Fuzzing 101

Erik Poll

Radboud Universiteit Nijmegen

The security testing paradox

- Security testing is *harder* than normal, functional testing
  - We have no idea what we are looking for!
    Bizarre inputs may trigger obscure bugs that are exploitable in unexpected ways, and finding these test inputs is hard!
  - Normal users are good testers, as they will complain about functional problems; but they will not complain about security flaws
- Security testing is *easier* than normal, functional testing
  - We can test for some classes of bugs in automatic way using fuzzing
  - Fuzzing is the great success story in software security in the past decade, esp. thanks to afl

The basic idea: how would you test this?

```
Please enter your first name

>
```

1. ridiculously long input, say a few MB

   If there is a buffer overflow, this is likely to trigger a SEG FAULT

2. `%x%x%x%x%x%x%x%x`

   To see if there is a format string vulnerability

3. More generally: include other problematic ingredients

   `\0 -  < | > & ; { } ' " <script> ;DROP TABLE`

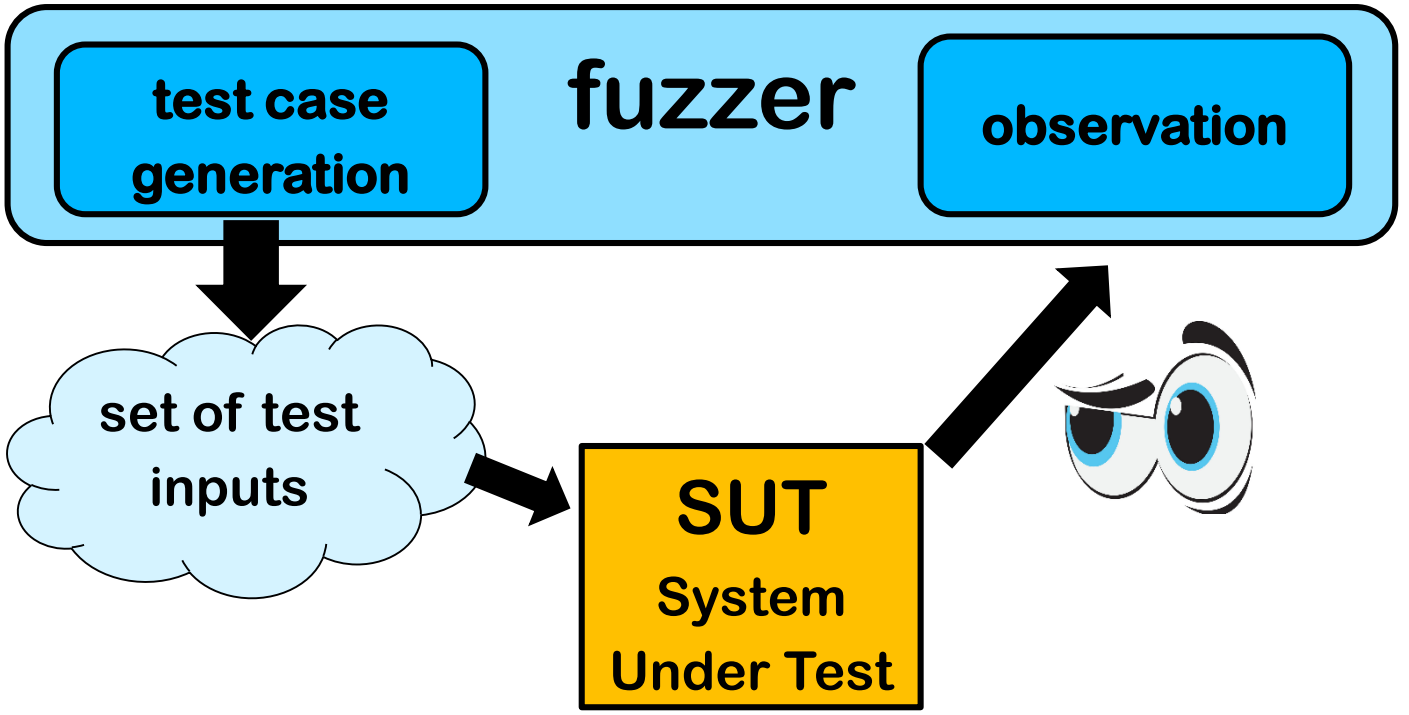
   depending on back-ends/technologies/APIs used.

   But: *if the system does not crash or misbehave in detectable way, we do not know if our input reveals a bug.*

Fuzzing aka fuzz testing

(semi) *automatically* generate 'random' inputs and check if an application *crashes* or *misbehaves in observable way*

Great for certain classes of bugs, esp. memory corruption bugs



[Miller et al., An empirical study of the reliability of UNIX utilities, CACM 1990]

Basic strategies for testcase generation

1. **Totally dumb fuzzing**

   generate random (long) inputs

2. **Mutation-based**

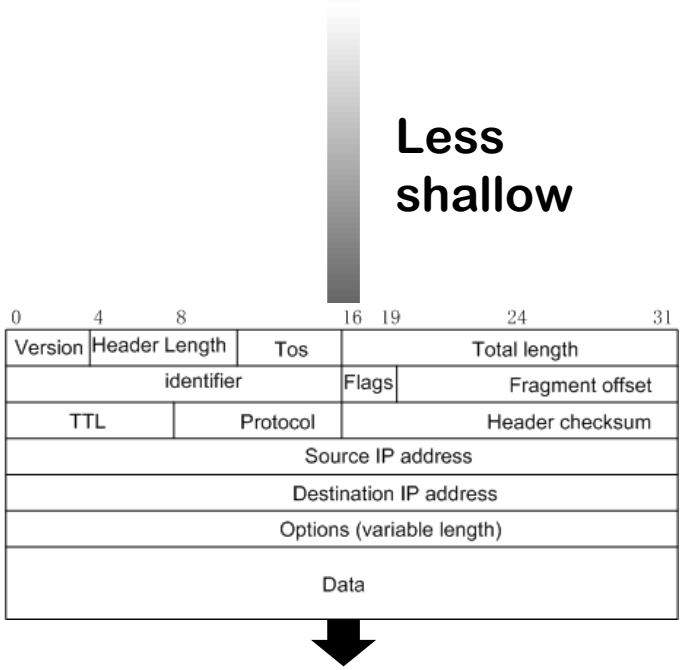
   apply random mutations to valid inputs

   - Tools: Radamsa, zzuf, …

3. **Generation-based** aka **grammar-based**

   generate semi-well-formed inputs
   based on knowledge of file format or protocol

   - With tailor-made fuzzer for a specific input format, eg FrankenCert , or a generic fuzzer configured with a grammar
   - Pro: can reach 'deeper' bugs than 1 & 2 ☺
   - Con: but lots of work to construct fuzzer or grammar ☹
   - Tools: SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, …



All fuzzers use variations & combinations of these 3 approaches

Better observation – to detect security flaws

- Instrumenting code with memory-safety checks using
  - ASan (AddressSanitizer)
  - MSan (MemorySanitizer)
  - valgrind
- Or using IdSan, identity-based memory sanitizer, made by Jos Craaijo in research internship @ Riscure with Alexandru Geana

But even without any instrumentation for better observation fuzzing can give great results!

Observation can also be used to improve test case generation, as we come to later

Fuzzing successes: payment terminals

An extended length APDU can crash contactless payment terminals

APDU Response		
Body	Trailer	
Data Field	SW1	SW2

Found accidentally, without even trying to fuzz

[Jordi van den Breekel, A security evaluation and proof-of-concept relay attack on EMV contactless transactions, MSc thesis, 2014]

Security evaluators are still not doing such basic testing!

NFC flaws let researchers hack an ATM by waving a phone

Flaws in card-reader technology can wreak havoc with point-of-sale systems and more.

ANDY GREENBERG, WIRED.COM 6/25/2021, 9:52 PM

Fuzzing successes: fonts

Google Project Zero found 29 flaws by fuzzing fonts in the Windows kernel

Tracker ID	Memory access type at crash	Crashing function	CVE
1022	Invalid write of *n* bytes (memcpy)	`usp10!otlList::insertAt`	CVE-2017-0108
1023	Invalid read / write of 2 bytes	`usp10!AssignGlyphTypes`	CVE-2017-0084
1025	Invalid write of *n* bytes (memset)	`usp10!otlCacheManager::GlyphsSubstituted`	CVE-2017-0086
1026	Invalid write of *n* bytes (memcpy)	`usp10!MergeLigRecords`	CVE-2017-0087
1027	Invalid write of 2 bytes	`usp10!ttoGetTableData`	CVE-2017-0088
1028	Invalid write of 2 bytes	`usp10!UpdateGlyphFlags`	CVE-2017-0089
1029	Invalid write of *n* bytes	`usp10!BuildFSM` and nearby functions	CVE-2017-0090
1030	Invalid write of *n* bytes	`usp10!FillAlternatesList`	CVE-2017-0072

https://googleprojectzero.blogspot.com/2017/04/notes-on-windows-uniscribe-fuzzing.html

More advanced strategies for testcase generation

Game changers in test case generation

4. Whitebox approach of SAGE

   analyse source code, using symbolic execution, to construct inputs

5. Coverage-guided evolutionary fuzzing with afl

   observe execution to try to learn which mutations are interesting

   - aka greybox approach

Whitebox fuzzing with SAGE

Whitebox fuzzing with SAGE

Basic fuzzing code below is unlikely to hit the error case

```
int foo(int x) {
        y = x+3;
        if (y==13) abort(); // error
}
```

Symbolic execution reveals `x+3=13` as interesting test case

Microsoft's SAGE uses symbolic execution of x86 binaries to generate test cases.

