Offense meets defense in software security:

Fuzzing

Erik Poll

Digital Security group

Radboud Universiteit Nijmegen

Software (in)security

Software as weakest & most dangerous link

I don't have an Achilles' heel, I have an Achilles' body

-- Woody Allen

Software

Software is key to the success of computers

but also key to security failures of computers

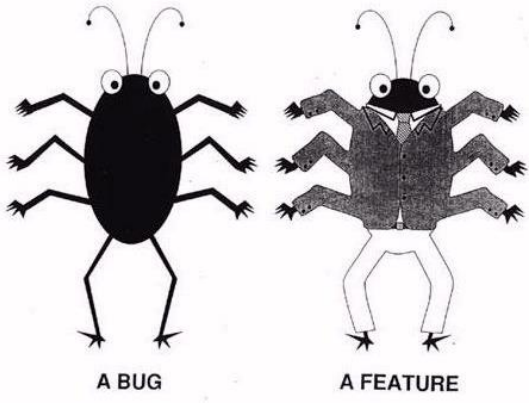
If something contains software, it can usually be hacked

Why?

Why is software so hard to secure?

- **Complexity**

  This causes bugs
  but also unwanted (interaction of) features



- **Programmability**

  The power & flexibility of software is great for users,
  but also for attackers

- **More security controls, more software, more security problems?**

  Access control, intrusion detection, VPNs, firewalls, crypto …
  all introduce more software… and hence more vulnerabilities

  Eg. recall HeartBleed or Crowdstrike

Fuzzing

The idea

```
Please enter your username

>
```

How would you test the security of this C/C++ program?

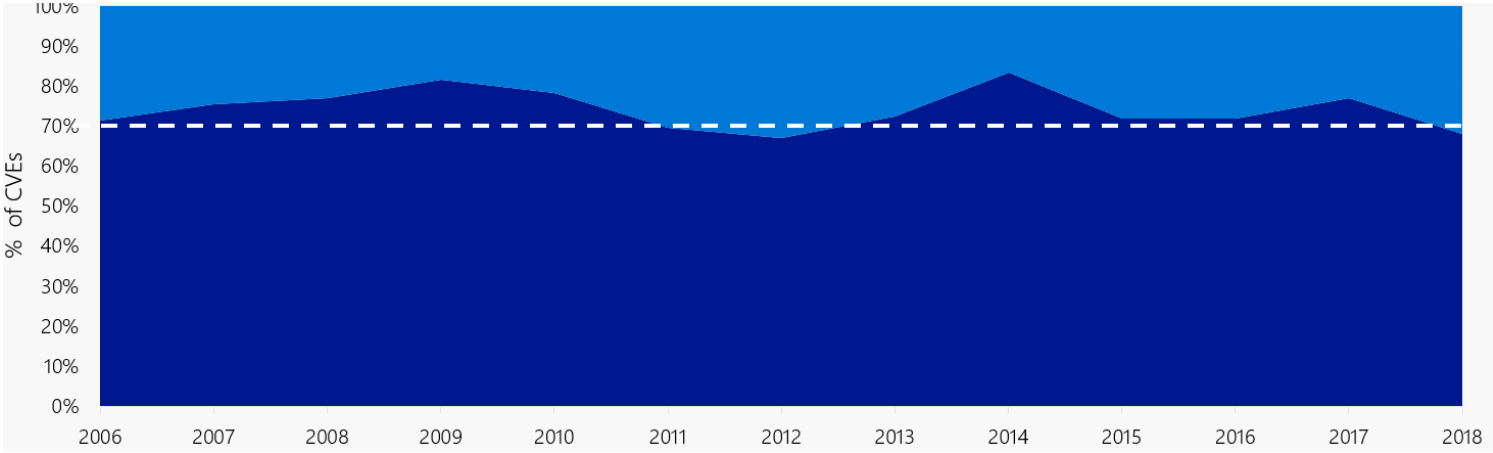
1. Try `admin,admin`
2. Ridiculous long inputs, say a few MB

   If there's a buffer overflow, this is likely to trigger a SEG FAULT
3. %x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x

   To see if there is a format string vulnerability
4. Other malicious inputs, depending on back-end APIs used:
   ' DROP TABLES; ../../../dev/urandom, <script>… </script>
   to test for SQL injections, path traversal, XSS, XXE …

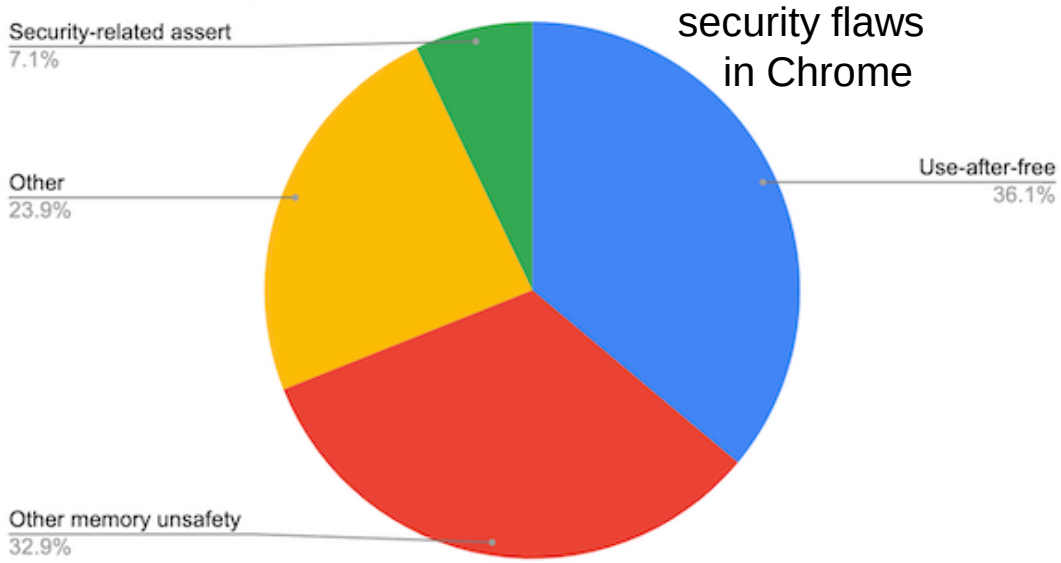
2-4 can be automated by fuzzing:
(semi)automatically generate 'random' inputs to cause problems

Memory corruption



memory safety vs non-memory safety bugs at Microsoft

Of 130 critical security flaws in Chrome, 5 were *not* due to memory-corruption



Only solution: move to memory safe languages, eg Rust

History of fuzzing (aka *fuzz testing* or *monkey testing*)

1. Random fuzzing

   random – long or short – inputs

   possibly with special characters & keywords: `\0 %x %n DROP`

2. Mutational fuzzing

   mutate normal inputs (eg network packet)

3. Grammar-based fuzzing

   use grammar of input format to generate inputs & mutate these

4. SAGE: using symbolic execution aka white box

5. *The real breakthrough!*

   afl: using evolution aka grey box

To help with finding memory-corruption bugs, code can be instrumented with sanitisers (ASan, MSan, UBSan, valgrind, …)

1. Random Fuzzing

Random fuzzing of UNIX utilities [1988]

Barton Miller let students fuzz UNIX utilities as part of his Operating System course

"While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive."

command that began with 'o/%8f.'" Since it does not find such a com- in seven bits. The equation processor (eqn) depends on this assump- emacs, receives the suspend character, it appears as an ordinary con-

required to make such strong cits the same standards of care hundred Unix workstations), w

https://pages.cs.wisc.edu/~bart/fuzz/

Barton P. Miller, Lars Fredriksen, Bryan So,
An empirical study of the reliability of UNIX utilities,
Communications of the ACM, Vol 33, Issue 12, 1990

Utility	VAX	Sun	HP	i386	AIX 1.1
adb	•○	•	•	○	–
as	•			•	•
awk					
bc				•○	
bib			–	–	–
calendar				–	
cat					
cb	•		•	•	○
cc					
/lib/ccom				–	–
checkeq				–	
checknr				–	–
col	•○	•	•	•○	•
colcrt				–	–
colrm				–	–
comm					
compress					–
/lib/cpp					
csh	•○	○	○	–	○
dbx		•	–	–	
dc				○	
deqn		•	–	–	–
deroff	•	•	•		•
diction	•	–	•		–
diff					
ditroff	•○	•	–	–	–
dtbl			–	–	–
emacs	•	•	○	–	–
eqn		•	•	•	
expand					–
f77	•		–	–	–
fmt					
fold					–
ftp	•	•	•	–	•
graph					–
grep					
grn			–	–	–
head					–
ideal			–	–	–
indent	•○	•○	•	–	–
join		•			
latex			–	–	–
lex	•	•	•	•	•
lint					
lisp		–		–	–
look	•	○	•	•	–

Table 2: List of Utilities Tested and the Systems on which They Were Tested (part 1)

*• = utility crashed, ○ = utility hung, * = crashed on SunOS 3.2 but not on SunOS 4.0, – = utility unavailable on that system. N.B. join crashed only on SunOS 4.0, not 3.2.*

Four decades later, OS utilities still suck

Google Project Zero found 119 CVEs 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

