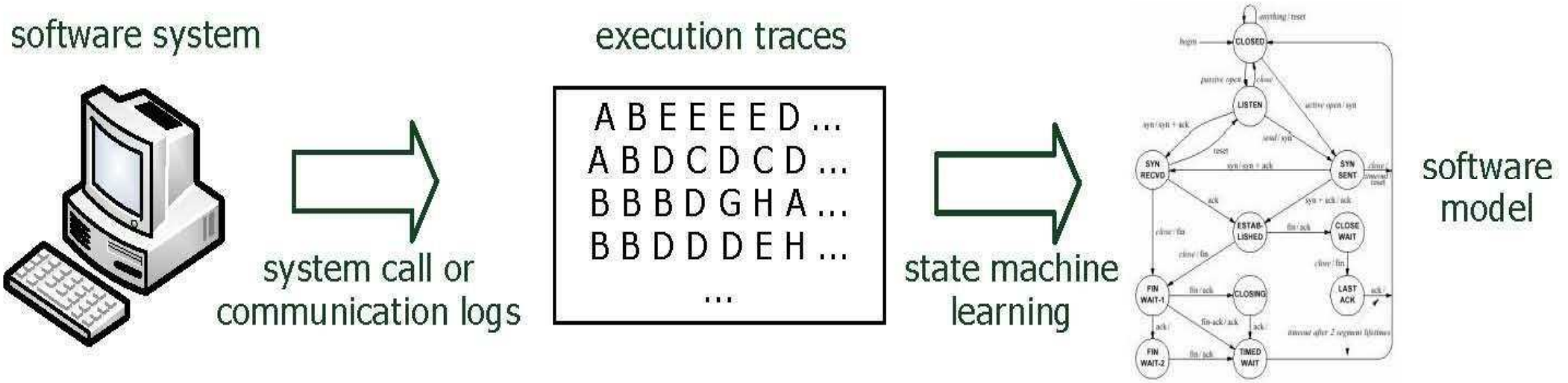
  - active learning involves a form of fuzzing
  - active learning is harder, as it requires more software in test harness that produces meaningful data
  - these approach learns different things;
    passive learning uses & produces statistics

- black box vs white box

  ie only observing in/output or also looking inside running code

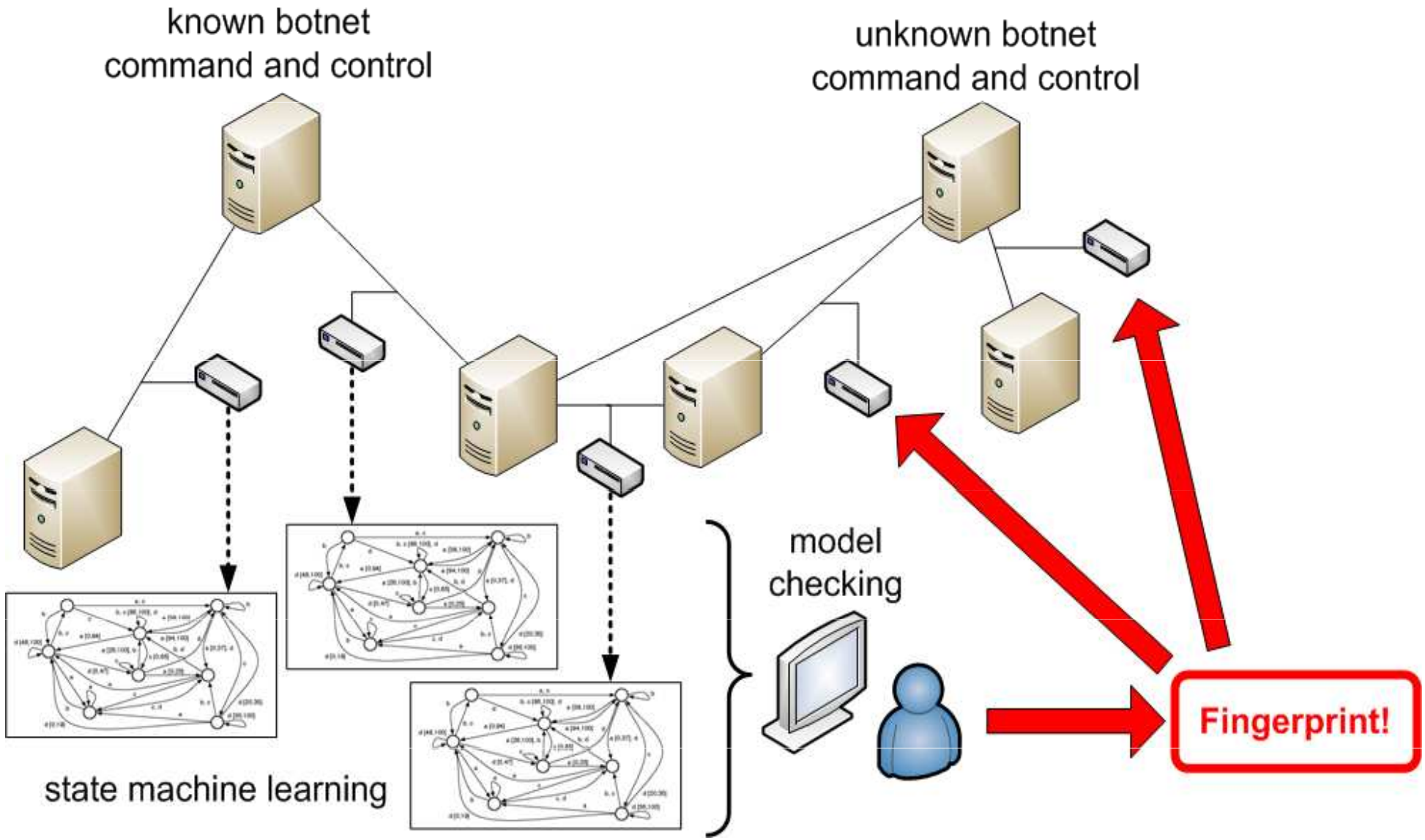
  - white-box analysis for eg encypted traffic, by looking at handling of data after decryption [eg ReFormat, TaintScope]

