Formal models of banking cards for free!

Fides Aarts Erik Poll Joeri de Ruiter

Radboud University Nijmegen

Program Verification

To verify a program you need:

1. a program logic
2. a tool supporting this program logic
3. *something to verify*

What to verify?

Not so obvious for most software. Some possibilities

- generic safety properties eg no NullpointerExceptions

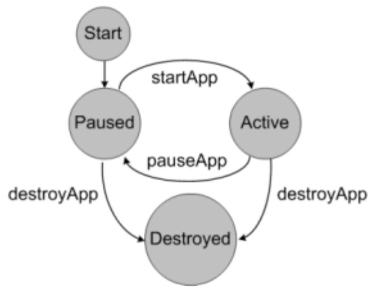
  pros: easy, generic, and obviously correct!

- class invariants

  pros: capture design decisions implicit in & orthogonal to code

- functional specs
  - pre & postconditions
  - state diagrams

but detailed postcondition is often just another (functional) implementation



What to verify?

What to verify?

What to verify? Correctness vs Security

Security is harder to specify (and test) than correctness

- Correctness is about presence of required functionality
- Security is (also?) about absence of unwanted functionality

One can argue about whether correctness implies security or vv.

For finite state machines: it is easier to draw a simple diagram for the normal paths than a complex diagram with also all abnormal paths

Case study: EMV

EMV

The standard for smartcards for banking

- started 1993 by EuroPay, MasterCard, Visa
- Specs controlled by EMVCo which is owned by MasterCard, VISA, American Express, JCB
- Over 1 billion cards in use
- EMV-compliance required for SEPA Single Euro Payments Area

EMV complexity

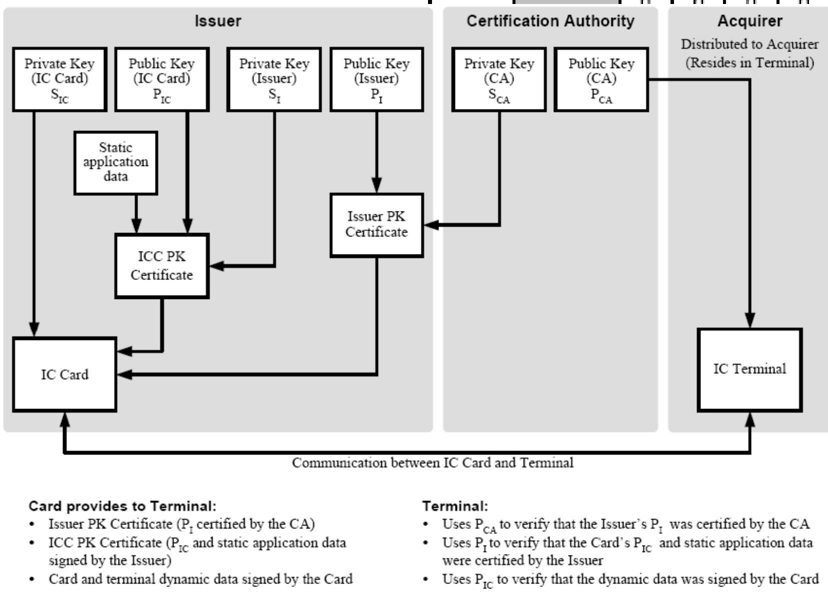
- EMV is not a protocol, but a "protocol toolkit suite": *many* options and parameterisations (incl. proprietary ones)
  - 3 different card authentication mechanisms
    - SDA, DDA, CDA
  - 5 different cardholder verification mechanisms
    - online PIN, offline plaintext PIN, offline encrypted PIN, handwritten signature, no card holder verification
  - 2 types of transactions: offline, online

  All these mechanisms again parameterised by Data Object Lists (DOLs)
- Specification public but very complex (>750 pages)