2. Mutational Fuzzing

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"  }
```

Fuzzing OCPP server

Mutational fuzzer generated 26,400 variants from 22 sample OCPP messages taken from the spec

Bugs detected:

- **945 malformed JSON requests resulted in malformed JSON response**
- **75 malformed JSON requests and 40 malformed OCPP requests resulted in valid OCPP response that is not an error message.**

Contributing root cause of problems: 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?***

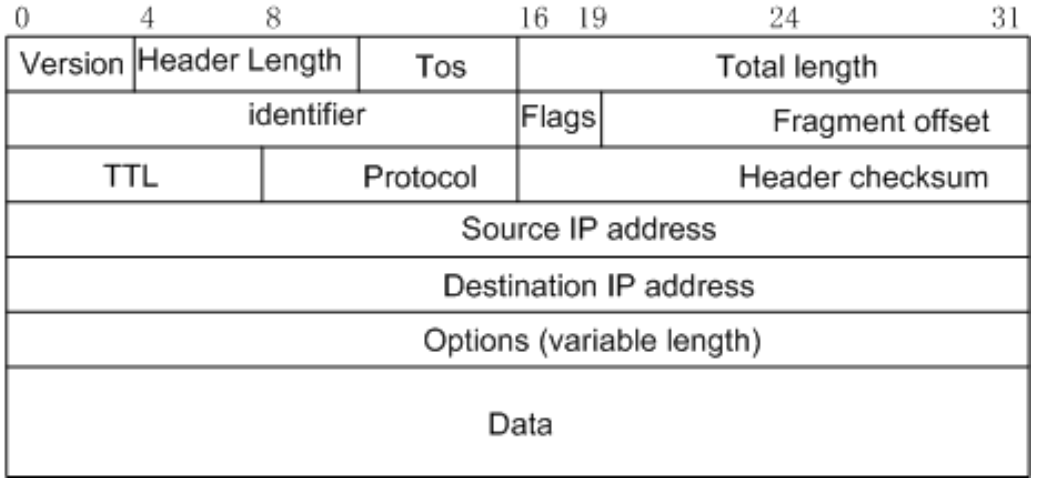
Fortunately, `gson` is written in Java, not C(++), so these flaws do not result in exploitable buffer overflows

2. Grammar-based Fuzzing

Grammar-based fuzzing

Generate malformed inputs that hit corner cases,
based on knowledge of input format/protocol

Eg using
- **regular expression**
- **context free grammar, or**
- **some other description**

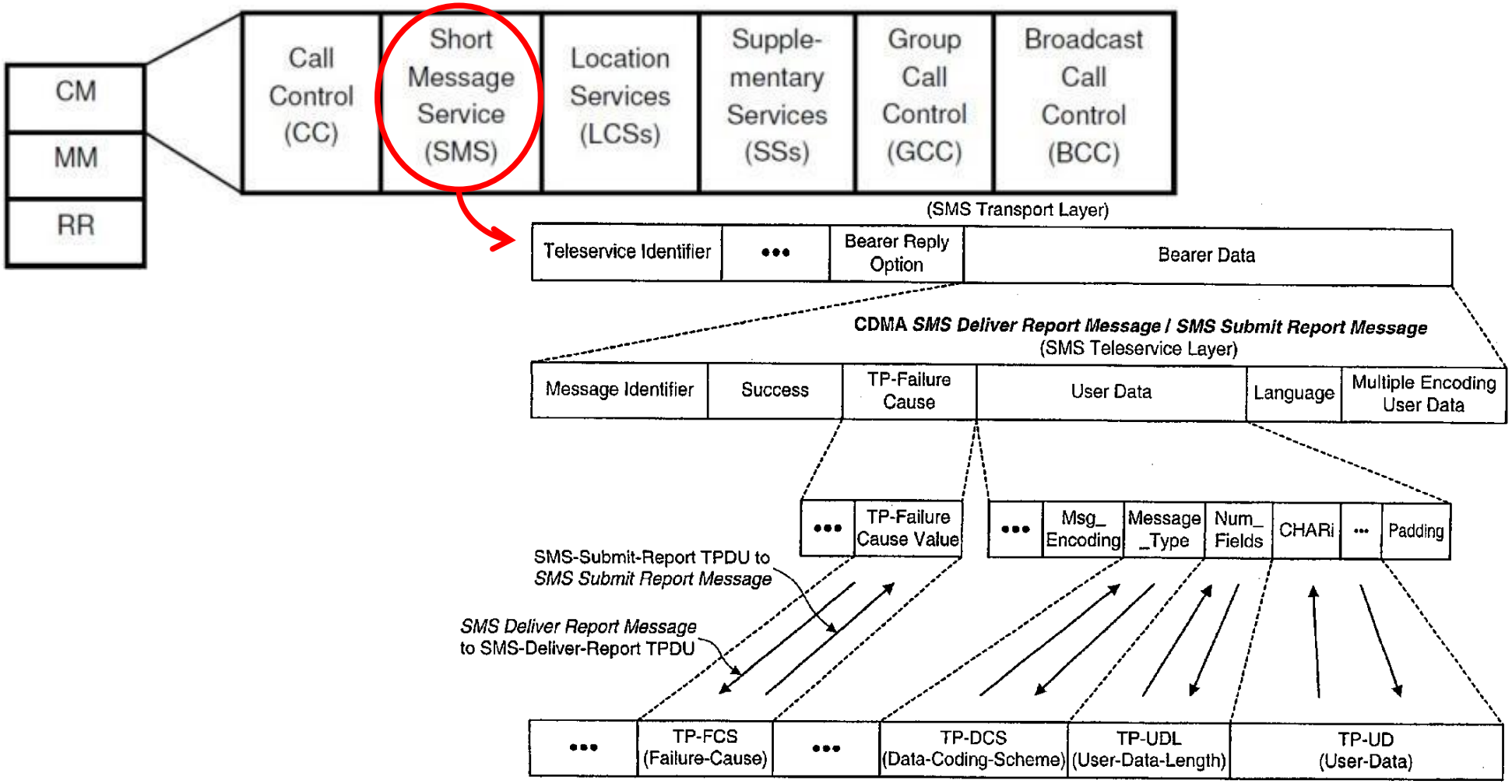


Tools: SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, ...

Also some commercial tools with built-in support for specific protocols

Example: grammar-based fuzzing of GSM

GSM is a rich & complicated protocol



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

Example: grammar-based fuzzing of GSM

We can fuzz phones over the the air using a USRP

with open source cell tower software (OpenBTS)

Example: grammar-based fuzzing of GSM

Fuzzing SMS layer of GSM reveals bugs & weird functionality

Example: grammar-based fuzzing of GSM

Fuzzing SMS layer of GSM reveals bugs & weird functionality

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

Example: grammar-based fuzzing of GSM

Many malformed SMS message crashed phones

Or triggered the phone to display raw memory content

Eg names of games would appear *inside* an SMS

Other SMS problems

iPhone /

This text message called the 'Unicode of Death' will crash your iPhone

By Jacques Coetzee: Staff Reporter on 28 May, 2015

effective.

Power

لُلُصّبُلُلصّبُررً ◌ّ ◌h ◌ّ ◌ّ

卐

Example dangerous SMS text message

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

4. White-box Fuzzing

Whitebox fuzzing using SAGE

SAGE from Microsoft Research uses symbolic execution of x86 binaries to generate test cases.

Key idea: analysis of code can suggest interesting test inputs

What are good inputs to test for code below?

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

Whitebox fuzzing

What are good inputs to test code below?

```
m(int x,y){
    x = x + y;
    y = y - x;
    if (2*y > 8) { ...
                 }
    else if (3*x < 10){ ...
                       }
  }
```

Symbolic execution