Passive learning

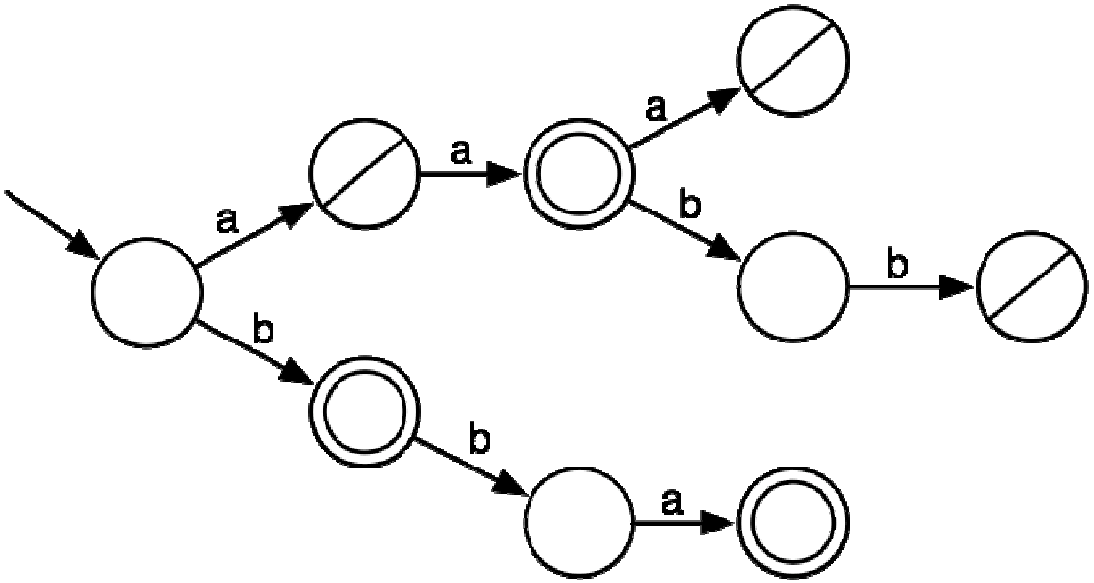


eg (timed) state machine

Learning malware models



Passive learning

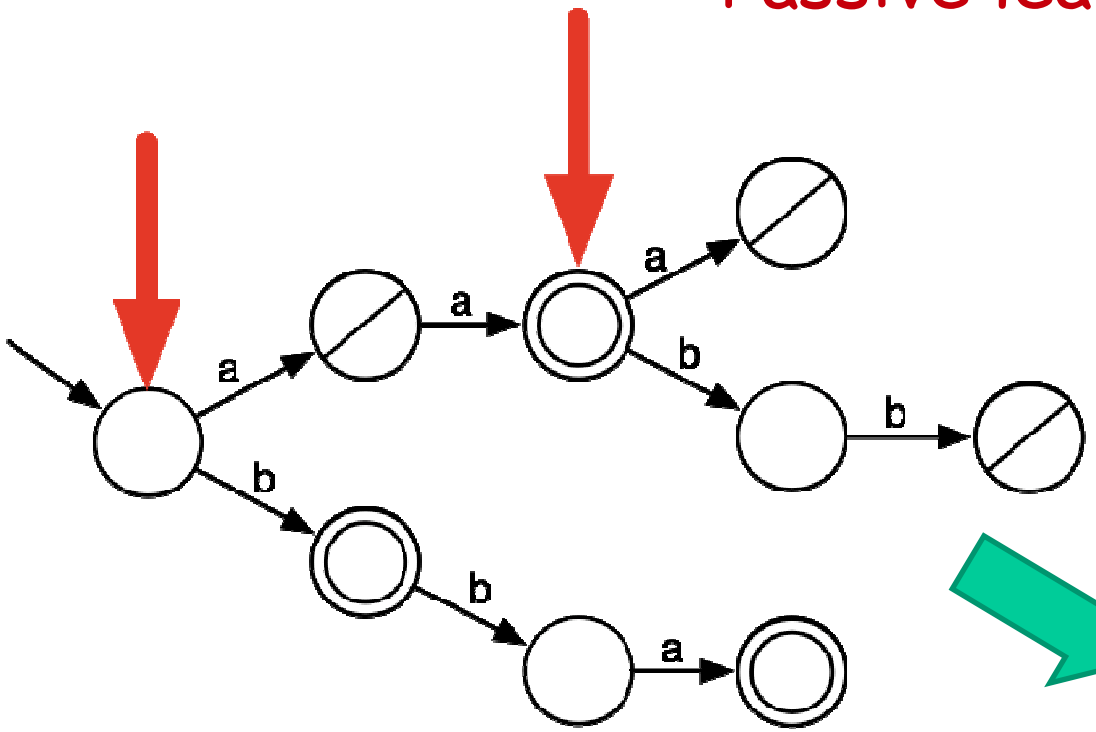


positive data: aa, b, bba

negative data: a, aaa, aabb

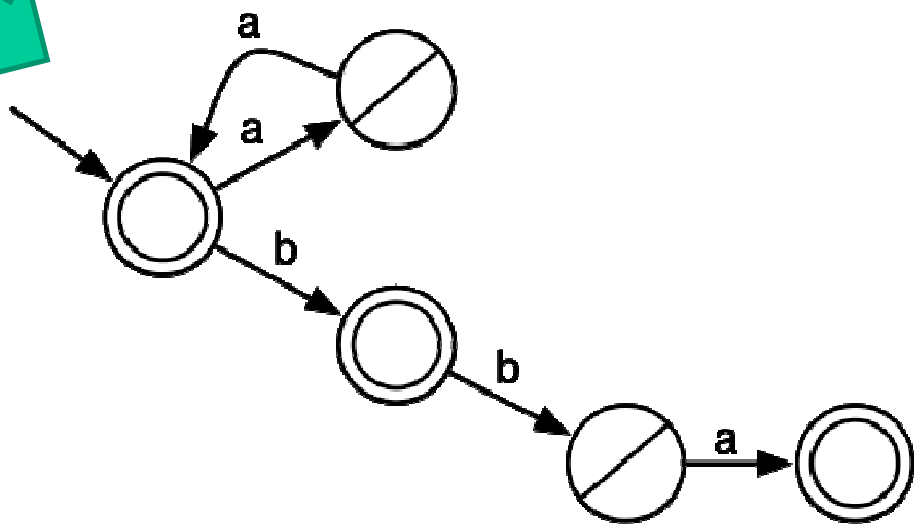
in a prefix tree

Passive learning



select two states
combine and iterate

positive data: aa, b, bba
negative data: a, aaa, aabb
in a prefix tree

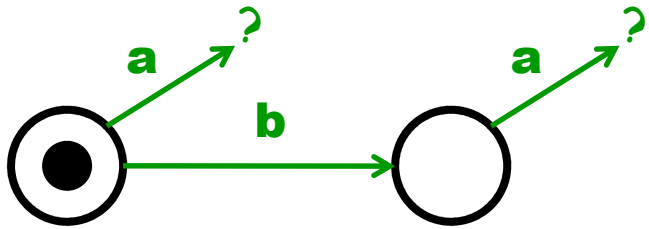


Active learning with Angluin's L* algorithm

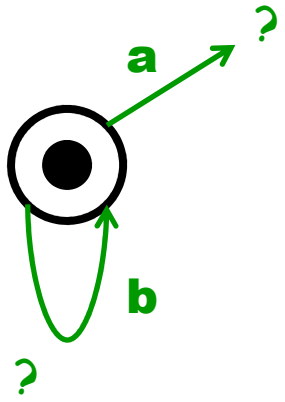
Basic idea: compare a deterministic system's response to