EMV specs

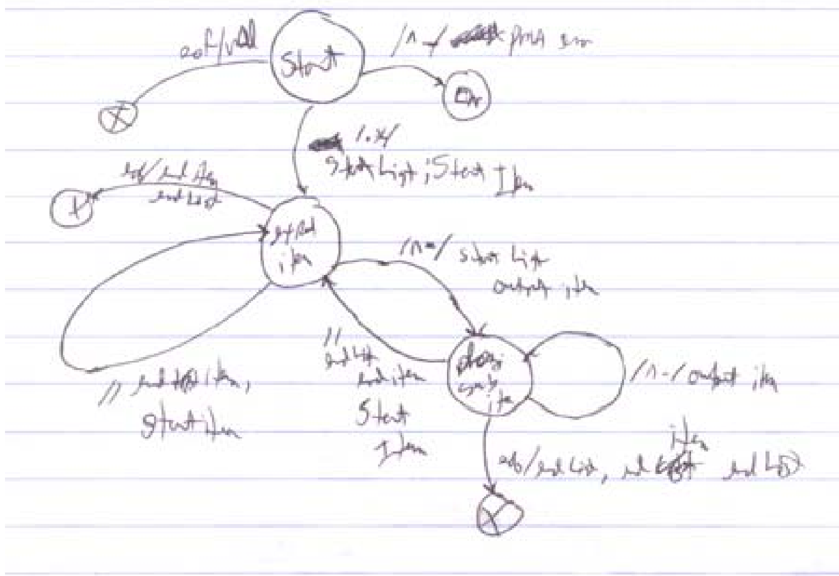
Value	Meaning
'00'	Always
'01'	If unattended cash
'02'	If not unattended cash and not manual cash and not purchase with cashback
'03'	If terminal supports the CVM [19]
'04'	If manual cash
'05'	If purchase with cashback
'06'	If transaction is in the applicati value (see section 10.5 for a disc
'07'	If transaction is in the applicati
'08'	If transaction is in the applicati (see section 10.5 for a discussior
'09'	If transaction is in the applicati
'0A' - '7F'	RFU
'80' - 'FF'	Reserved for use by individual p

b8	b7	b6	b5	b4	b3	b2	b1	Meaning
0								RFU
	0							Fail cardholder verification if this CVM is unsuccessful
	1							Apply succeeding CV Rule if this CVM is unsuccessful
		0	0	0	0	0	0	Fail CVM processing
		0	0	0	0	0	1	Plaintext PIN verification performed by ICC
		0	0	0	0	1	0	Enciphered PIN verified online
						1	1	Plaintext PIN verification performed by ICC and signature (paper)
						0	0	Enciphered PIN verification performed by ICC
						0	1	Enciphered PIN verification performed by ICC and signature (paper)
						x	x	Values in the range 000110-011101 reserved for future use by this specification
						1	0	Signature (paper)
						1	1	No CVM required
						x	x	Values in the range 100000-101111 reserved for use by the individual payment systems
						x	x	Values in the range 110000-111110 reserved for use by the issuer
						1	1	This value is not available for use



- 750 pages of this...
- We made a formal model in F# and verified it with ProVerif [TOSCA 2011], but this is at some level of abstraction...
  Does this model really correctly describe implementations?

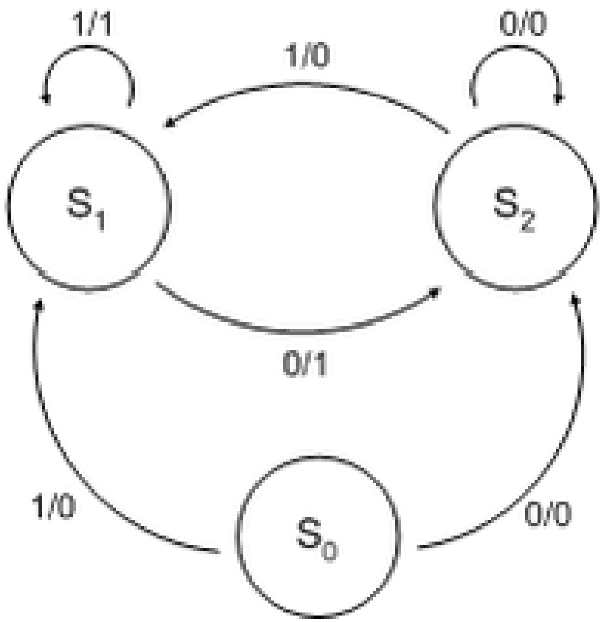
Coming up with formal specs?