Clever idea here: *one* symbolic execution is used to generate *many* interesting test cases

How SAGE works

```
void top(char input[4]) {
            int cnt = 0;
            if (input[0] == 'b') cnt++;
            if (input[1] == 'a') cnt++;
            if (input[2] == 'd') cnt++;
            if (input[3] == '!') cnt++;
            if (cnt >= 3) crash();
        }
```

path contraints:

$i_0 \neq$ 'b'

$i_1 \neq$ 'a'

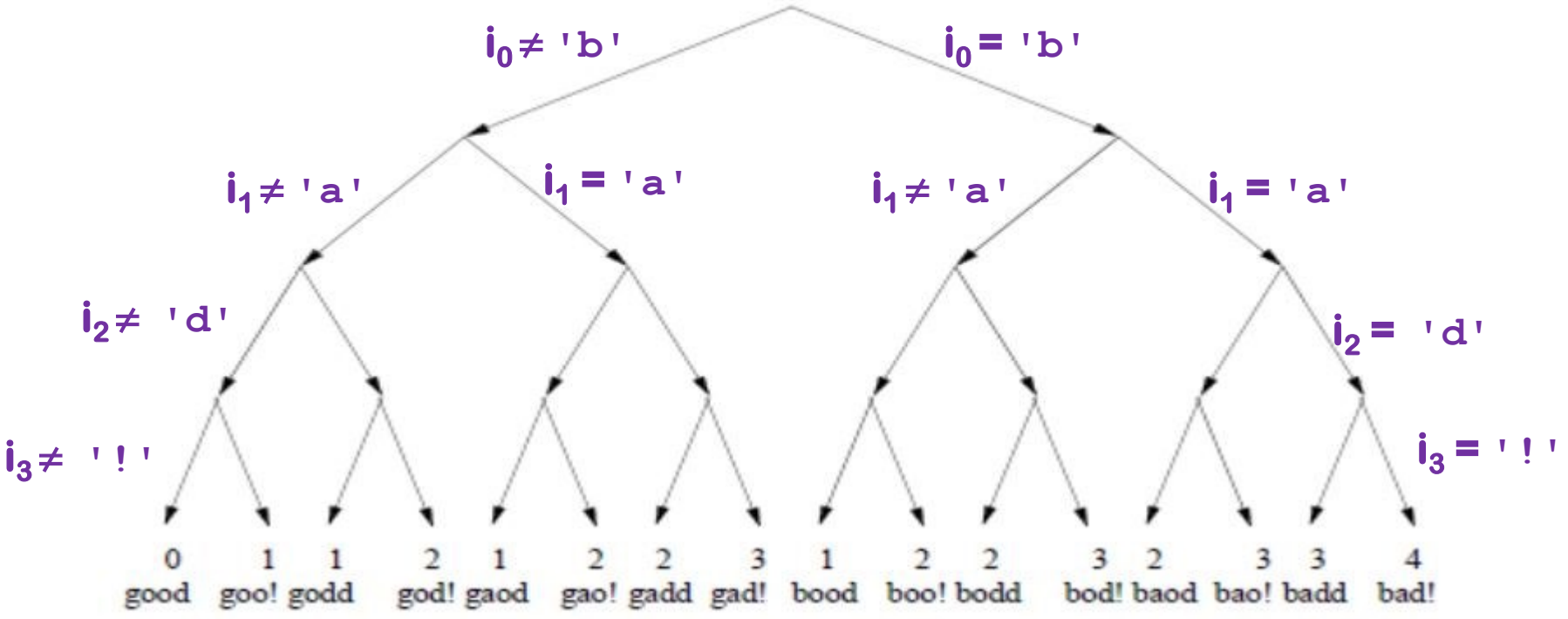
$i_2 \neq$ 'd'

$i_3 \neq$ '!'

1. Execute the code for some concrete `input`, say `'good'` resulting in a concrete execution path
2. Collect path constraints for an arbitrary symbolic input $i_0 i_1 i_2 i_3$
3. Let SMT solver (Z3, Yikes,…) try to solve all combinations of path constraints & their negations to produce additional test cases

Generating interesting inputs from path constraints

Based on this *one* execution, combining the 4 path constraints found yields $2^4 = 16$ test cases



Note: the initial execution with the input 'good' was not interesting, but some of these others are

Example of SAGE success

SAGE proved very successful at uncovering security bugs, eg

> Microsoft Security Bulletin MS07-017 aka CVE-2007-0038: Critical
>
> Vulnerabilities in GDI Allow Remote Code Execution
>
> Stack-based buffer overflow in the animated cursor code in Windows… allows remote attackers to execute arbitrary code via a large length value in the second (or later) `anih` block of a RIFF .ANI, cur, or .ico file, which results in memory corruption when processing cursors, animated cursors, and icons

Root cause: vulnerablity in PARSING of RIFF .ANI, cur, and ico-formats.

NB SAGE automatically generates inputs triggering this bug *without* knowing these formats

[Godefroid et al., *SAGE: Whitebox Fuzzing for Security Testing*, ACM Queue 2012]

[Patrice Godefroid, *Fuzzing: Hack, Art, and Science*, Communications of the ACM, 2020]

Coverage-guided evolutionary fuzzing with `afl`

(American Fuzzy Lop)

Evolutionary fuzzing with `afl` [http://lcamtuf.coredump.cx/afl]

Evolutionary approach to learn interesting mutations based on measuring execution path coverage

- Basic idea: if mutation triggers new execution paths, it is an interesting mutation to keep; if not, the mutation is discarded.
- To observe execution paths:
  - if source code is available, by instrumention added by compiler
  - if source code is not available, by running code in an emulator
- Execution path coverage represented as a 64KB bitmap, each control flow jump mapped to change in this bitmap

  Different executions could result in same bitmap, but chance is small
- Mutation strategies include: bit flips, incrementing/decrementing integers, using pre-defined interesting integer values (eg. 0, -1, MAX_INT,....) or user-supplied dictionary, deleting/combining/zeroing input blocks, ...
- The fuzzer forks the SUT to speed up the fuzzing

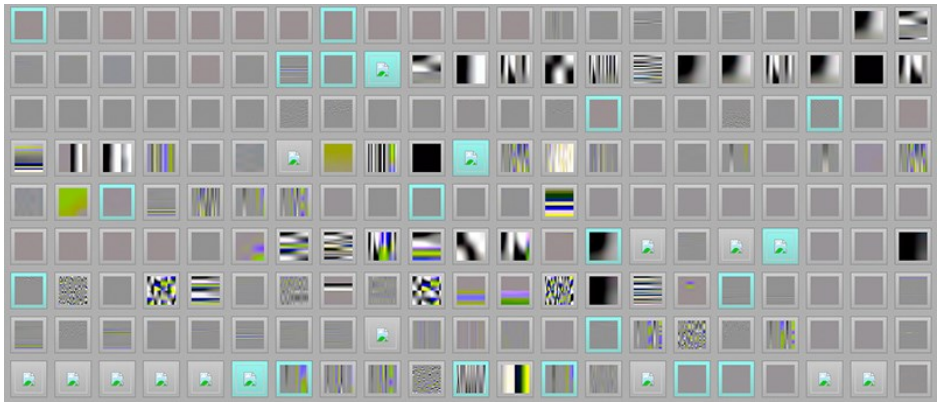
Cool example: learning the JPG file format

Fuzzing a program that expects JPG as input, starting with 'hello world' as initial test input, afl learn to produce legal JPG files

along the way producing/discovering error messages such as

- `Not a JPEG file: starts with 0x68 0x65`
- `Not a JPEG file: starts with 0xff 0x65`
- `Premature end of JPEG file`
- `Invalid JPEG file structure: two SOI markers`
- `Quantization table 0x0e was not defined`

and then JPGs like



[Source http://lcamtuf.blogspot.nl/2014/11/pulling-jpegs-out-of-thin-air.html]