- **a**
- **b ; a**

If response is different, then



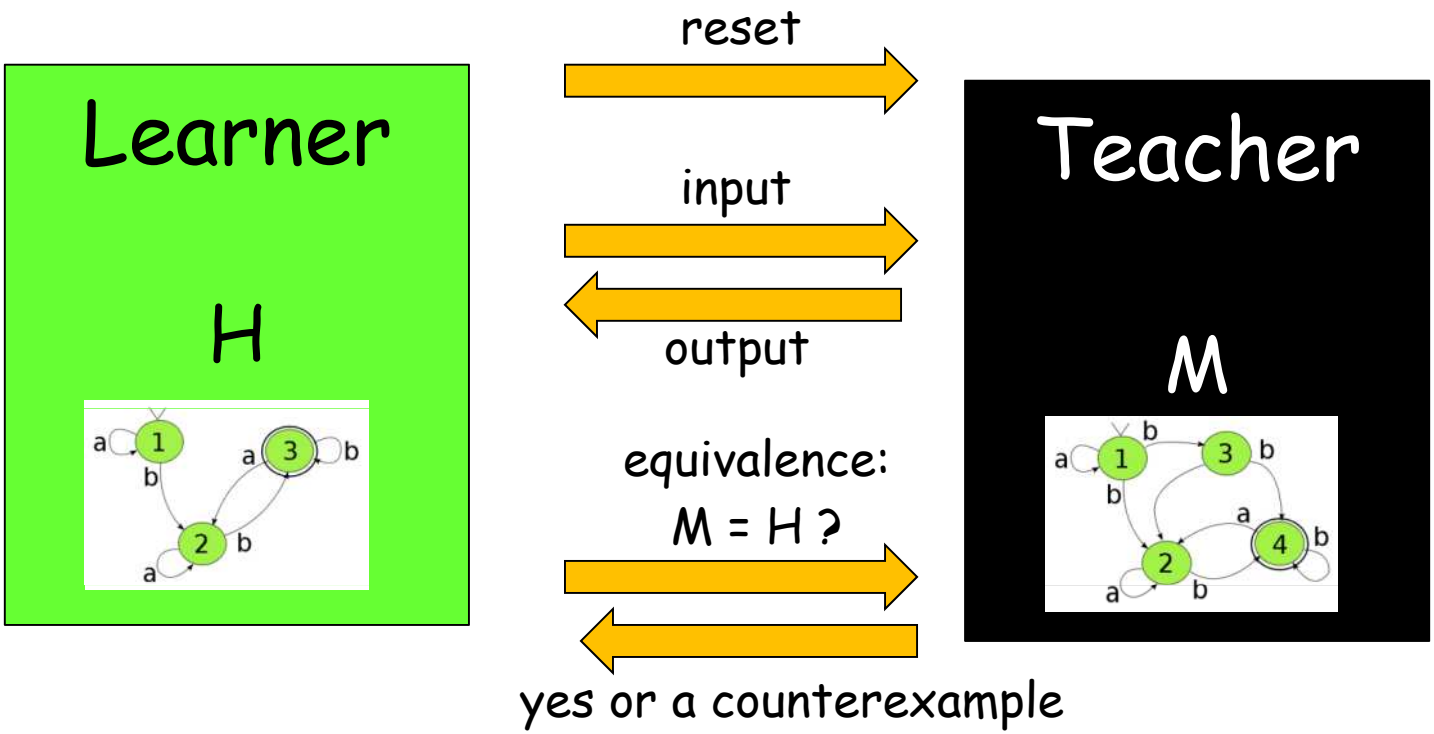
otherwise



?