Smartcards are Mealy machines

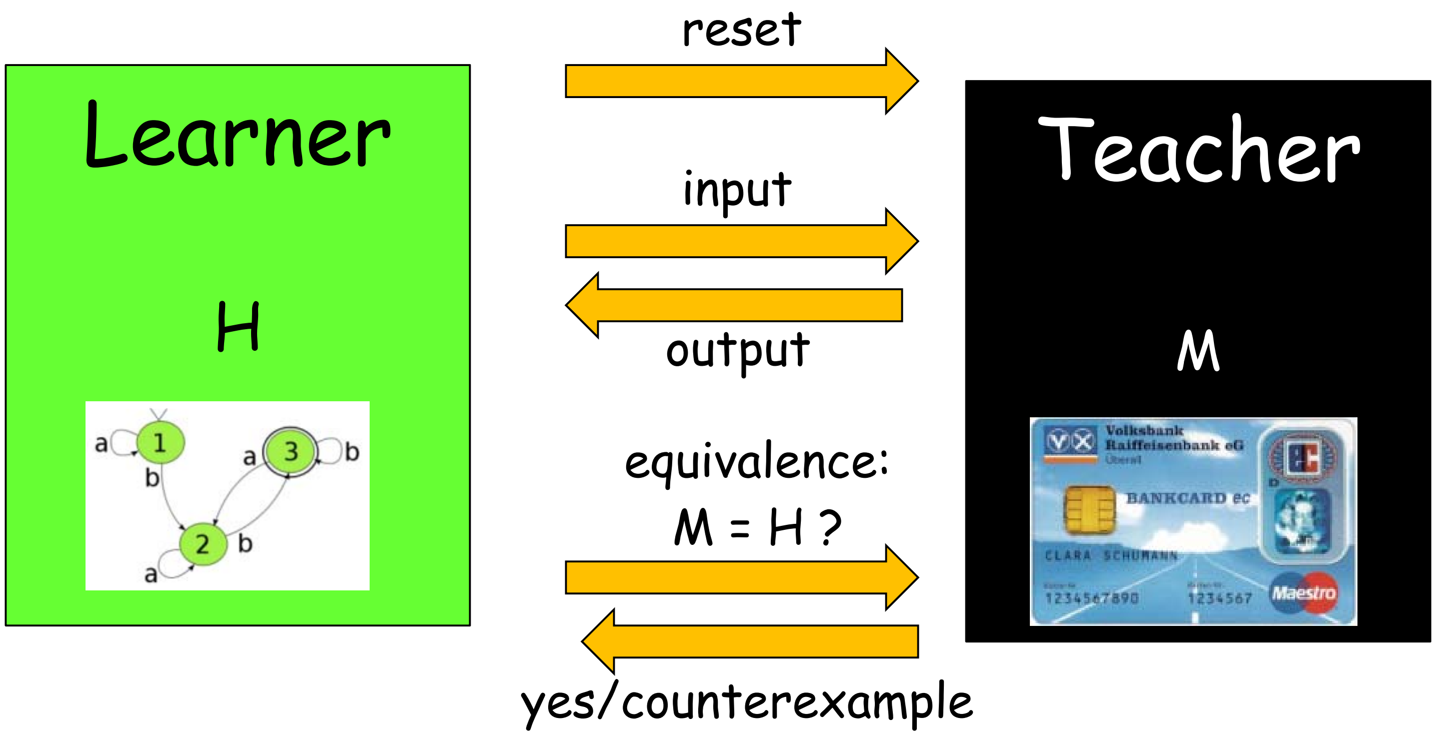
A smartcard is an input-enabled Mealy machine

- Mealy machine: has input and output on every transition
- input-enabled: we can try any input in any state