```
m(int x,y){
    x = x + y;
    y = y - x;
    if (2*y > 8) { ...
                 }
    else if (3*x < 10){ ...
    }                   }
```

Suppose `x` = *N and* `y` = *M.*

`x` *becomes N+M*

`y` *becomes M - (N+M) = -N*

*if-branch taken if 2 * -N > 8, i.e. N < -4*

2nd if-branch taken if
*N ≥ -4 AND 3 * (M+N) < 10*

Given a set of constraints, an SMT solver (Yikes, Z3, ...)
produces values that satisfy it, or proves that it are not satisfiable.

This generates test data (i) *automatically* and (ii) *with good coverage*

Whitebox fuzzing using SAGE

SAGE succcesfully used on Office and Windows 7 & later.
Sample bug found

Microsoft Security Bulletin MS07-017 aka CVE-2007-0038: Critical

Vulnerabilities in GDI Could 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 anih block of a RIFF .ANI, cur, or .ico file, that results in memory corruption when processing cursors, animated cursors and icons

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

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

5. Evolutionary fuzzing

aka

Grey box fuzzing

Evolutionary Fuzzing

Aka greybox fuzzing or coverage-guided fuzzing

Pioneered by `afl`

Starting with some sample inputs,

1. Make random mutations to some input
2. Observe the effect on the execution

   Do we see a new code path?
3. Keep the 'interesting' mutations to let them evolve further
4. Repeat

alf: observing jumps to find interesting inputs/input changes

input

code

line	instruction
1	**JMP 6**
2	..
3	..
4	..
5	**JZ (Jump If Zero) 7**
6	..
7	*arraycopy (dst, input[i..j]);*
8	
9	..
10	**JCXZ 2**
11	..
12	..
13	*println (part of input);*
14	..
15	**JNE 103131**
16	..

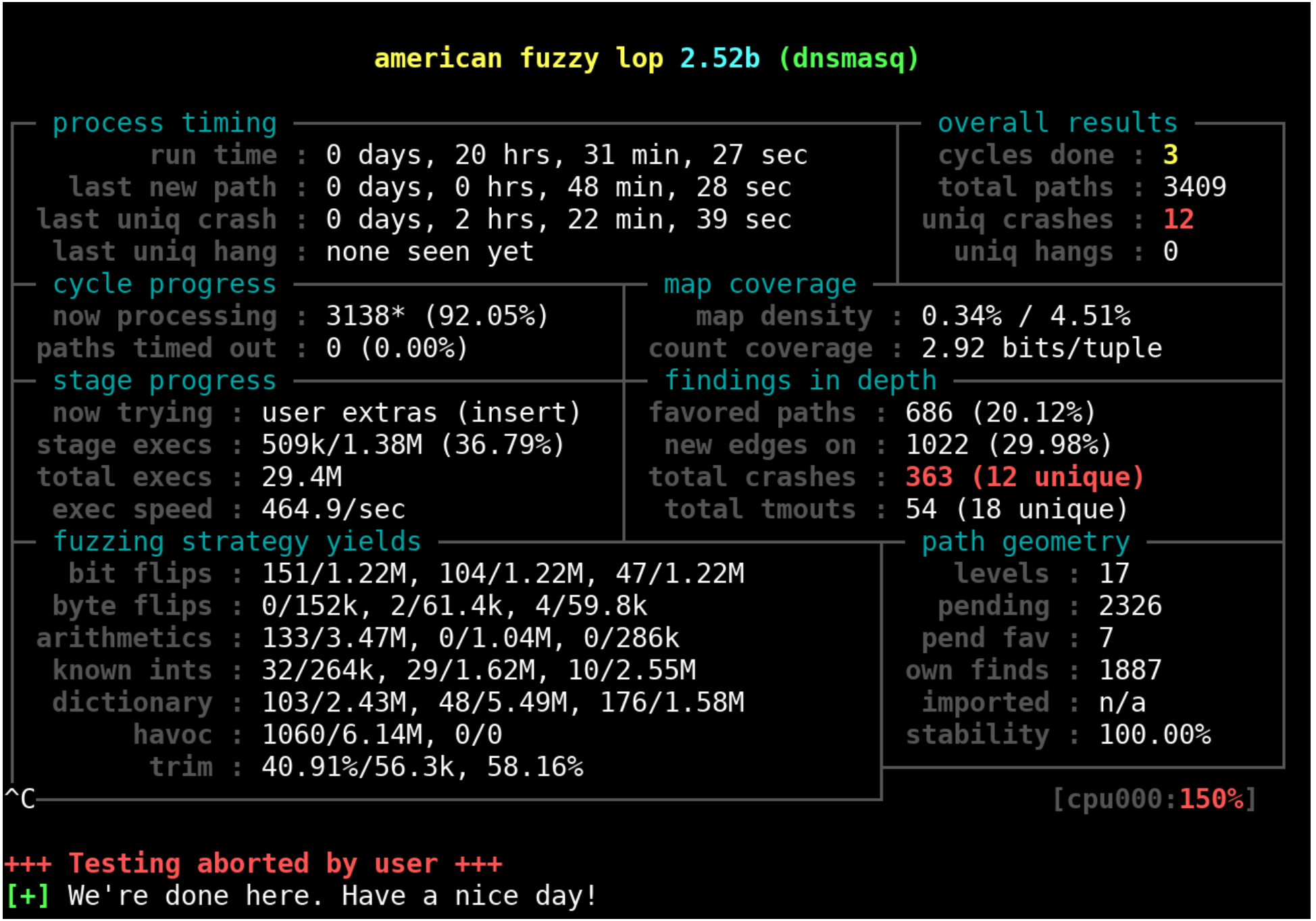
afl bitmap `shared_mem`

	1	2	3	4	5	6	7	8	9	10	11	12
1						☑[1]						
2												
3												
4												
5							☑[3]					
6												
7												
8												
9												
10		☑[2]										
11												