Vulnerabilities found with `afl`

IJG jpeg [1]
libtiff [1] [2] [3] [4] [5]
Mozilla Firefox [1] [2] [3] [4]
Adobe Flash / PCRE [1] [2] [3] [4]
LibreOffice [1] [2] [3] [4]
GnuTLS [1]
PuTTY [1] [2]
bash (post-Shellshock) [1] [2]
pdfium [1] [2]
BIND [1] [2] [3] ...
Oracle BerkeleyDB [1] [2]
FLAC audio library [1] [2]
strings (+ related tools) [1] [2] [3] [4] [5] [6] [7]
Info-Zip unzip [1] [2]
NetBSD bpf [1]
clang / llvm [1] [2] [3] [4] [5] [6] [7] [8] ...
mutt [1]
pdksh [1] [2]
redis / lua-cmsgpack [1]
perl [1] [2] [3] [4] [5] [6] [7...]
SleuthKit [1]
exifprobe [1]
Xerces-C [1] [2] [3]

libjpeg-turbo [1] [2]
mozjpeg [1]
Internet Explorer [1] [2] [3] [4]
sqlite [1] [2] [3] [4...]
poppler [1]
GnuPG [1] [2] [3] [4]
ntpd [1] [2]
tcpdump [1] [2] [3] [4] [5] [6] [7] [8] [9]
ffmpeg [1] [2] [3] [4] [5]
QEMU [1] [2]
Android / libstagefright [1] [2]
libsndfile [1] [2] [3] [4]
file [1] [2] [3] [4]
libtasn1 [1] [2] ...
man & mandoc [1] [2] [3] [4] [5] ...
nasm [1] [2]
procmail [1]
Qt [1] [2...]
taglib [1] [2] [3]
libxmp
fwknop [reported by author]
jhead [?]
metacam [1]

libpng [1]
PHP [1] [2] [3] [4] [5]
Apple Safari [1]
OpenSSL [1] [2] [3] [4] [5] [6] [7]
freetype [1] [2]
OpenSSH [1] [2] [3]
nginx [1] [2] [3]
JavaScriptCore [1] [2] [3] [4]
libmatroska [1]
lcms [1]
iOS / ImageIO [1]
less / lesspipe [1] [2] [3]
dpkg [1] [2]
OpenBSD pfctl [1]
IDA Pro [reported by authors]
ctags [1]
fontconfig [1]
wavpack [1]
privoxy [1] [2] [3]
radare2 [1] [2]
X.Org [1] [2]
capnproto [1]
djvulibre [1]

Challenges in fuzzing

- **Fuzzers won't get past integrity & security checks**
  e.g. CRC checksums or digital signatures
  Such checks have to be removed from SUT, or
  fuzzer has to be tailor-made to add correct checksums and signatures
- **Protocols very low in protocol stack (eg Bluetooth) are hard to fuzz**
  Implementations baked into chips cannot be used for sending malformed packets
- **Embedded systems are harder to fuzz**
  - Harder/impossible to instrument to observe execution,
  - possibly harder to detect crashes,
  - slower to fuzz

  Research idea: side-channel info could be used to guide fuzzing
- **Stateful programs are harder to fuzz**

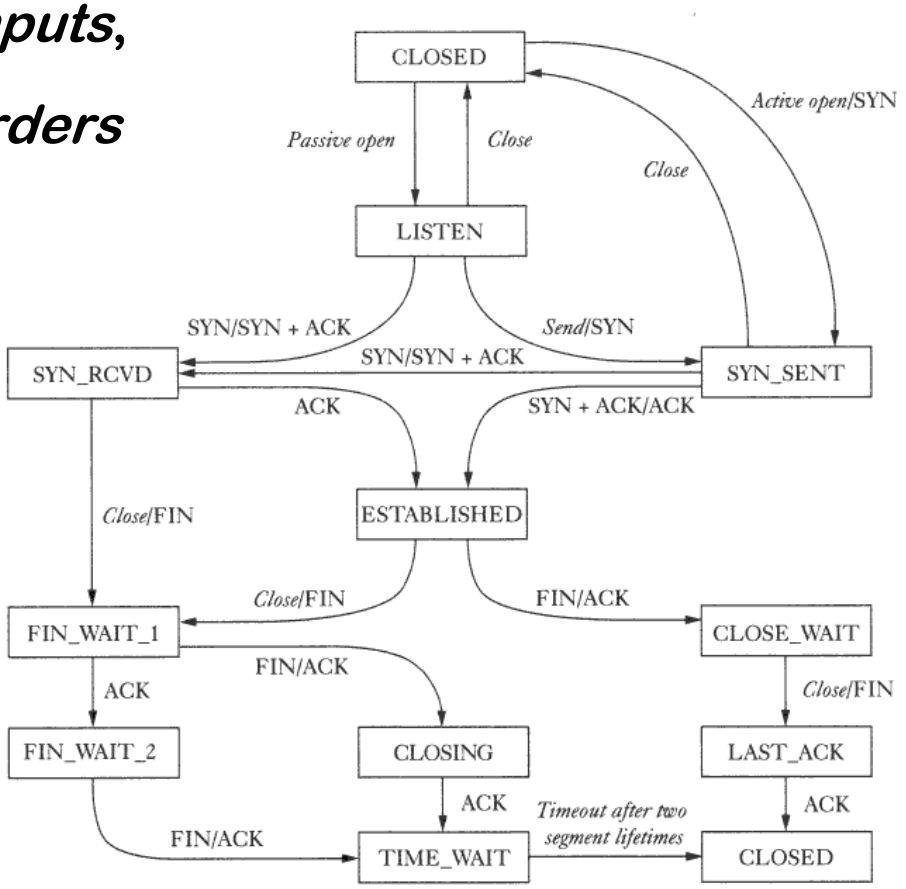
Fuzzing stateful programs

Fuzzing stateful programs

- Many programs/protocols involve *sequences* of input
- Problematic for fuzzers:

  not only *many possible inputs*,

  also *many possible orders*



Approaches to fuzzing stateful programs

- Grammar/specification/model-based

  specify a state machine model to guide fuzzer to 'deeper' states

  - Tools: BooFuzz

- Random mutation

  of a trace consisting of multiple inputs

- *-guided evolutionary fuzzing

  Combine random mutation with some coverage measure

  - Tools: AFLnet

  Instead of measuring execution path coverage we'd like to measure state coverage. But can we observe the state?

- State machine inference

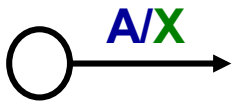
  To reduce the state-space explosion, we can restrict ourselves to *incorrect* sequences of *correct* messages & infer state behaviour

  - Tools: LearnLib

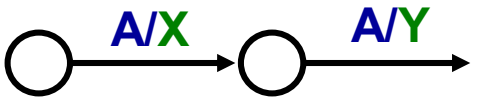
State machine inference by fuzzing

Just try out many sequences of inputs, and observe outputs

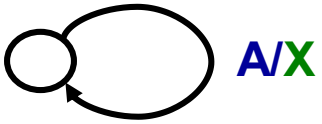
Suppose input A results in output X



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

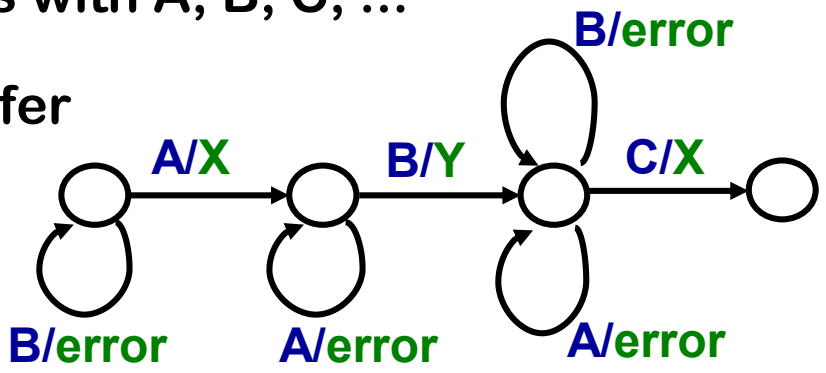


- If second input A results in the *same* output X



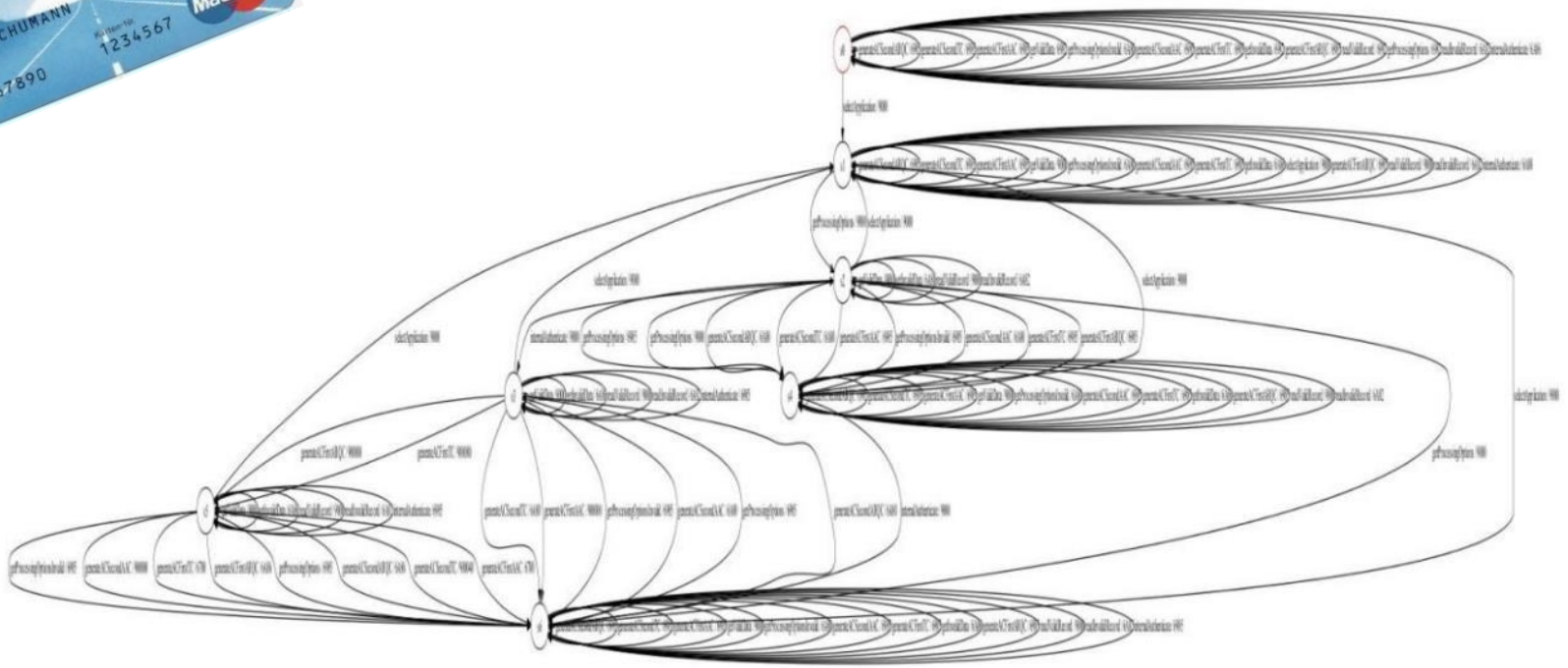
Now try more sequences of inputs with A, B, C, ...