Active learning with L*

Implemented in LearnLib library



Equivalence can only be approximated in a black box setting

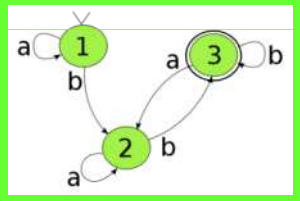
Learning set-up for banking cards

abstract instructions
and response

concrete instructions
and response

Learner

H



instruction
INS

2 byte
status word SW

Teacher

test
harness

M

INS + args

data + SW

Test harness for EMV

Our test harness implements standard EMV instructions

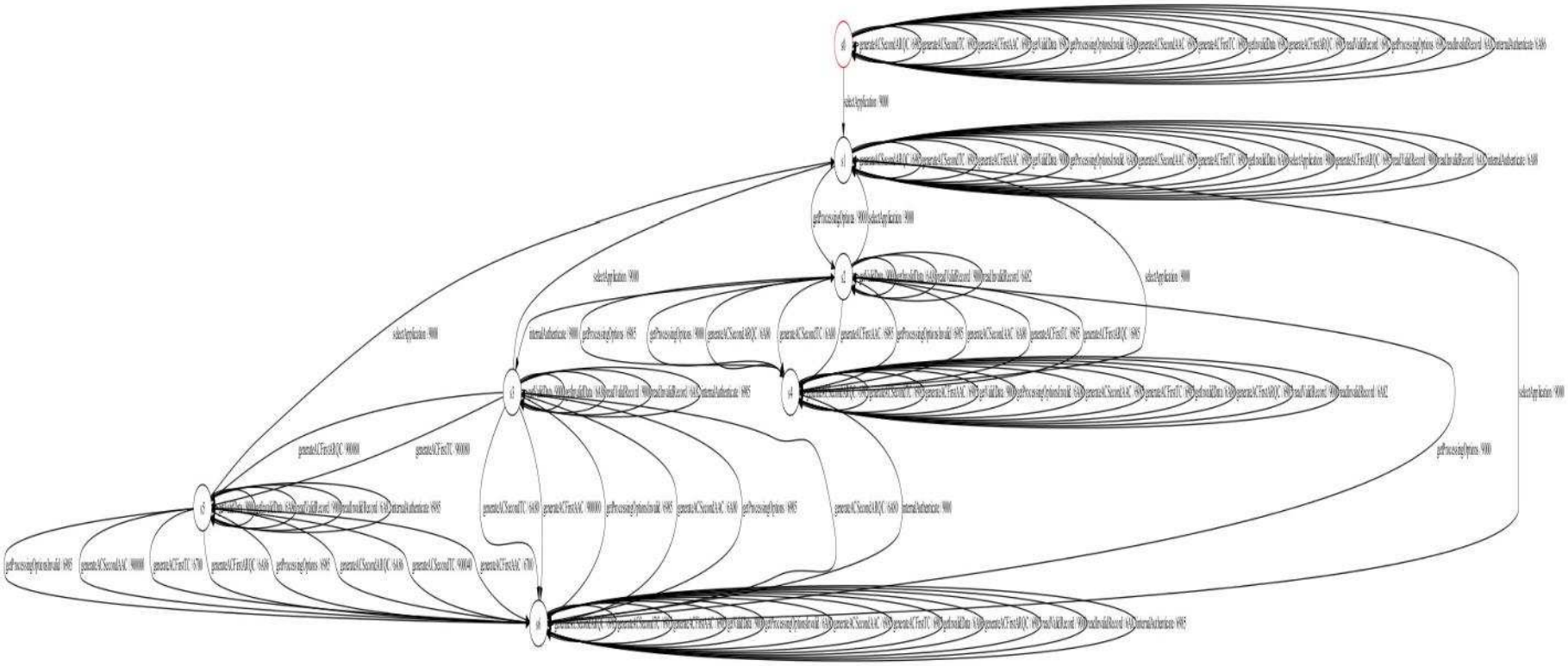
- SELECT (to select application)
- INTERNAL AUTHENTICATE (for a challenge-response)
- VERIFY (to check the PIN code)
- READ RECORD
- GENERATE AC (to generate application cryptogram)