12												
13												
14												
15												

afl [http://lcamtuf.coredump.cx/afl]

- Code instrumented to observe execution paths:
  - if source code is available, by using modified compiler
  - if source code is not available, by running code in an emulator
- Fuzzing guided by code coverage
  or -- more accurately: branch coverage
  - 64KB bitmap to record jumps: each control flow jump is mapped to a change in this bitmap
  - different executions could result in same bitmap, but chance is small
- Big win: no need to specify the input format,
  but evolution will lead to good code coverage
- afl is *fast!*

Moral of the story: it can be better to be fast & simple (evolution in afl) than trying to be very clever (symbolic execution in SAGE)


american fuzzy lop 2.52b (dnsmasq)
process timing
run time : 0 days, 20 hrs, 31 min, 27 sec
last new path : 0 days, 0 hrs, 48 min, 28 sec
last uniq crash : 0 days, 2 hrs, 22 min, 39 sec
last uniq hang : none seen yet
overall results
cycles done : 3
total paths : 3409
uniq crashes : 12
uniq hangs : 0
cycle progress
now processing : 3138* (92.05%)
paths timed out : 0 (0.00%)
map coverage
map density : 0.34% / 4.51%
count coverage : 2.92 bits/tuple
stage progress
now trying : user extras (insert)
stage execs : 509k/1.38M (36.79%)
total execs : 29.4M
exec speed : 464.9/sec
findings in depth
favored paths : 686 (20.12%)
new edges on : 1022 (29.98%)
total crashes : 363 (12 unique)
total tmouts : 54 (18 unique)
fuzzing strategy yields
bit flips : 151/1.22M, 104/1.22M, 47/1.22M
byte flips : 0/152k, 2/61.4k, 4/59.8k
arithmetics : 133/3.47M, 0/1.04M, 0/286k
known ints : 32/264k, 29/1.62M, 10/2.55M
dictionary : 103/2.43M, 48/5.49M, 176/1.58M
havoc : 1060/6.14M, 0/0
trim : 40.91%/56.3k, 58.16%
path geometry
levels : 17
pending : 2326
pend fav : 7
own finds : 1887
imported : n/a
stability : 100.00%
^C
[cpu000:150%]
+++ Testing aborted by user +++