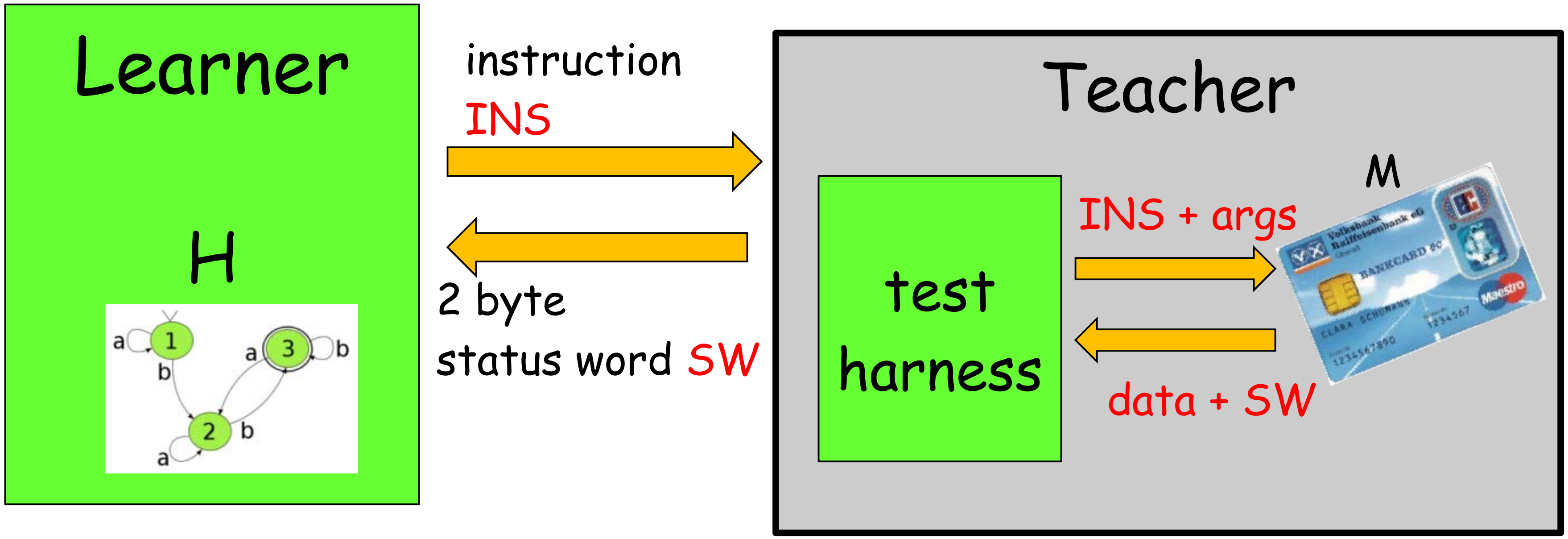
L* learning algorithm for Mealy machines

Implemented in LearnLib library



equivalence can only be approximated in a black box setting

learning set-up for banking cards



Test harness for EMV

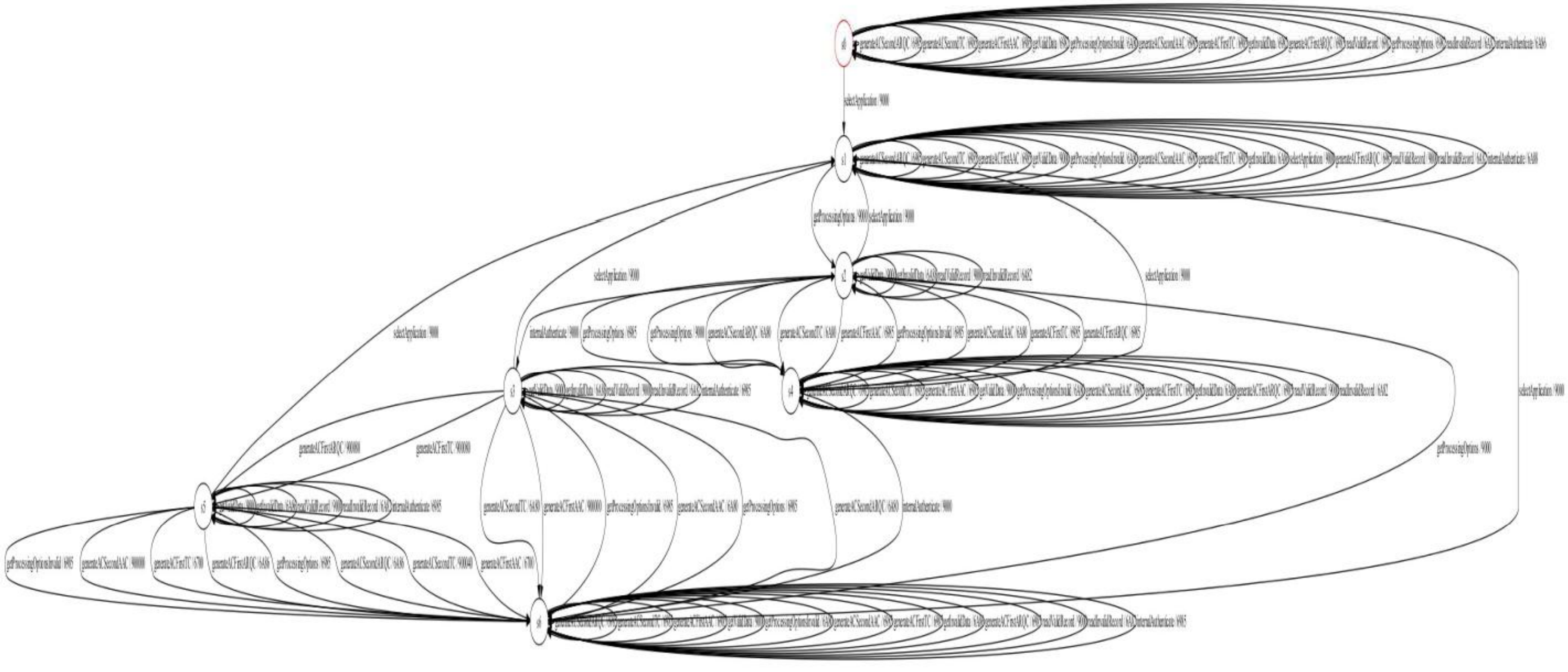
Our test harness implements standard EMV instructions