LearnLib then tries to learn all possible combinations

- Most commands with fixed parameters, but some with different options

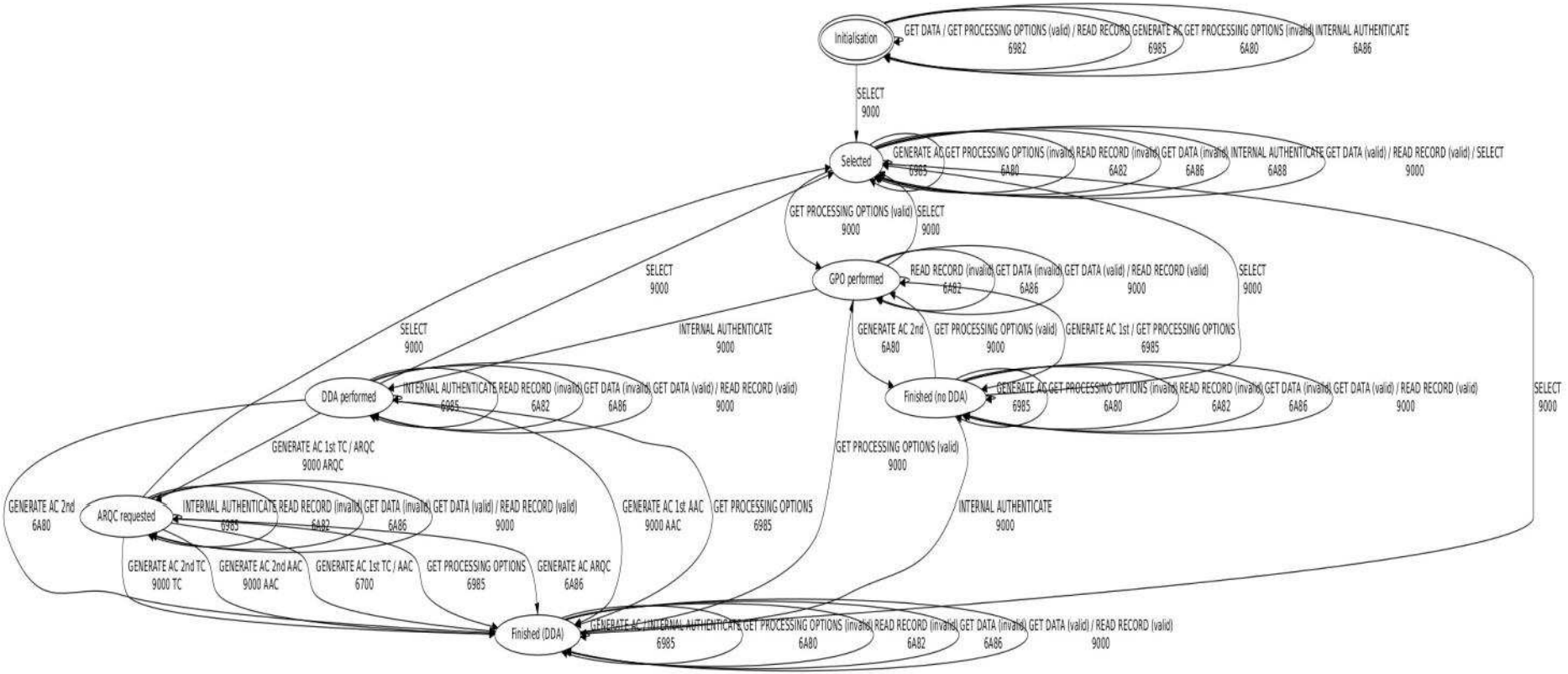
Maestro application on Volksbank bank card

raw result



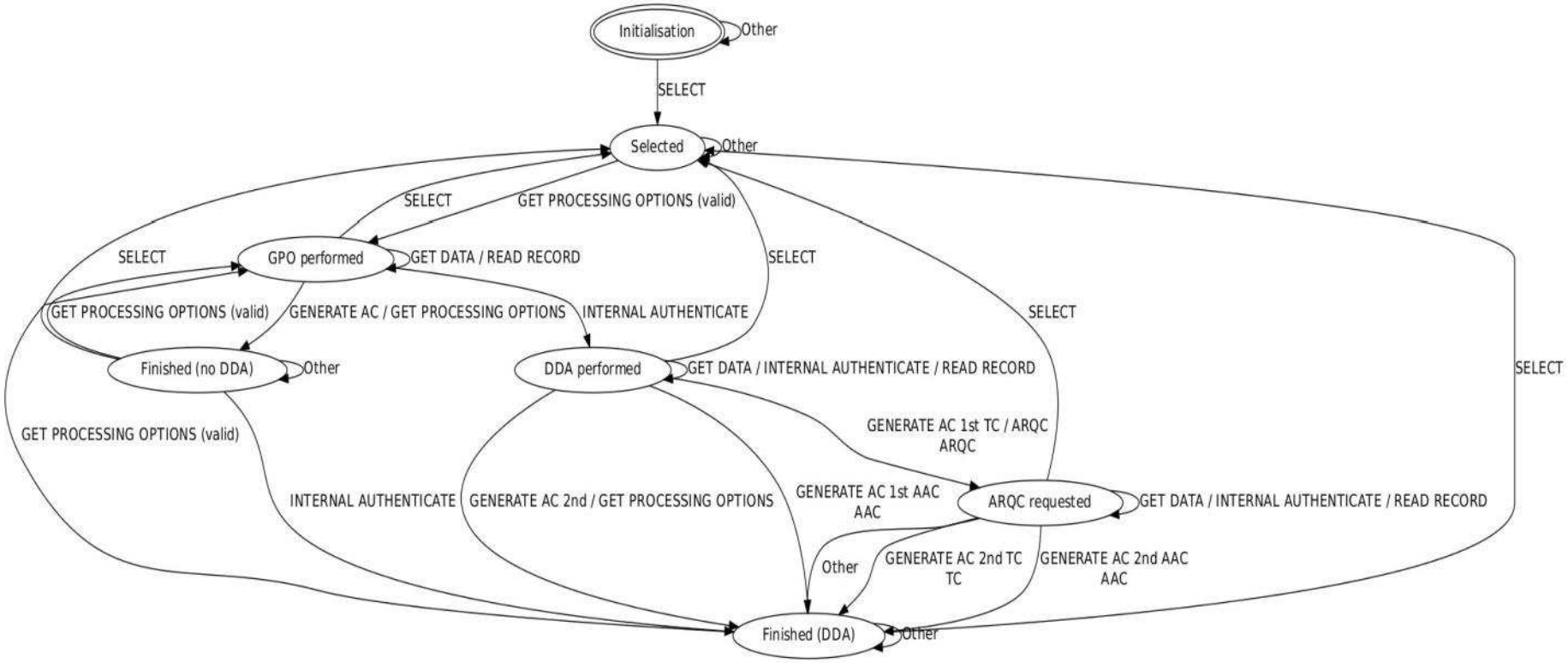
Maestro application on Volksbank bank card