[+] We're done here. Have a nice day!

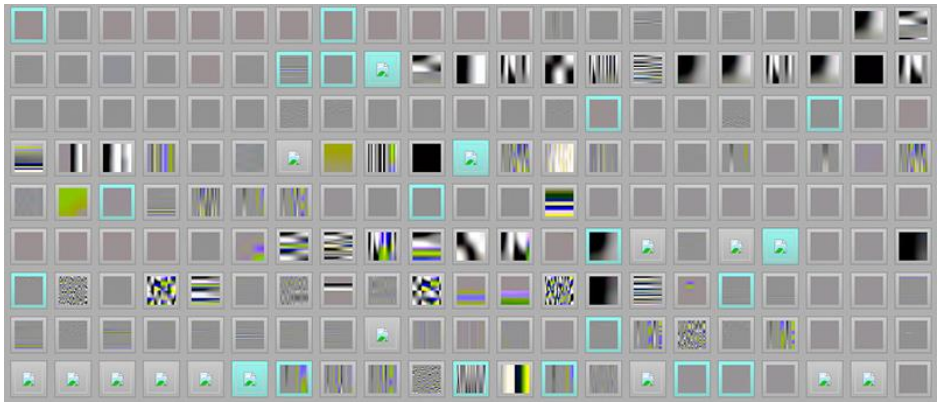
Cool example: learning the JPG file format

Fuzzing a JPG library, starting with `hello world` as initial test input, afl automatically learns the JPG file format

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]

afl trophy list

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]

Quick security assessment of C/C++ code

crashes with a dumb fuzzer

crashes with afl

crashes with afl & ASan

does not crash with any fuzzer

OSS Fuzz

Google's OSS-Fuzz initiative discovered over 36,000 bugs in over 1000 open source projects since 2016

- **Using the evolutionary fuzzers afl++, Honggfuzz and Libfuzzer**

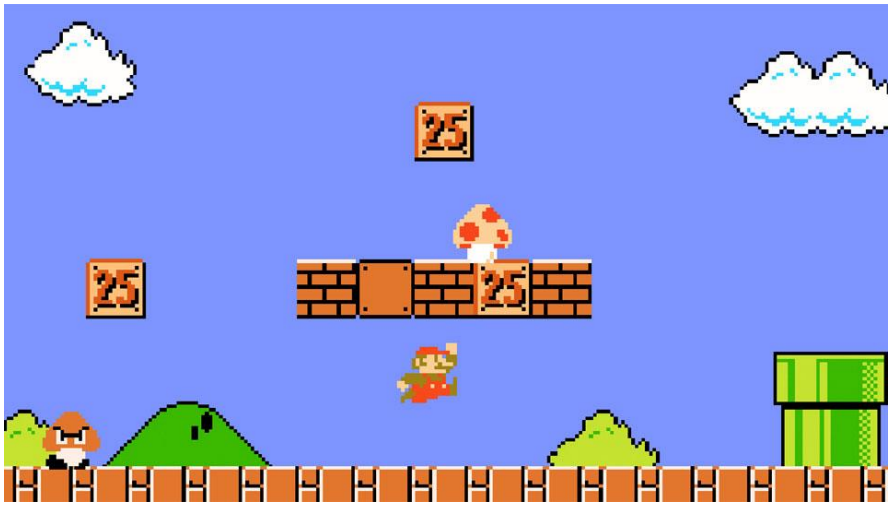
Kostya Serebryany, OSS-Fuzz - Google's continuous fuzzing service for open source software, invited talk at USENIX 2017

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

Other strategies in evolutionary fuzzing

Evolution can be guided by different properties than code coverage.

Eg maximum value of some variable, eg x-coordinate of Super Mario



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

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

Why is fuzzing so successful?

Root cause: COMPLEXITY of input formats

Fuzzing finds flaws in INPUT handling

The more complex the input format, the more bugs there are to find

Example complex input formats:

- any graphics, audio or video format: Flash, JPEG, MPEG, mp3, ...
- PDF, Word, SMS, animated cursors in Windows, HTML, URLs, ...

Searching the CVE list gives an indication of complexity

https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=PDF

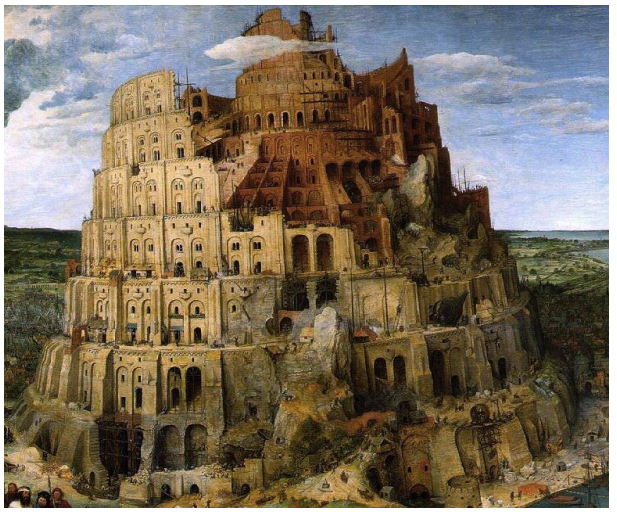
as do stories in news

Why is input handling often so insecure?

Not just COMPLEX input formats/languages, but also

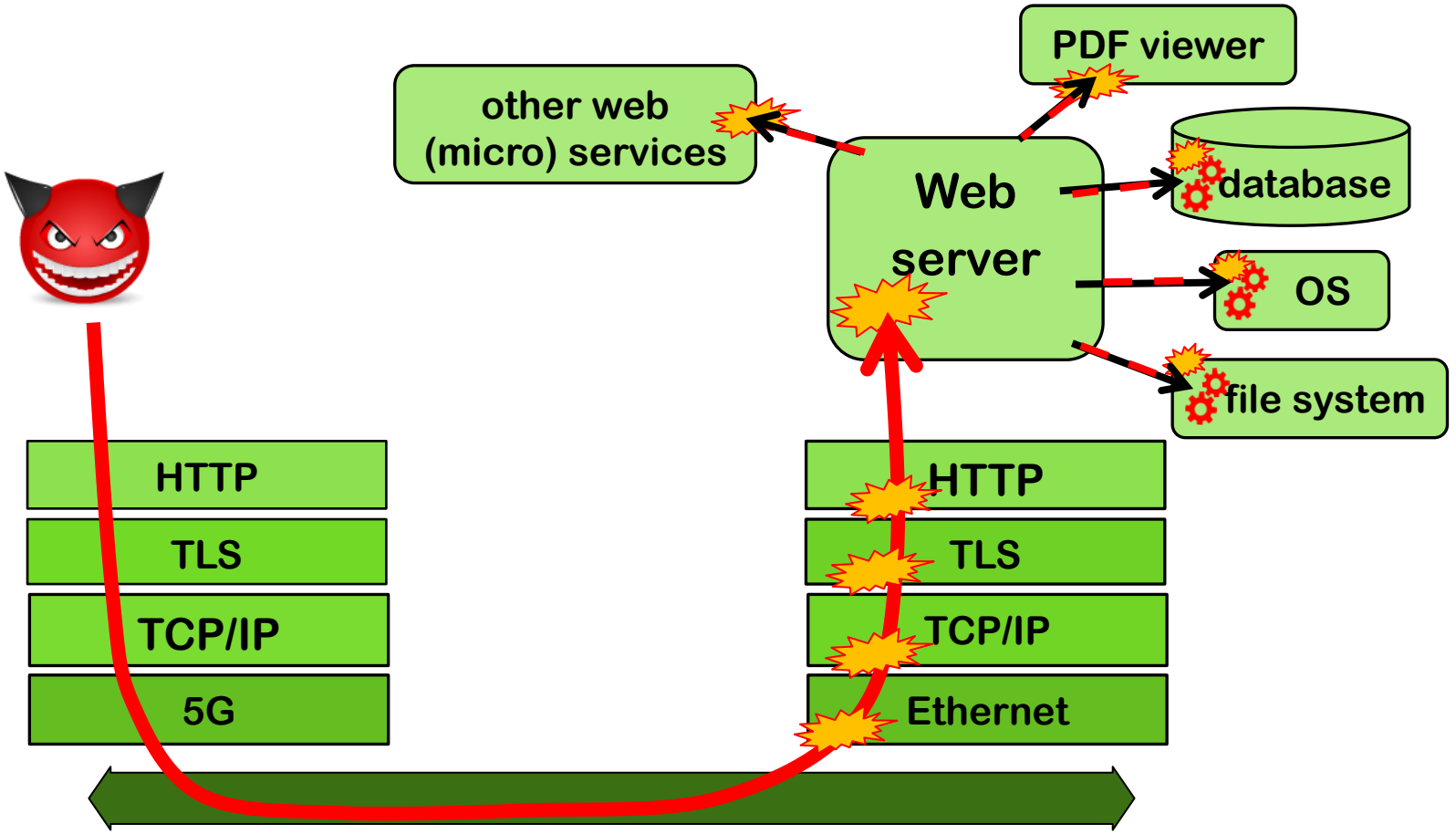
- MANY input formats
- POORLY SPECIFIED input formats
- EXPRESSIVE input formats