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

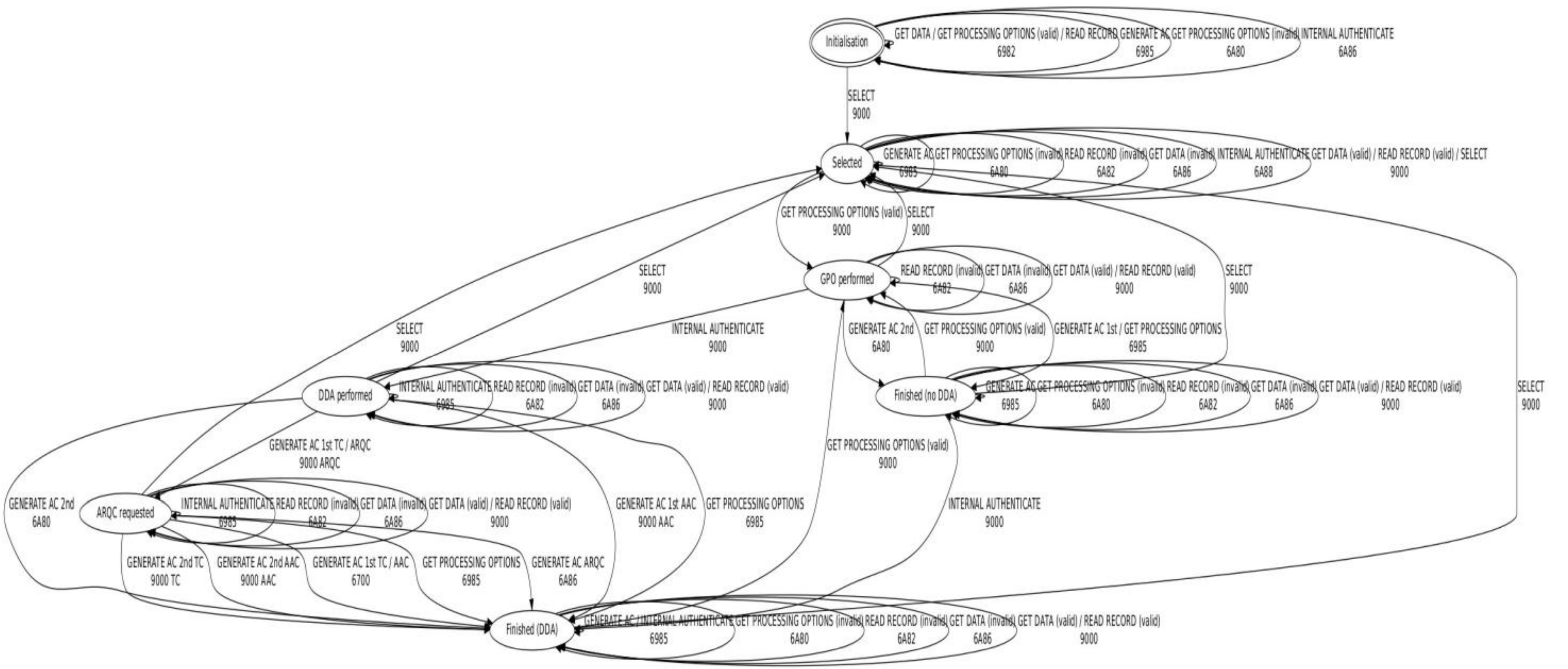
LearnLib then tries to learn all possible combinations