to e.g. infer

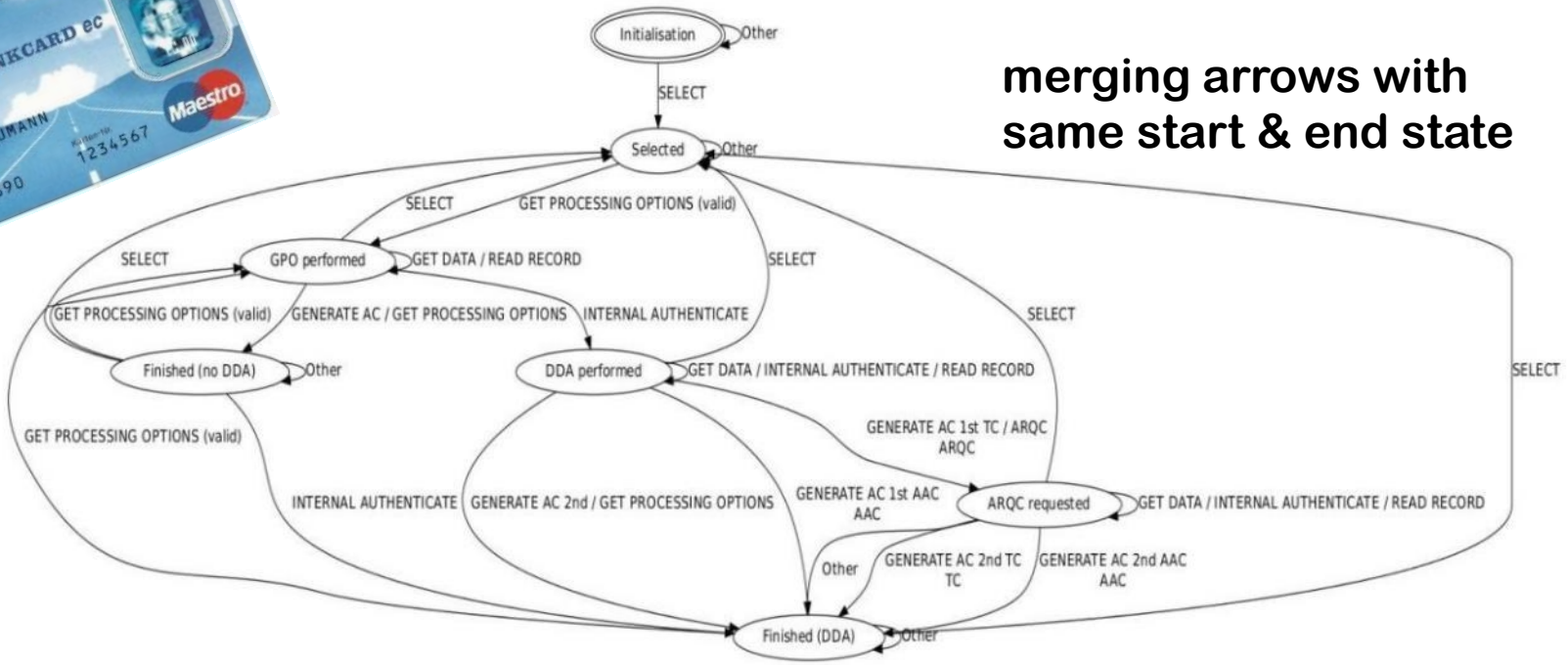


The inferred state machine is an under-approximation of real system

State machine inference of Maestro card



State machine inference of Maestro card



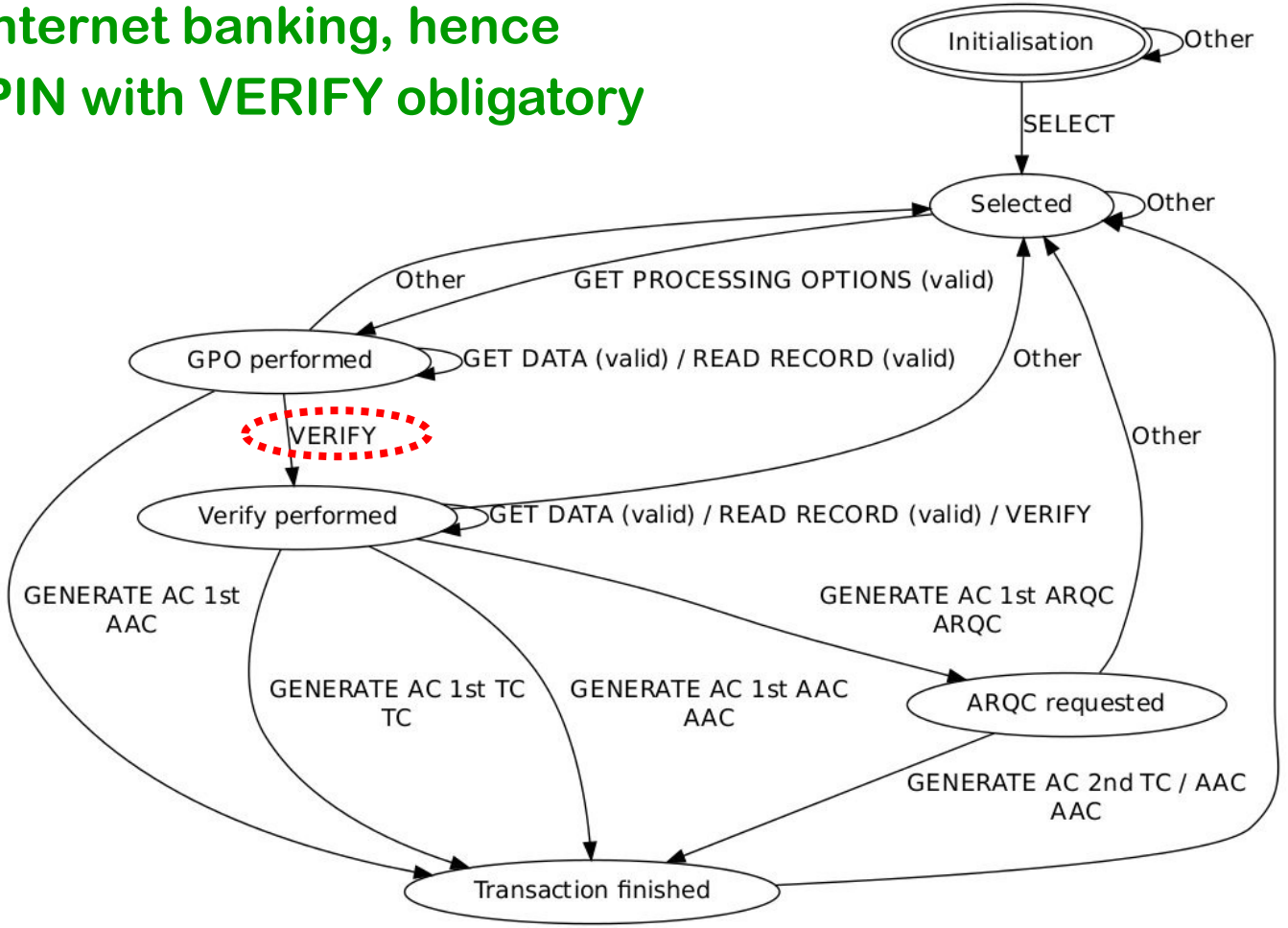
merging arrows with same start & end state