  eg. database commands in SQL
  JavaScript in HTML
  macros in Office formats
  JavaScript & ActionScript in PDF



Complexity & poor specs lead to memory corruption bugs in parsers,
expressivity leads to injection attacks

The many input languages of a typical application

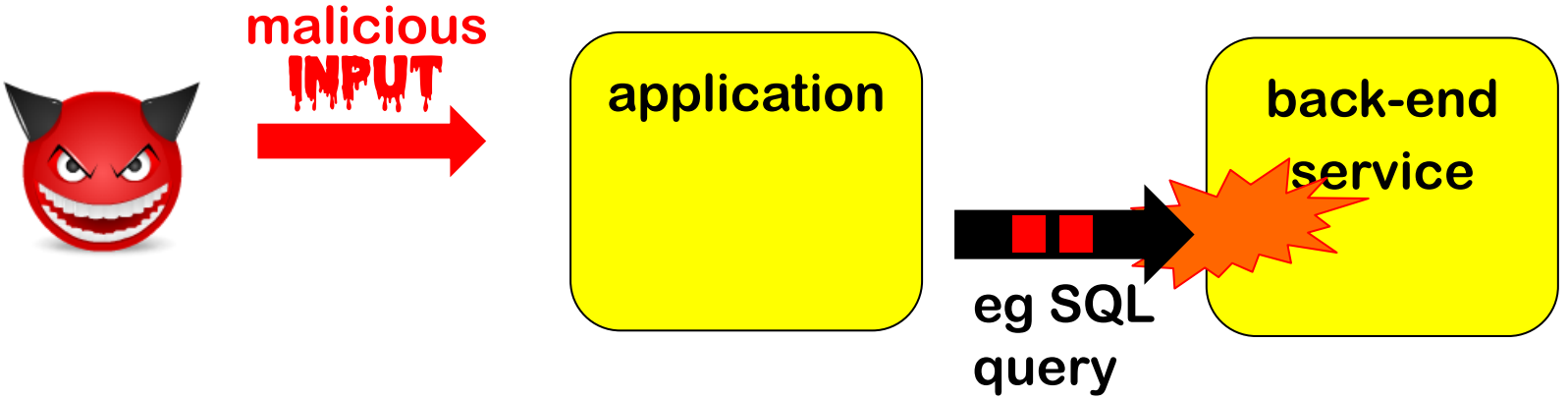


Attack surface in the underlying tech stack and in libraries & 'services'

Two main types of input problems

1. Buggy, insecure parsing

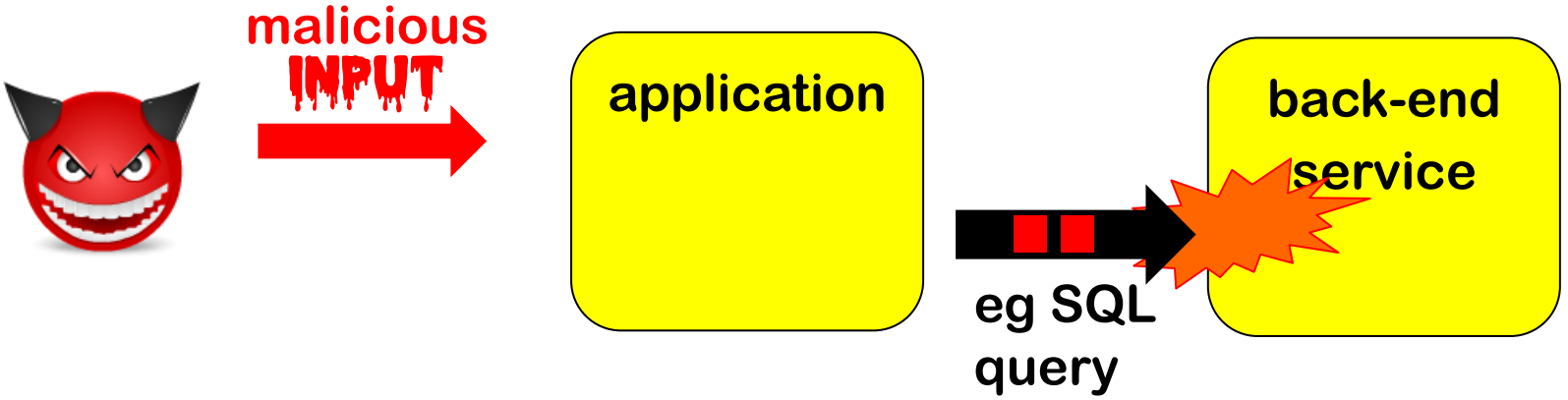
2. Injection attacks



Two main types of input problems

1. Buggy, insecure parsing

2. *Correct, but unintended* parsing



LangSec (Language-Theoretic Security)

LangSec approach highlighted the role of input languages on security problems, identifying root causes & countermeasures:

1. ***precisely defined* input languages**
   ideally with regular expression or context-free grammar (eg EBNF)
2. ***generated* parser code**
3. ***complete parsing before processing***
4. ***keeping the input language simple & clear***
   to make bugs are less likely
   to give minimal processing power to attackers

'The science of insecurity'

Sergey Bratus &
Meredith Patterson
presenting LangSec at CCC 2012

Example COMPLEXITY & EXPRESSIVITY : Windows file names

What can a Windows file name (incl. path) look like?

- classic MS-DOS notation C:\MyData\file.txt
- file URLs file:///C|/MyData/file.txt
- UNC (Uniform Naming Convention) \\192.1.1.1\MyData\file.txt

which can be combined in fun ways, eg file://///192.1.1.1/MyData/file.txt

This makes input validation of file names tricky.

To make matters worse: some formats can trigger unexpected behaviour

- UNC paths to remote servers are handled by SMB protocol
- SMB sends password hashes to authenticate: pass the hash

This can be exploited by SMB relay attacks

- CVE-2000-0834 in Windows telnet
- CVE-2008-4037 in Windows XP/Server/Vista
- CVE-2016-5166 in Chromium