merging arrows with identical outputs



Maestro application on Volksbank card

merging all arrows with same start & end state

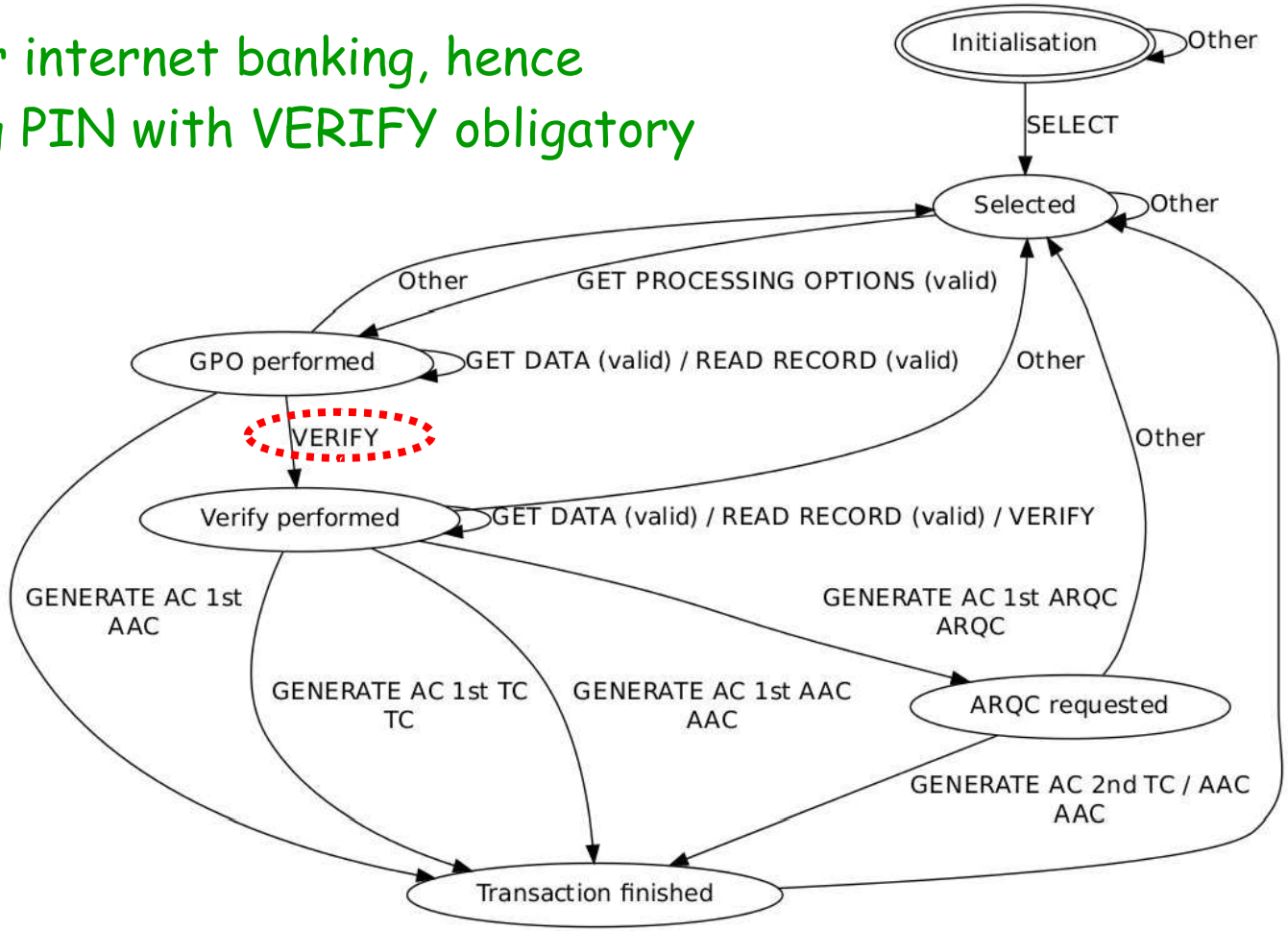


Formal models for free!