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

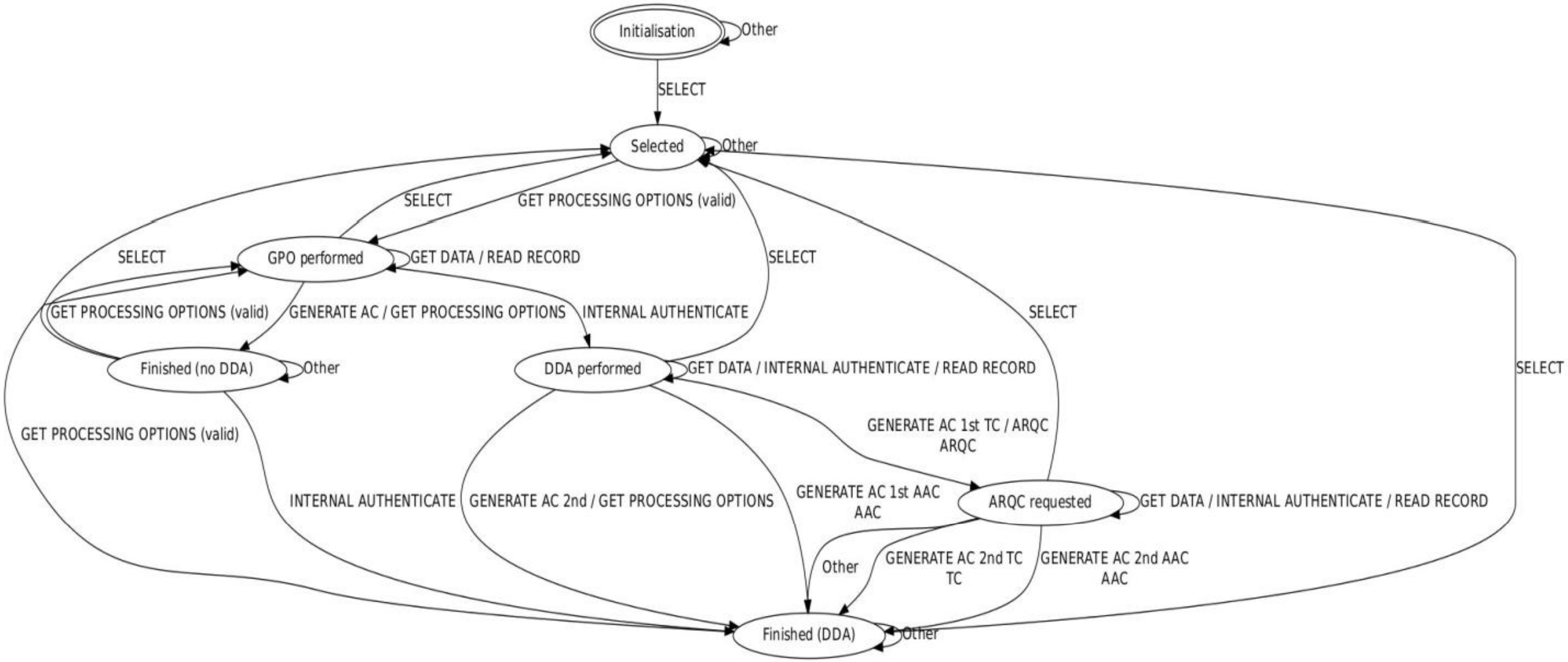
Maestro application on Volksbank bank card
raw result