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

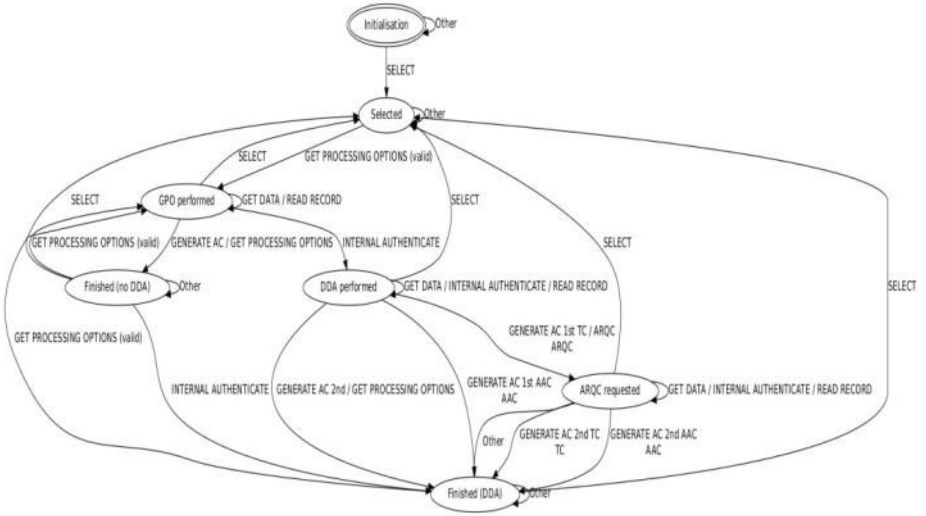
[Fides Aarts et al., Formal models of bank cards for free, SECTEST 2013]

SecureCode application on Rabobank card

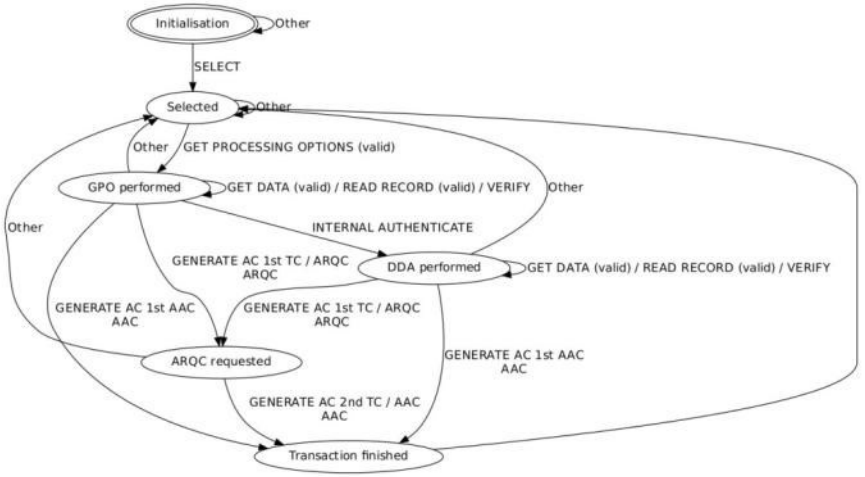
used for internet banking, hence entering PIN with VERIFY obligatory



Understanding & comparing EMV implementations



Volksbank Maestro implementation



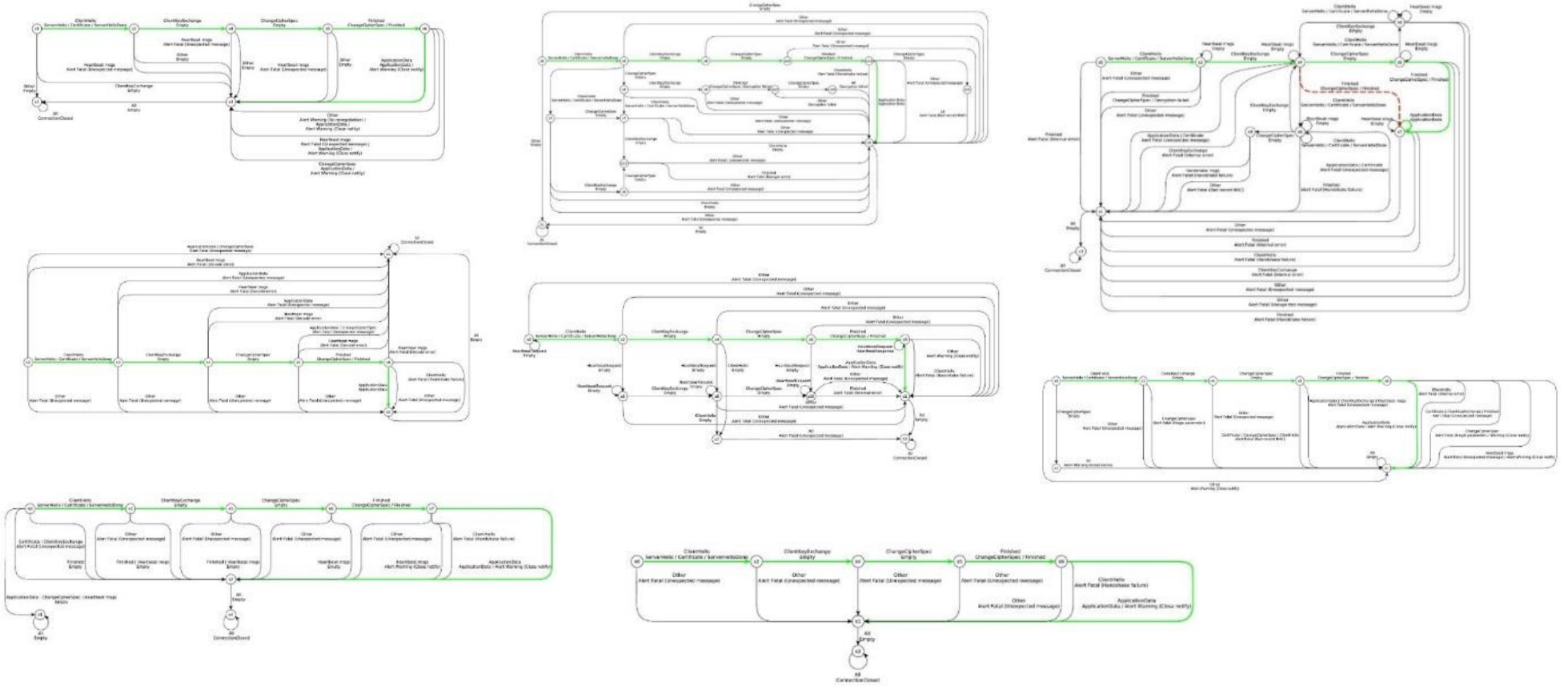
Rabobank Maestro implementation

Are both implementations correct & secure? And compatible?

Presumably they both pass a Maestro compliance test-suite…

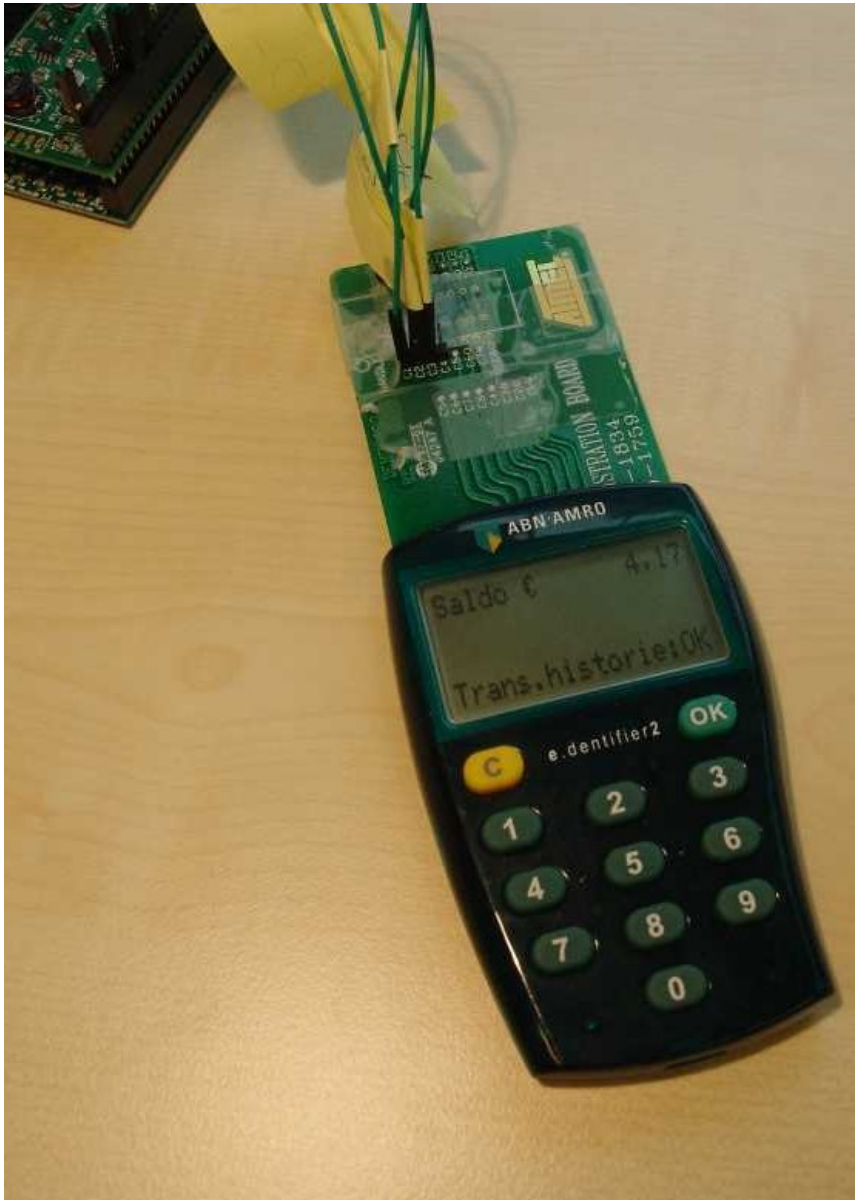
So some paths (and maybe some states) are superfluous?