- CVE-2017-3085 & CVE-2016-4271 in Adobe Flash
- ZDI-16-395 in Foxit PDF viewer

Example: X.509 certificates

This X.509 format is COMPLEX and was POORLY SPECIFIED

- The Common Name in a certificate, eg paypal.com,
  can be a comma-separated list, eg paypal.com, mafia.com

  Parsers in browsers and used by Certificate Authorities parsed this differently: some took all names, some the first, some the last …

- The Common Name is a string in ANS.1 format, which uses a length field instead of a string terminator \0

  Some X.509 parsers read www.paypal.com\0.mafia.com
  as www.paypal.com

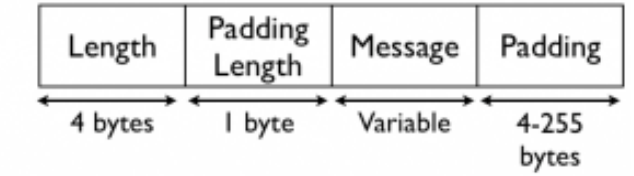
Using differential fuzzing we can try to find such differences

These bugs were used to spoof certificates in `SSL-strip` (by Moxie Marlinspike)
[Dan Kaminsky, Meredith L. Patterson and Len Sassaman, KI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure, FC'2010]

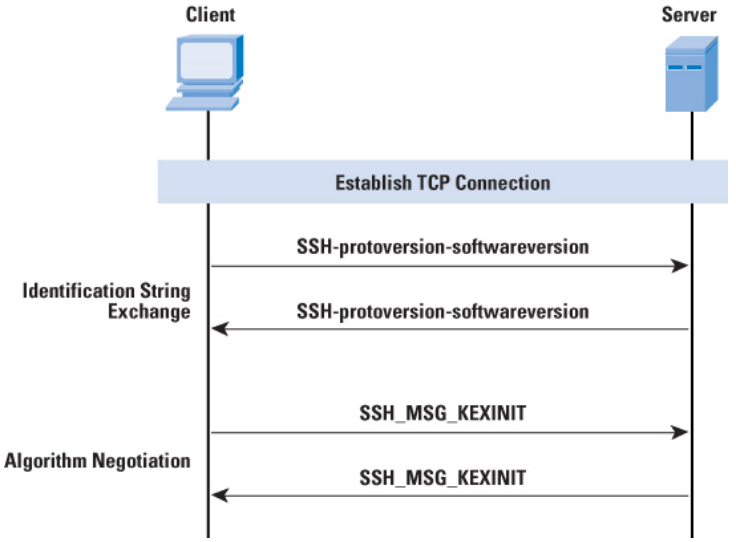
Protocol state fuzzing
aka
state machine inference

Protocol sessions

Many protocols not only involve input messages



but also *sequence* of input messages



Normal interactions follow the happy flow

For security, getting unhappy flows correct can be crucial!

A secure implemention implements a protocol state machine to detect and abort unhappy flows

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.

If client sends SSH2_MSG_USERAUTH_*SUCCESS* message to the server, in place of the SSH2_MSG_USERAUTH_*REQUEST* message which the server would expect to initiate authentication, the attacker successfully authenticates without credentials.

https://www.libssh.org/security/advisories/CVE-2018-10933.txt

Finding such bugs?

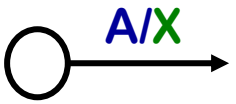
We can find such bugs by active learning aka state machine inference

Basic idea: we try random sequences of commands to see if something weird happens

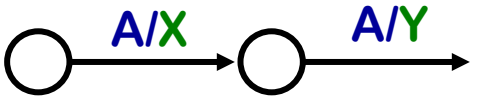