Maestro application on Volksbank bank card merging arrows with identical outputs



Maestro application on Volksbank card merging all arrows with same start & end state



Learning experiments, efforts, and limitations

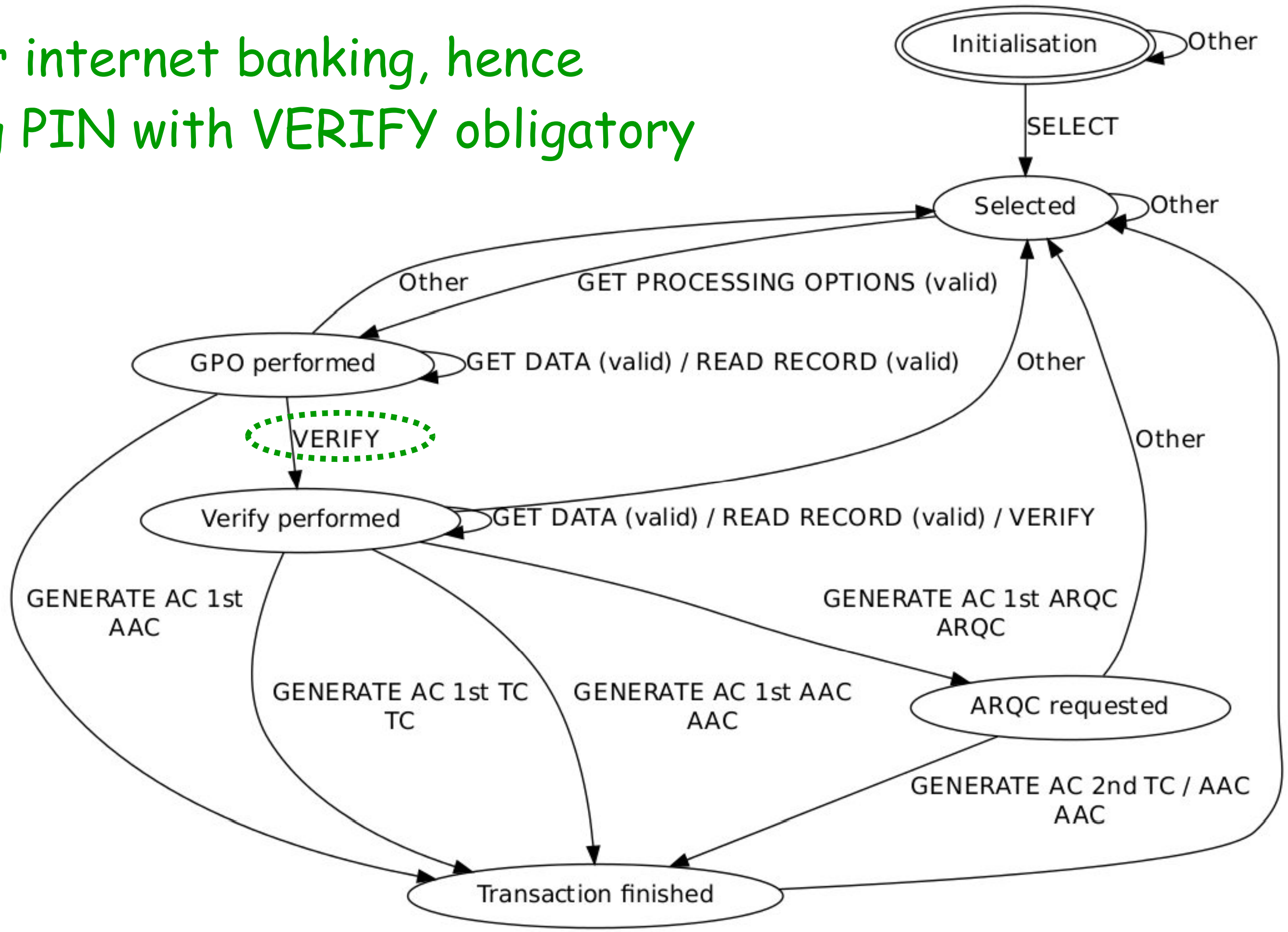
- Experiments with Dutch, German and Swedish banking and credit cards
- No security problems found, but interesting insight in implementations
- Learning takes between 9 and 26 minutes
- Editing by hand to merge arrows and choose sensible names for states
  - could be automated
  - alternative: using (nested) hyperstates
- 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