Protocol state fuzzing of TLS

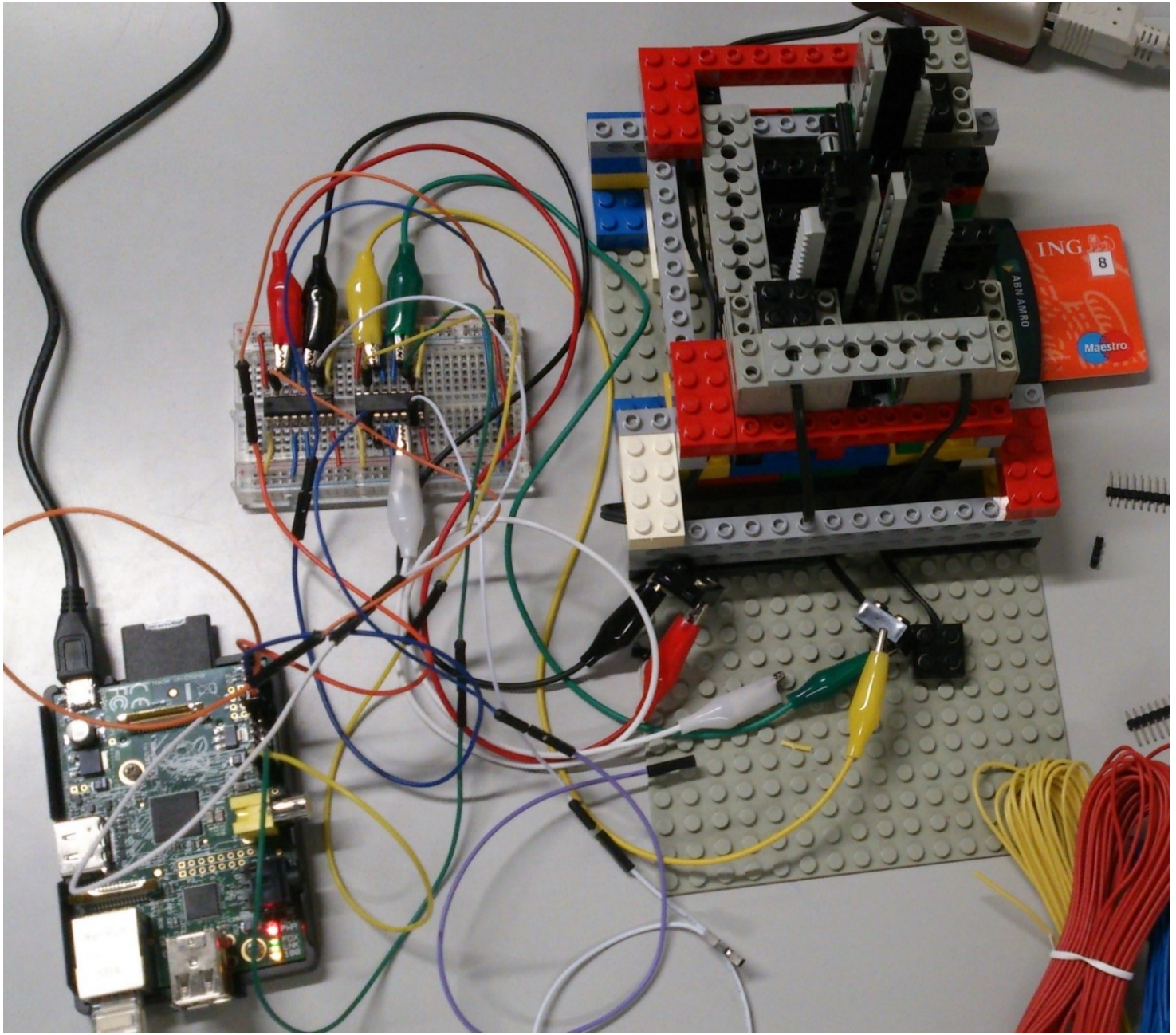


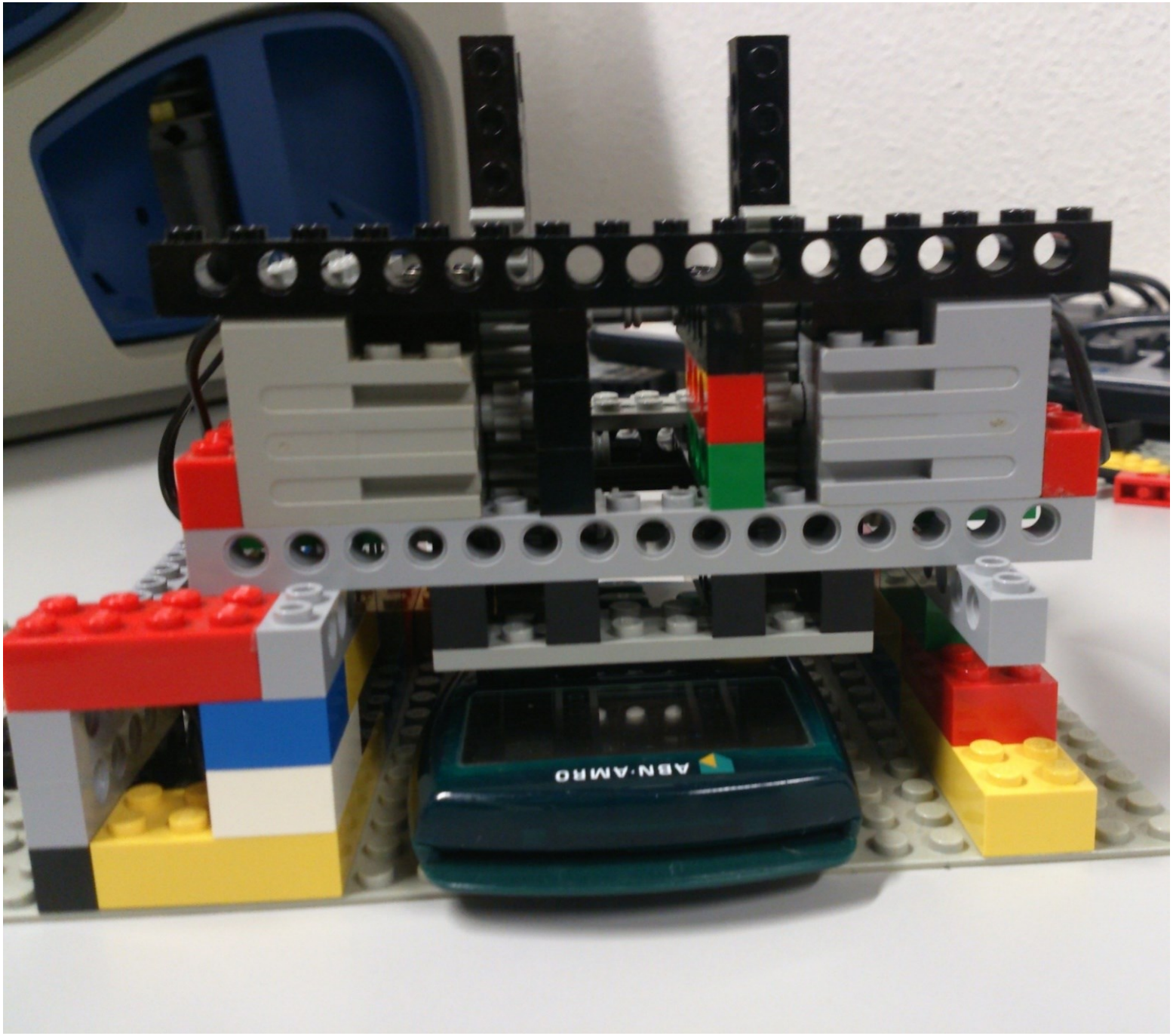
[Joeri de Ruiter et al., Protocol state fuzzing of TLS implementations, Usenix Security 2015]