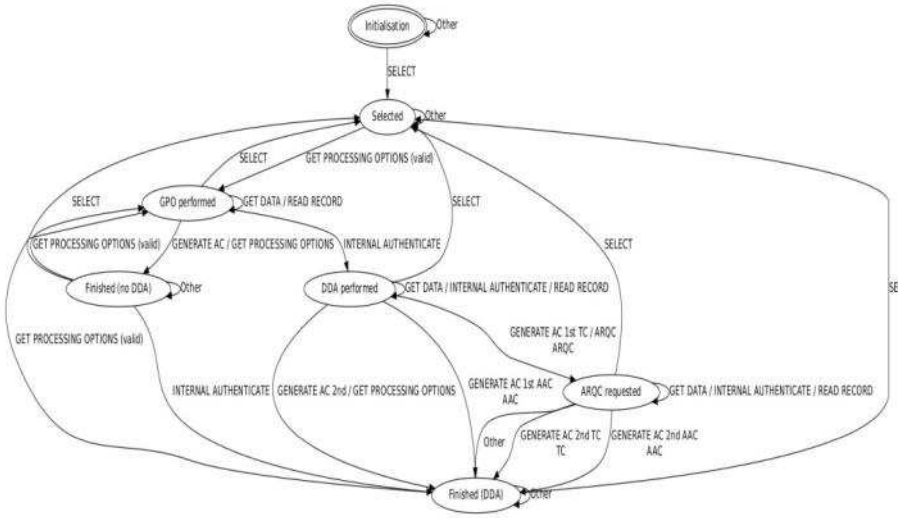
- Experiments with Dutch, German and Swedish banking and credit cards
- Learning takes between 9 and 26 minutes
- Editing by hand to merge arrows and choose sensible names for states
  - could be automated
- Limitations
  - We do not try to learn response to incorrect PIN as cards would quickly block...
  - We cannot learn about one protocol step which requires knowledge of card's secret 3DES key
- No security problems found, but interesting insight in implementations

SecureCode application on Rabobank card

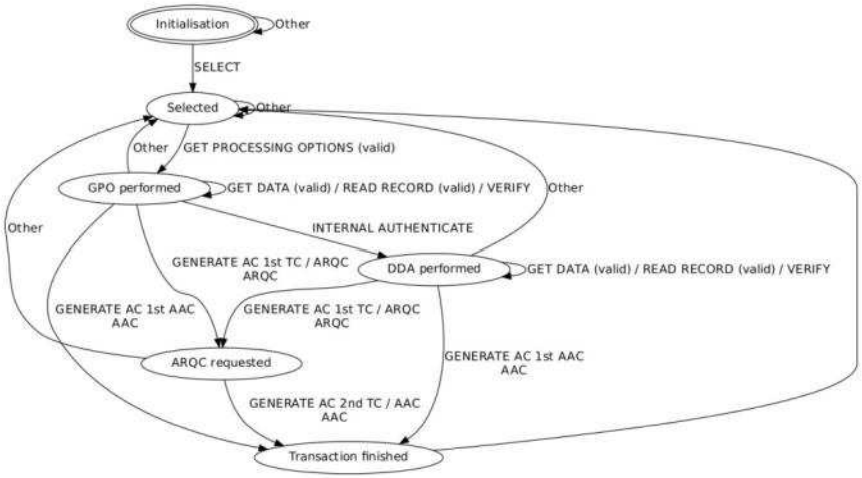
used for internet banking, hence
entering PIN with VERIFY obligatory



understanding & comparing implementations



Volksbank Maestro implementation



Rabobank Maestro implementation

Are both implementations correct & secure? And compatible?

Using these state-diagrams

- Analysing the models by hand, or with model checker, for flaws
  - to see if *all paths* are correct & secure
- Fuzzing or model-based testing
  - using the diagram as basis for automated fuzz testing
  - fuzzing the order and/or the parameters of commands
- Program verification
  - proving that there is no functionality beyond that in the diagram
- Using it when doing a manual code review

Case study: analysis of internet banking devices

- Just in case you think that *of course there won't be security flaws in banking soft/hardware that could be found using these techniques...*
- We analysed a USB-connected smartcard reader for internet banking that provides a trusted display for What-You-Sign-Is-What-You-See

- Reverse engineering reveal a major security protocol flaw
- This was done manually but could have been found by fuzzing USB instructions

[Blom et al., Designed to fail, NORDSEC 2012]

Conclusions

- Fuzzing (model-based testing) and active learning are closely related
- State machines are a great specification formalism
  - easy to draw on white boards, typically omitted in official specs

  and you can extract them for free from implementations
  - using very standard, off-the-shelf, tools like LearnLib
- Useful for security analysis of protocol implementations
  - for reverse engineering, fuzz testing, or formal verification
- Future work: learning *extended* finite state machines with variables (eg the internal transaction counter in EMV cards)

Questions?