Using these diagrams

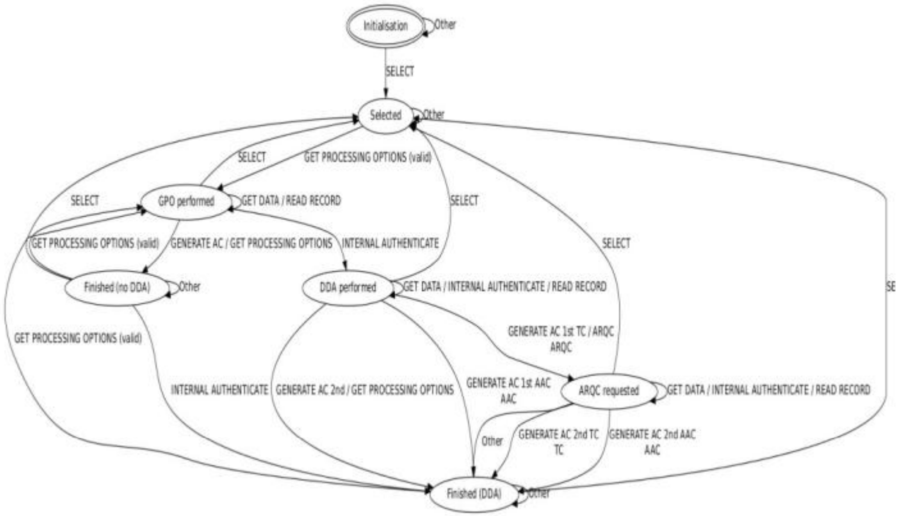
- just reverse engineering
  - looking at the diagrams to see if *all paths* are correct & secure
- fuzzing or model-based testing
  - using the diagram as basis for automated fuzz testing
  - one can fuzz the order and/or the parameters of commands
  - aka protocol fuzzing or model-based testing
- program verification
  - proving that there is no functionality beyond that in the diagram

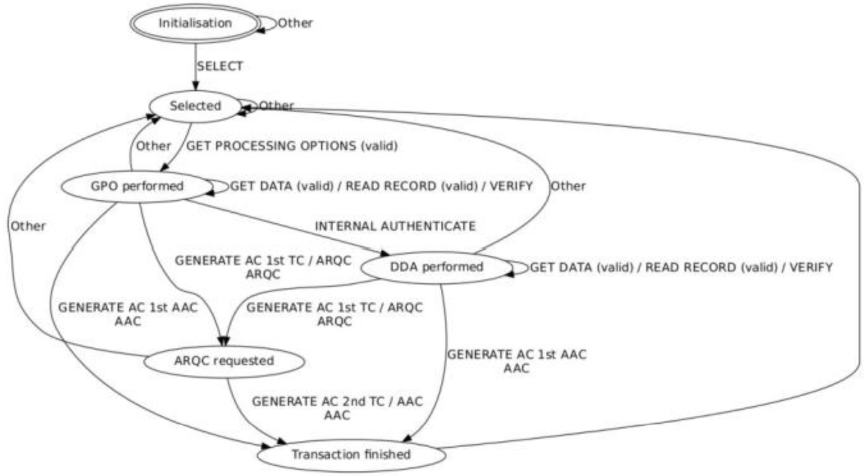
SecureCode application on Rabobank card

used for internet banking, hence
entering PIN with VERIFY obligatory



understanding & comparing implementations





Volksbank Maestro implementation

Rabobank Maestro implementation

Are both implementations correct & secure? Or compatible?

Related work

Learning for automated protocol reverse engineering

- We use active learning, other approaches use passive learning
- Some approaches also try to infer message formats;
  we assume message formats are known (here: given by EMV specs)

Protocol fuzzing

- Our active learning involves state-based protocol fuzzing, which is a form of model-based testing
  - Protocol fuzzing typically only involves fuzzing message contents;
    but state-based fuzzers take the protocol state & message order into account
- Learning automata and state-based protocol fuzzing can be seen as duals

Conclusions

- Finite 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
- 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)