Protocol state fuzzing of e.dentifier

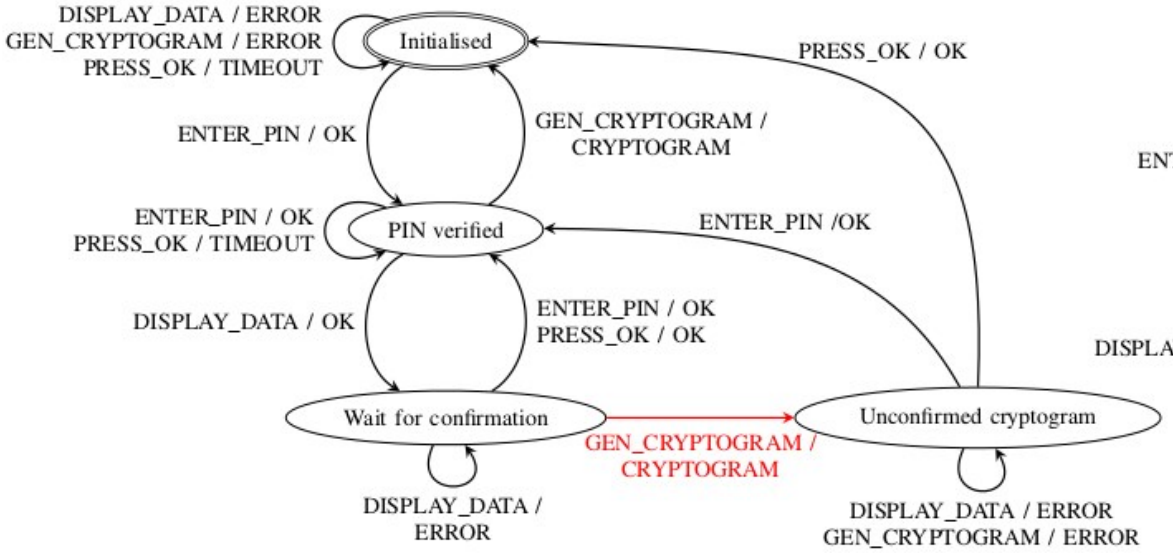


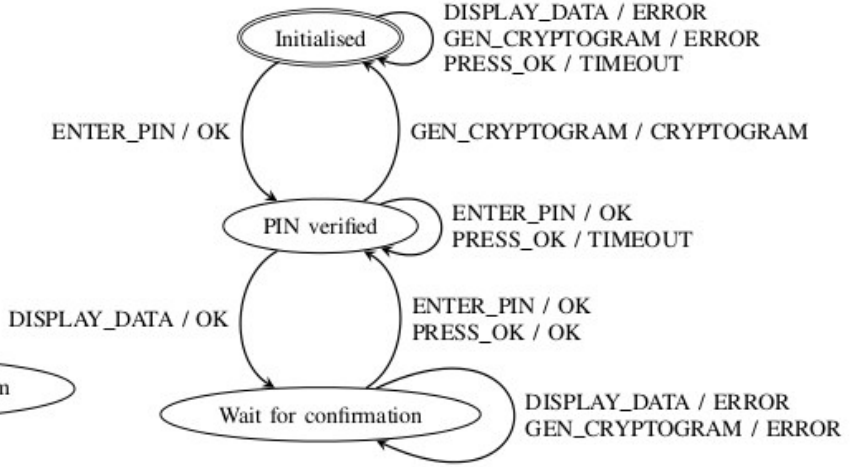
Operating the keyboard using LEGO

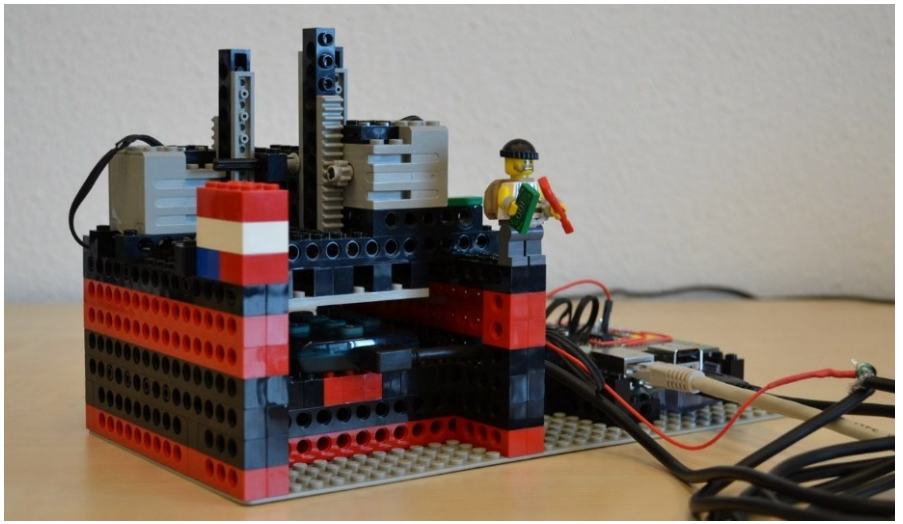

ING
8
ABN AMRO
Maestro


ABN·AMRO

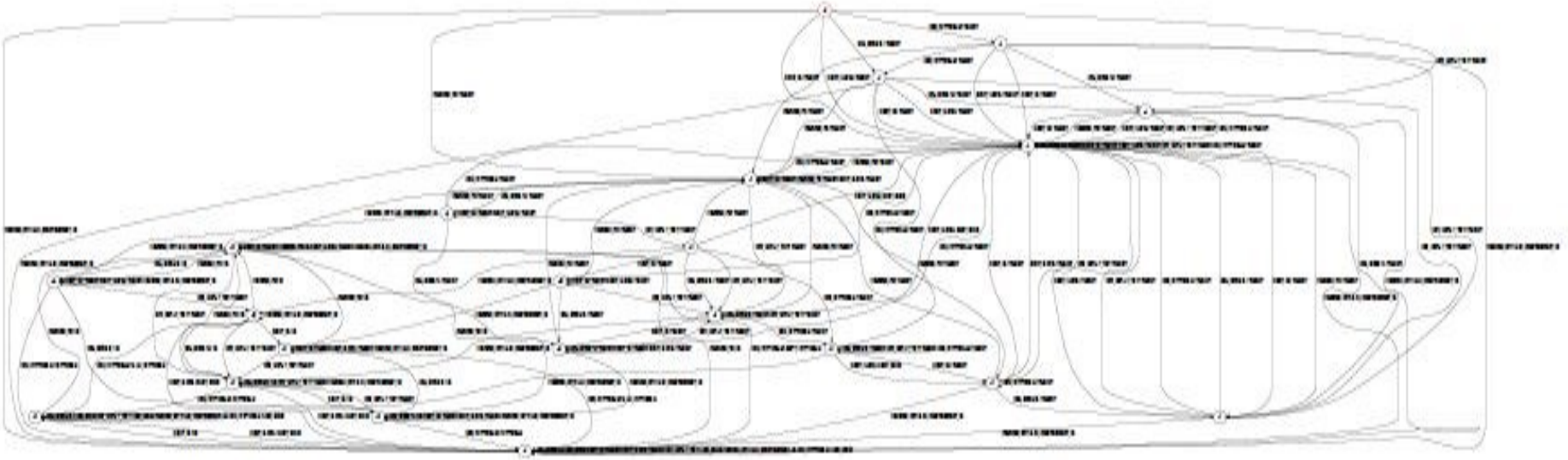
State machines of old vs new e.dentifier2







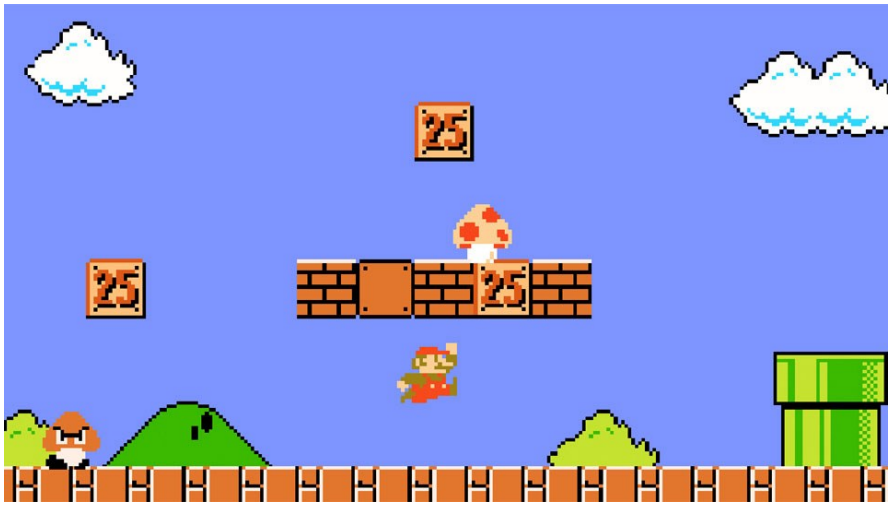
Would you trust this to be secure?



More detailed inferred state machine, using richer input alphabet.

Other strategies in evolutionary fuzzing

Instead of maximizing path/code coverage, a fuzzer can also let inputs evolve to maximize some other variable or property



Eg the x-coordinate of Super Mario

Code has to instrumented to let fuzzer observe that property

[Aschermann et al., *IJON: Exploring Deep State Spaces via Fuzzing,* IEEE S&P 2020]

https://www.youtube.com/watch?v=3PyhXlHDkNI

Summary/Conclusions

- Mutational Fuzzing
  Little work, but likely to find only few new paths
- Generational aka grammar-based fuzzing
  More work, but more likely to find more paths
- Whitebox fuzzing with eg. SAGE
  In principle, hits every possible execution path. But requires source code & has practical limits whe code gets too complex or involves system calls.
- Coverage-guided evolutionary / greybox fuzzing with eg. `afl`
  In practice, discovers many interesting execution paths
- Practical challenges/research opportunities
  - better instrumentation to observe & guide fuzzing?
  - coping with statefulness?
  - coping with embedded systems?

Beware: terminology for various forms of fuzzing is messy

More examples:
Mutational fuzzing

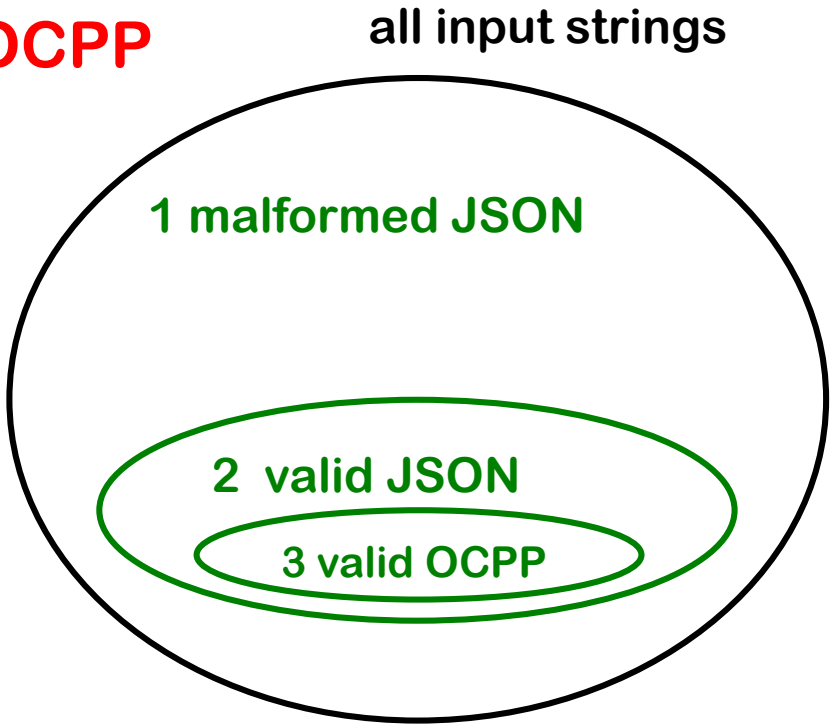
Example: Fuzzing OCPP [research internship Ivar Derksen]

- OCPP is a protocol for charge points to talk to a back-end server
- OCPP can use XML or JSON messages

Example message in JSON format

```
{ "location": NijmegenMercator215672,
  "retries": 5,
  "retryInterval": 30,
  "startTime": "2018-10-27T19:10:11",
  "stopTime":  "2018-10-27T22:10:11"  }
```

Example: Fuzzing OCPP



Classification of messages into

1. malformed JSON/XML
   eg missing quote, bracket or comma
2. well-formed JSON/XML, but not legal OCPP
   eg with field names not in OCPP specs
3. well-formed OCPP

can be used for a simple test oracle:

- The application should never crash
- Malformed messages (type 1 & 2) should generate generic error response
- Well-formed messages (type 3) should not

Note: this does not require *any* understanding of the protocol semantics!
Figuring out correct responses to type 3 would require that.

Test results with fuzzing OCPP server

- Mutation fuzzer generated 26,400 variants from 22 example OCPP messages in JSON format
- Problems spotted by this simple test oracle:
  - 945 malformed JSON requests (type 1) resulted in malformed JSON response

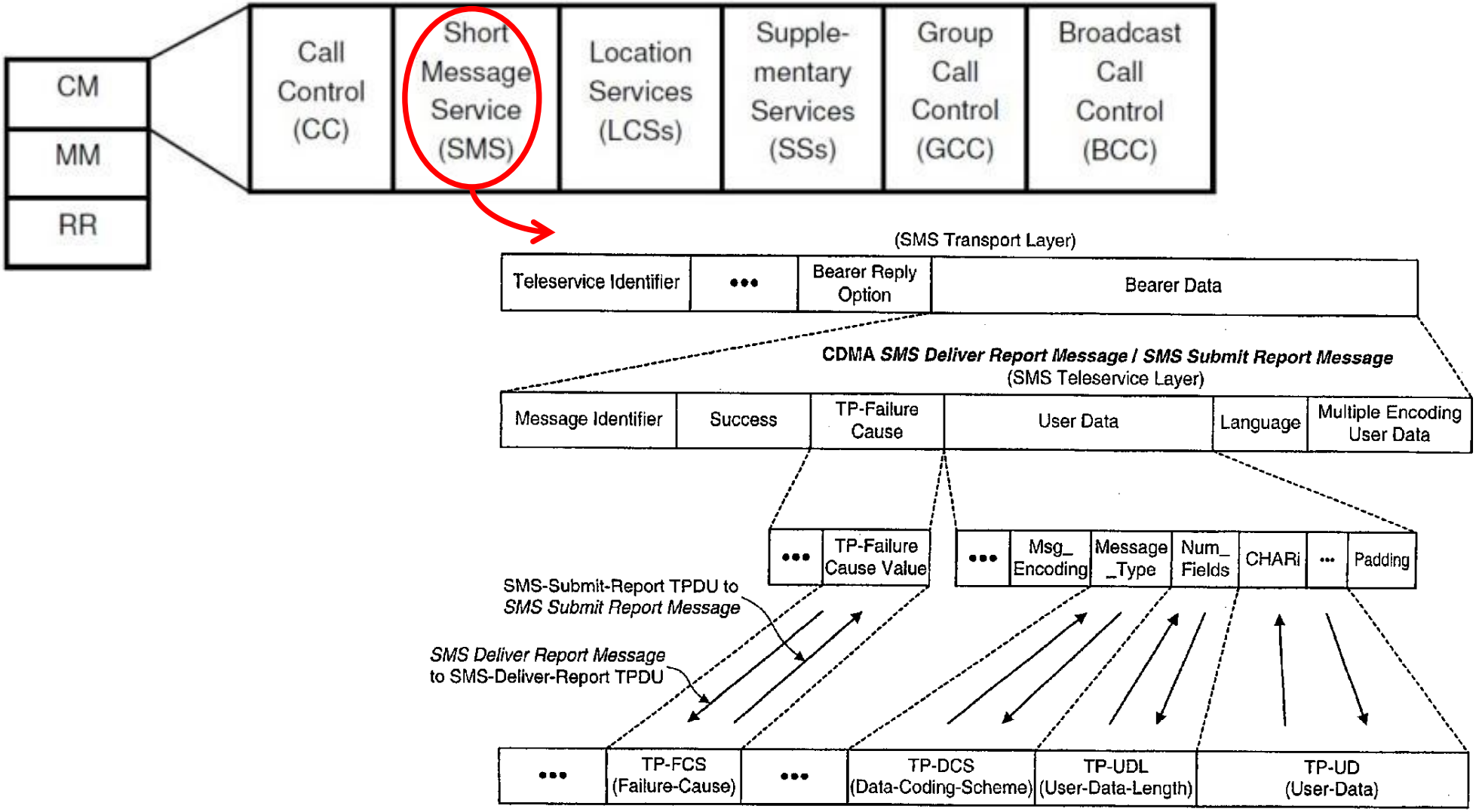
    Server should never emit malformed JSN!
  - 75 malformed JSON requests (type 1) and 40 malformed OCPP requests (type 2) result in valid OCPP response that is not an error message

    Server should not process malformed requests!
- One root cause of problems: the Google's `gson` library for parsing JSON by default uses lenient mode rather than strict mode
  - *Why does `gson` even have a lenient mode, let alone by default?*
- Fortunately, `gson` is written in Java, not C(++), so these flaws do not result in exploitable buffer overflows

More examples:
Generational fuzzing

Example: generation-based fuzzing of GSM

GSM is a extremely rich & complicated protocol



SMS message fields

Field	size
Message Type Indicator	2 bit
Reject Duplicates	1 bit
Validity Period Format	2 bit
User Data Header Indicator	1 bit
Reply Path	1 bit
Message Reference	integer
Destination Address	2-12 byte
Protocol Identifier	1 byte
Data Coding Scheme (CDS)	1 byte
Validity Period	1 byte/7 bytes
User Data Length (UDL)	integer
User Data	depends on CDS and UDL

Example: GSM protocol fuzzing

Lots of stuff to fuzz!

We can use a USRP

with open source cell tower software (OpenBTS)

to fuzz any phone

Example: GSM protocol fuzzing

Fuzzing SMS layer reveals weird functionality in GSM standard and in phones

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones

- **eg possibility to receive faxes (!?)**

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

Example: GSM protocol fuzzing

Malformed SMS text messages showing raw memory contents, rather than content of the text message

random garbage in SMS message

names of games installed on phone showing up *inside* SMS messages

Our results with GSM fuzzing

- Lots of success to DoS phones:
  phone crashes, disconnects from network, stops accepting calls,…
  - eg requiring reboot or battery removal to restart, to accept calls again, or to remove weird icons
  - after reboot, the network might redeliver the SMS message, if no acknowledgement was sent before crashing, re-crashing phone

  But: not all these SMS messages could be sent over real network
- There is surprisingly little correlation between problems and phone brands & firmware versions
  - how many implementations of the GSM stack did Nokia have?
- *The scary part: what would happen if we fuzz base stations?*

[Fabian van den Broek, Brinio Hond and Arturo Cedillo Torres, Security Testing of GSM Implementations, ESSOS 2014]

[Mulliner et al., SMS of Death, USENIX 2011]

More SMS problems

effective.

Power

لُلُصّبُلُلصّبُررً ◌ّ ◌ّh ◌ّ ◌ّ

卐

Example dangerous SMS text message

- **This message *can* be sent over the network**
- **Different characters sets & characters encoding are a constant source of problems.**