State machine inference

Just try out many sequences of inputs, and observe outputs

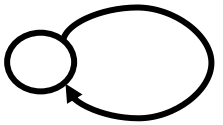
Suppose input A results in output X



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

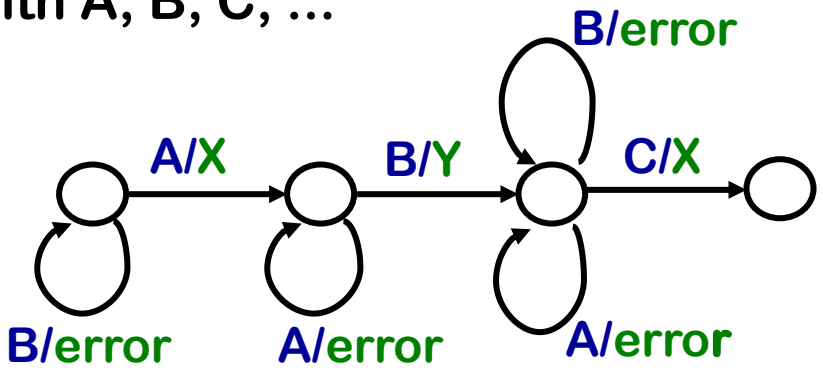


- If second input A results in the *same* output X



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

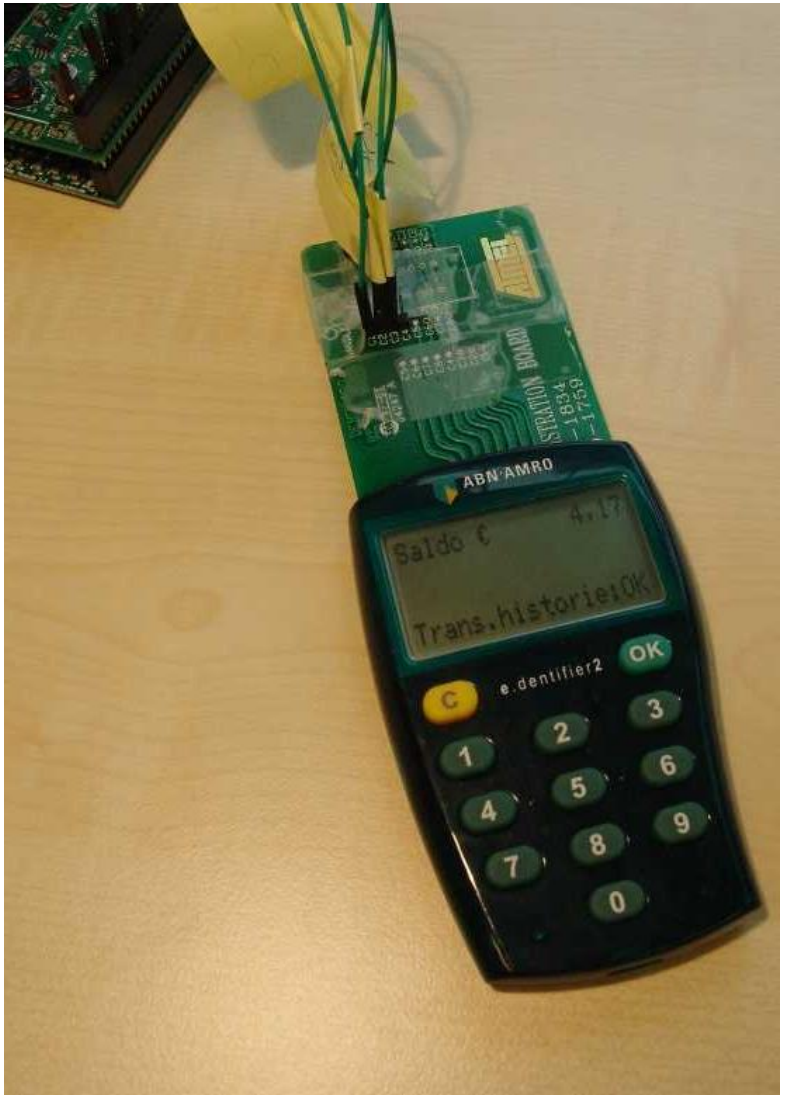
Case study: the USB-connected e.dentifier2

Can we use state machine learning with

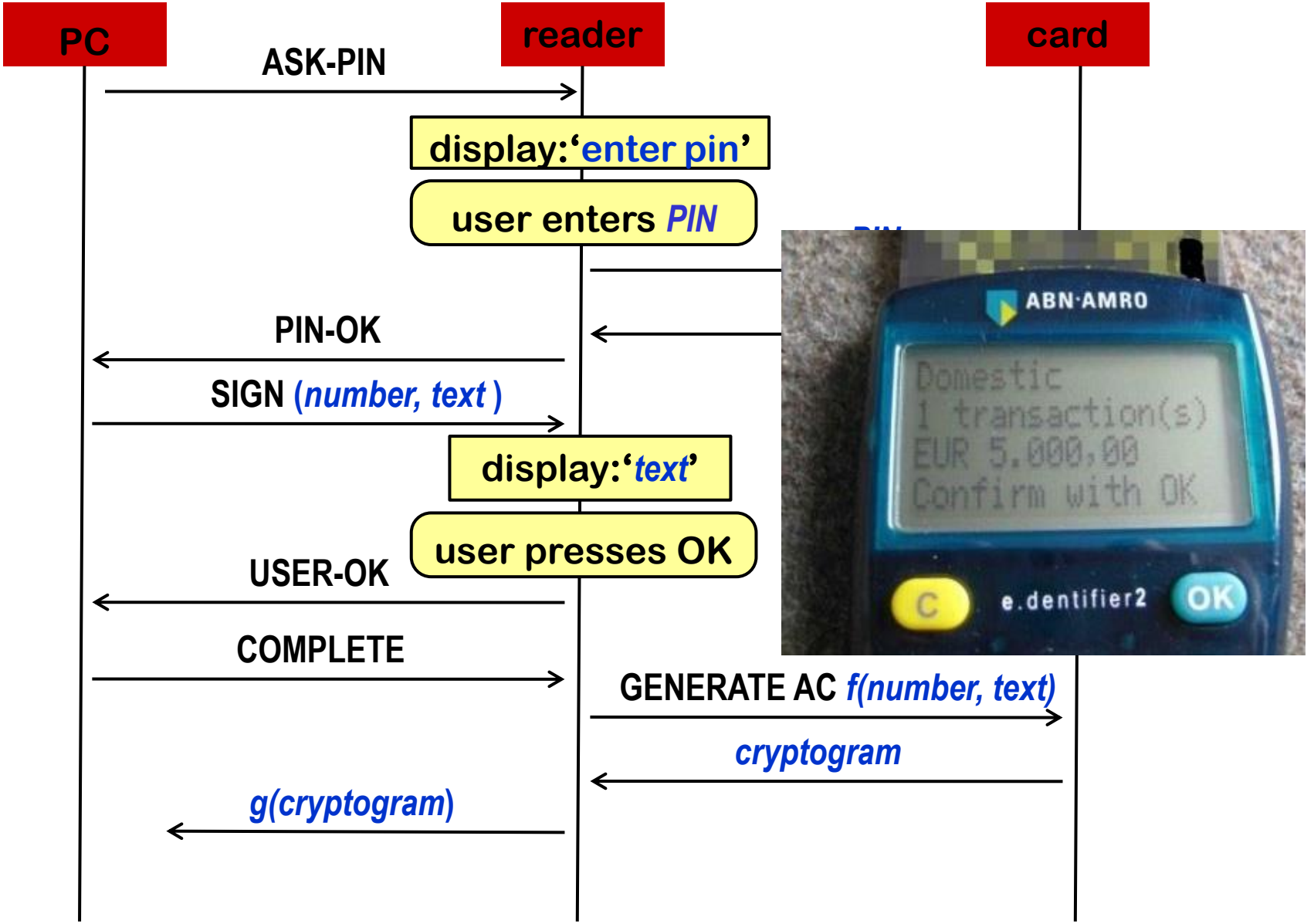
- USB commands
- user actions via keyboard

to obtain the state machine

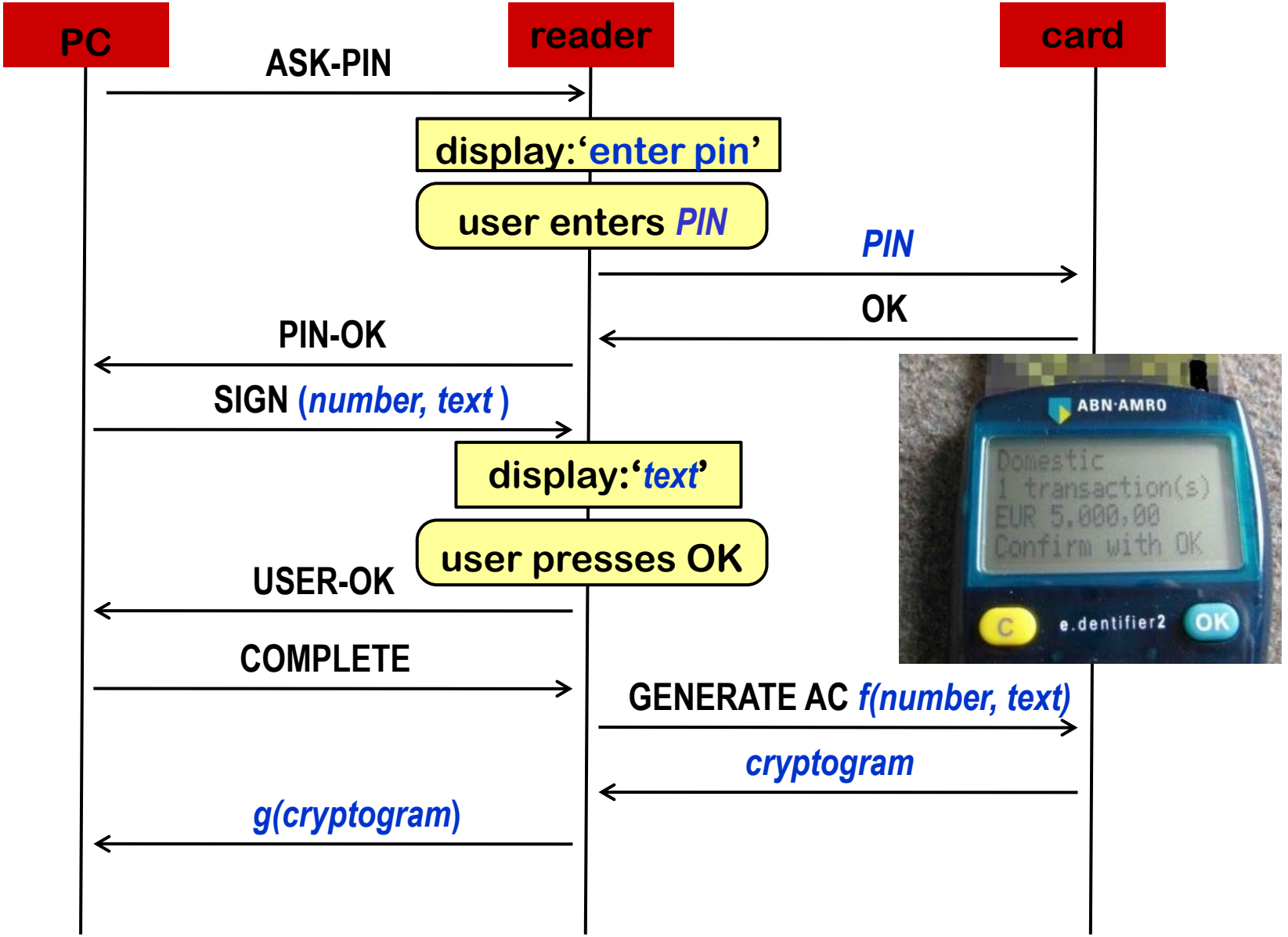
internet banking device?



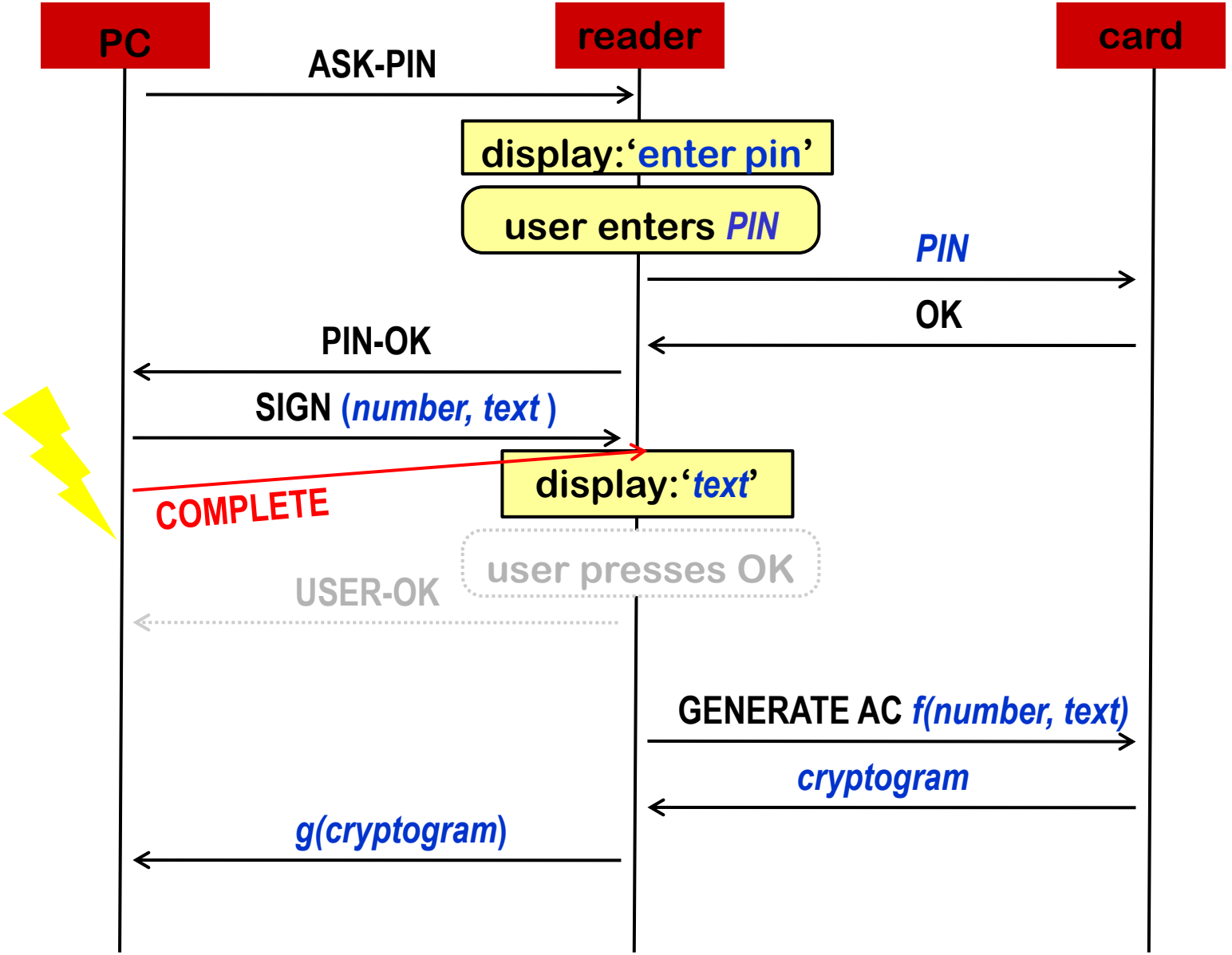
(Manually) reverse-engineered protocol



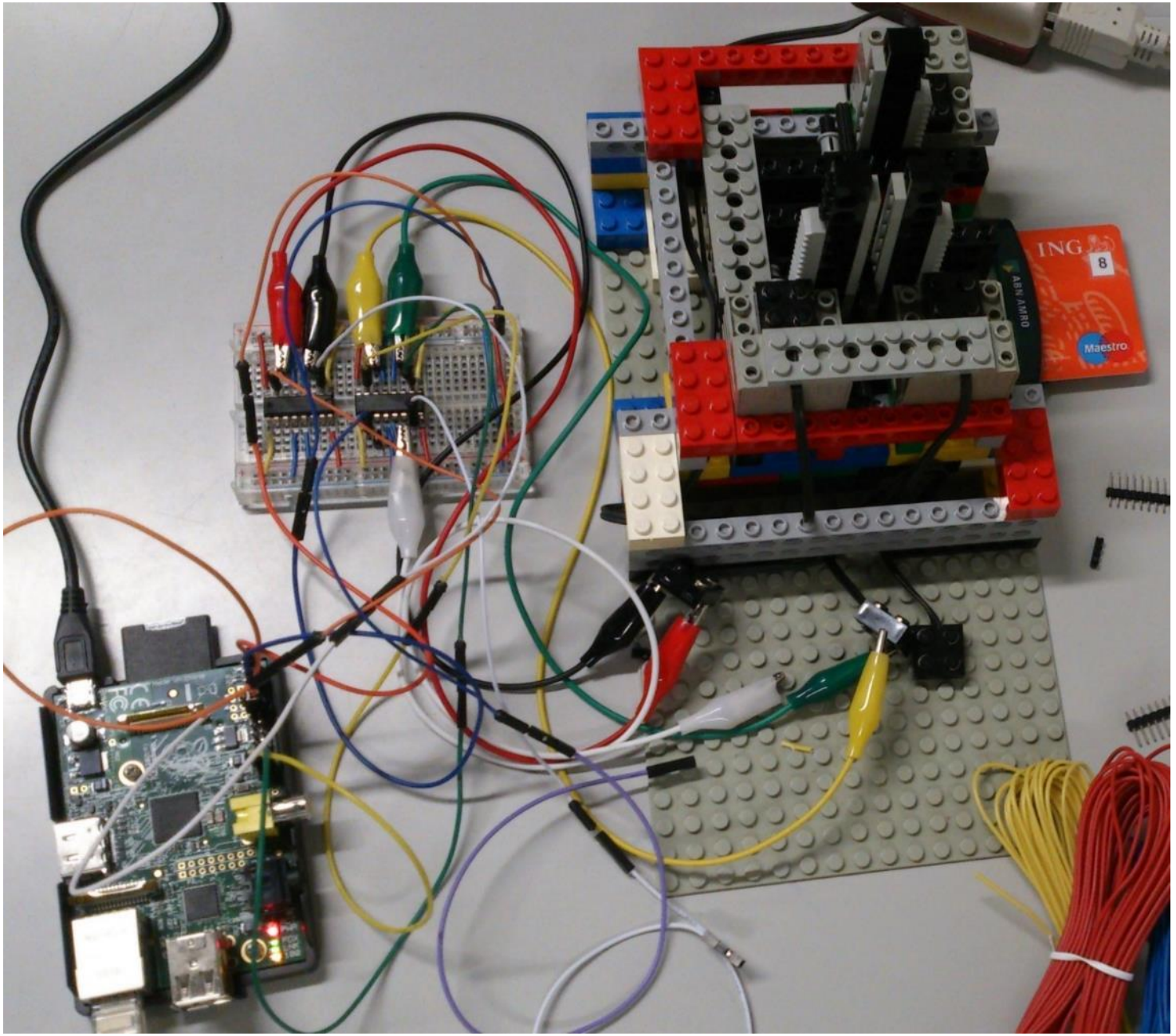
Spot the defect!

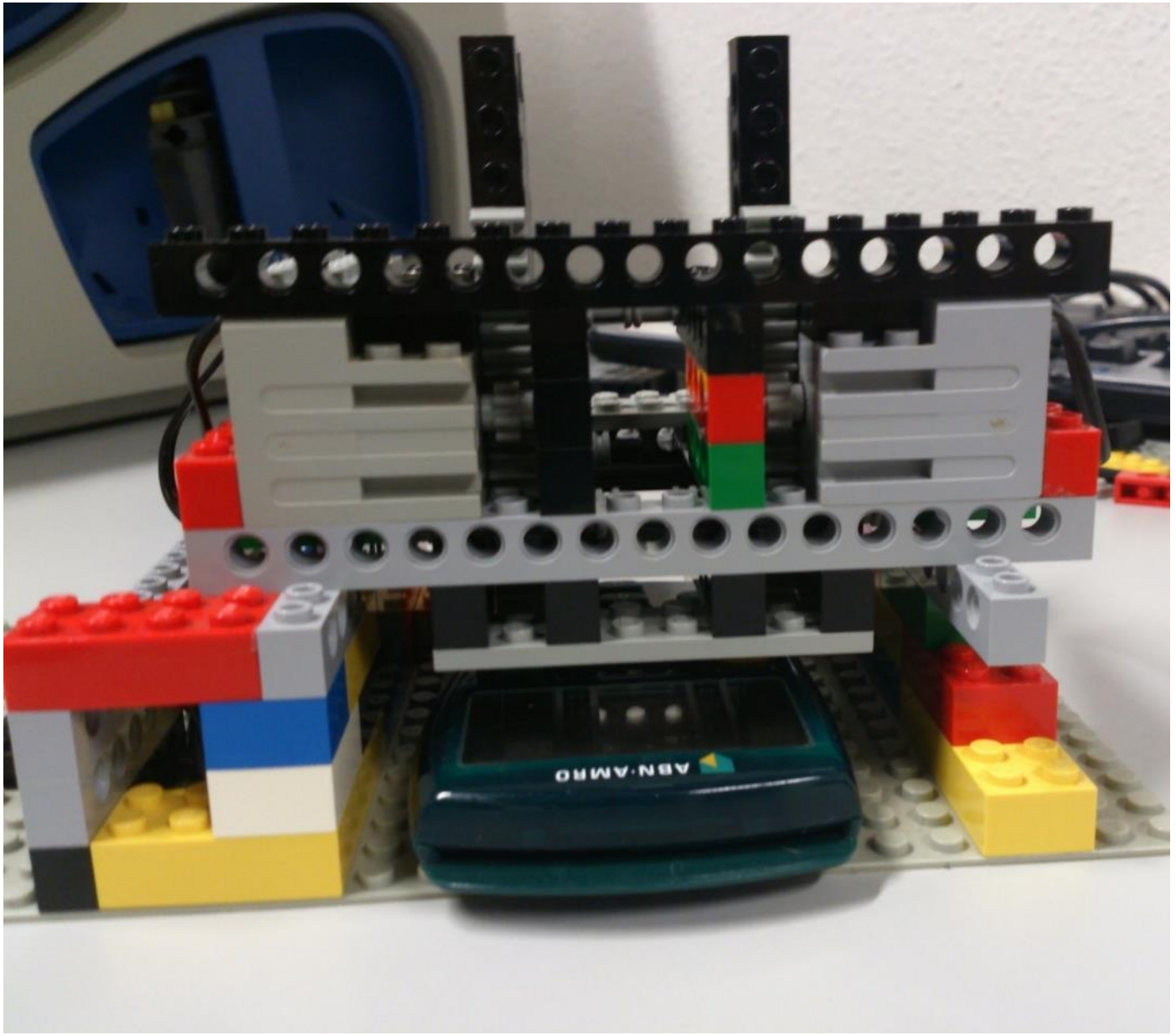


Attack!

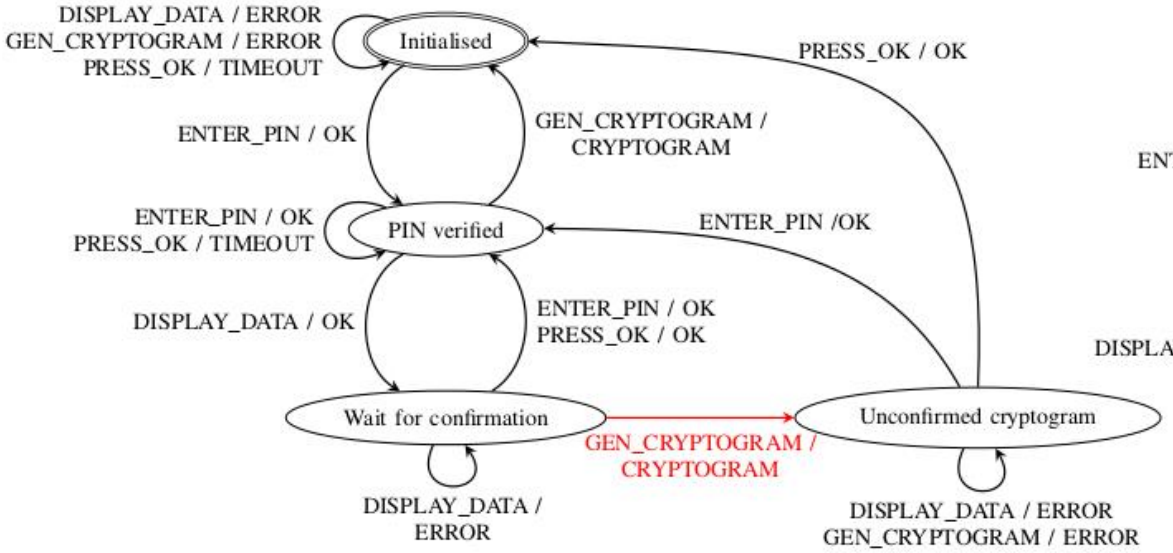

PC
reader
card
ASK-PIN
display:'enter pin'
user enters PIN
PIN
OK
PIN-OK
SIGN (number, text)
COMPLETE
display:'text'
user presses OK
USER-OK
GENERATE AC f(number, text)
cryptogram
g(cryptogram)

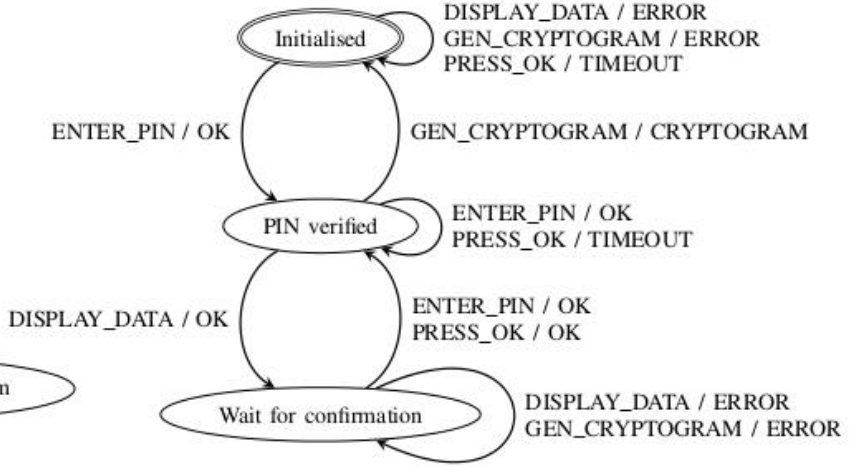
Operating the keyboard using LEGO

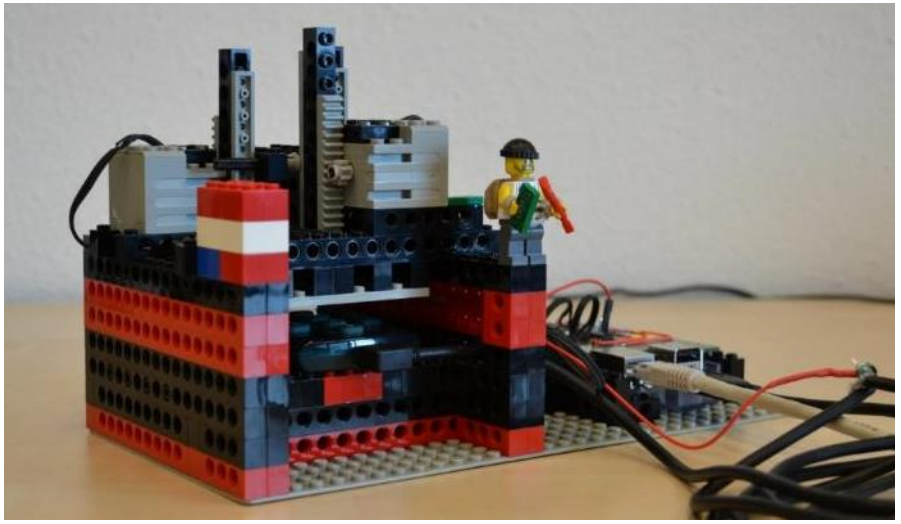

ING
ABN AMRO
Maestro


ABN·AMRO

State machines of old vs new e.dentifier2

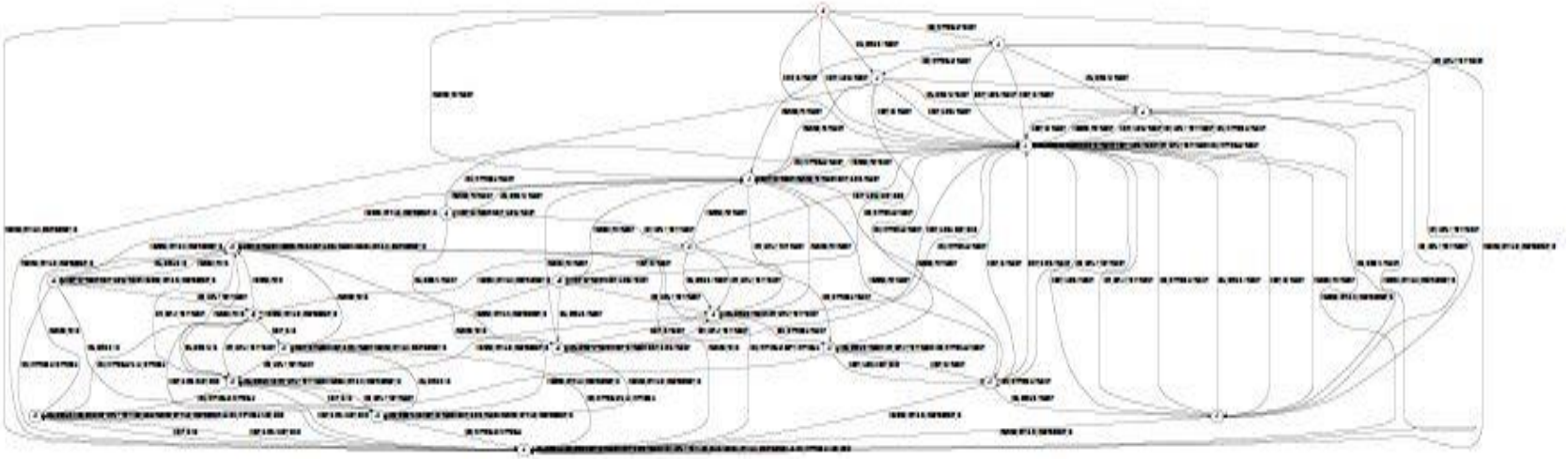






https://www.youtube.com/watch?v=hyQubPvAyq4

Would you trust this to be secure?



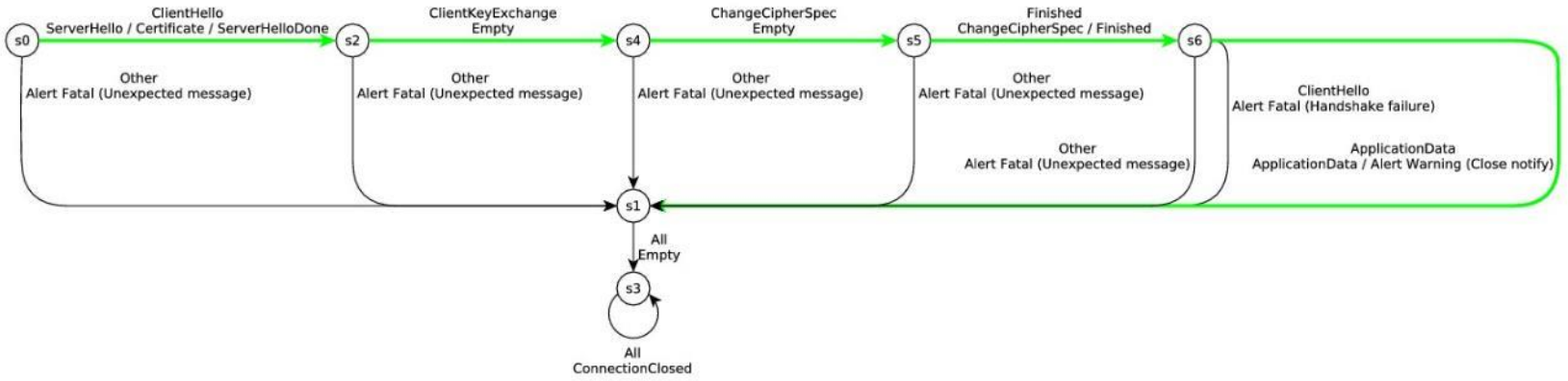
More detailed inferred state machine, using richer input alphabet.

Do you think whoever designed or implemented this is confident that this is secure?

Or that all this behaviour is necessary?

[Georg Chalupar et al., Automated Reverse Engineering using Lego, WOOT 2014]

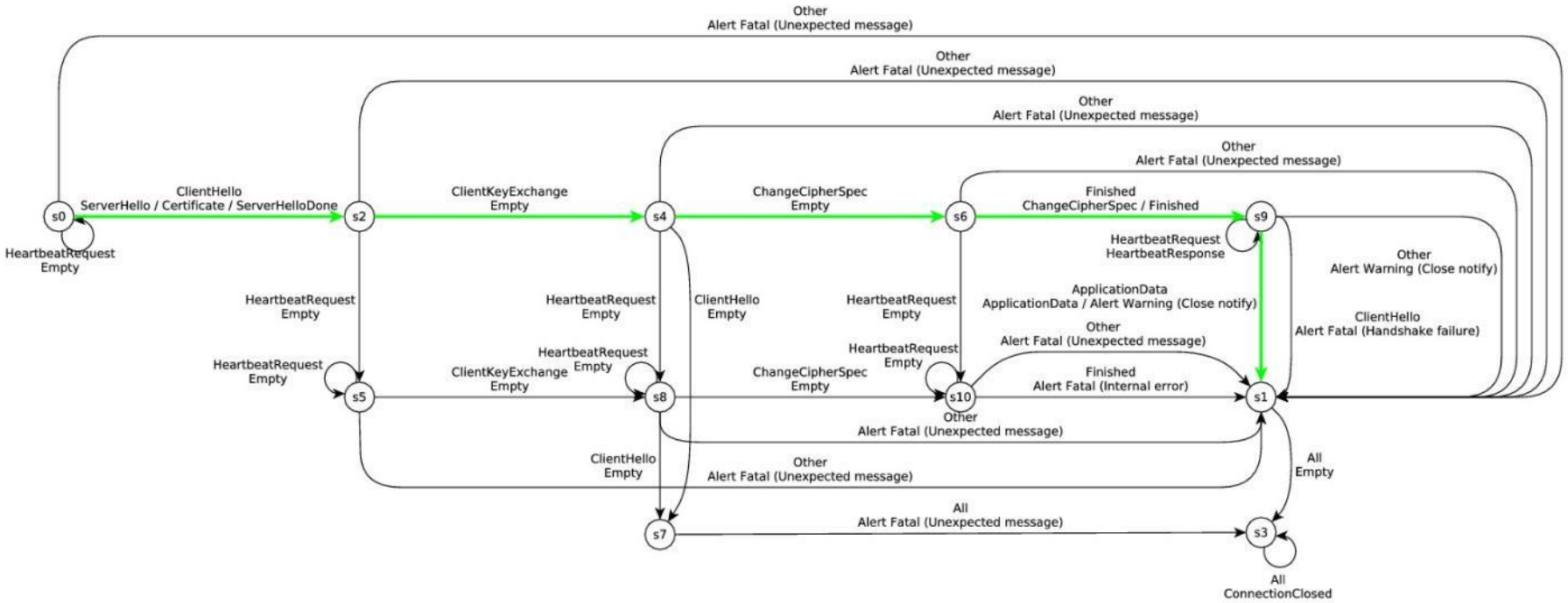
Case study: TLS



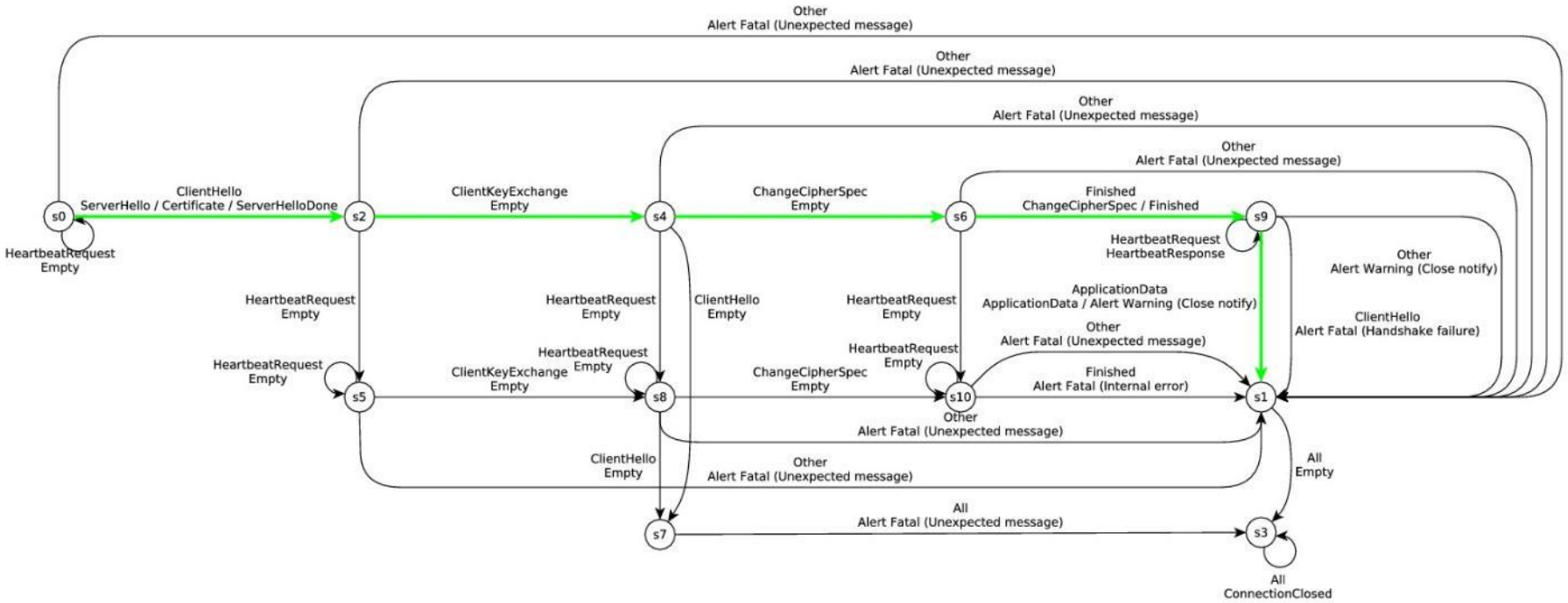
State machine inferred from NSS implementation

Comforting to see this is so simple!

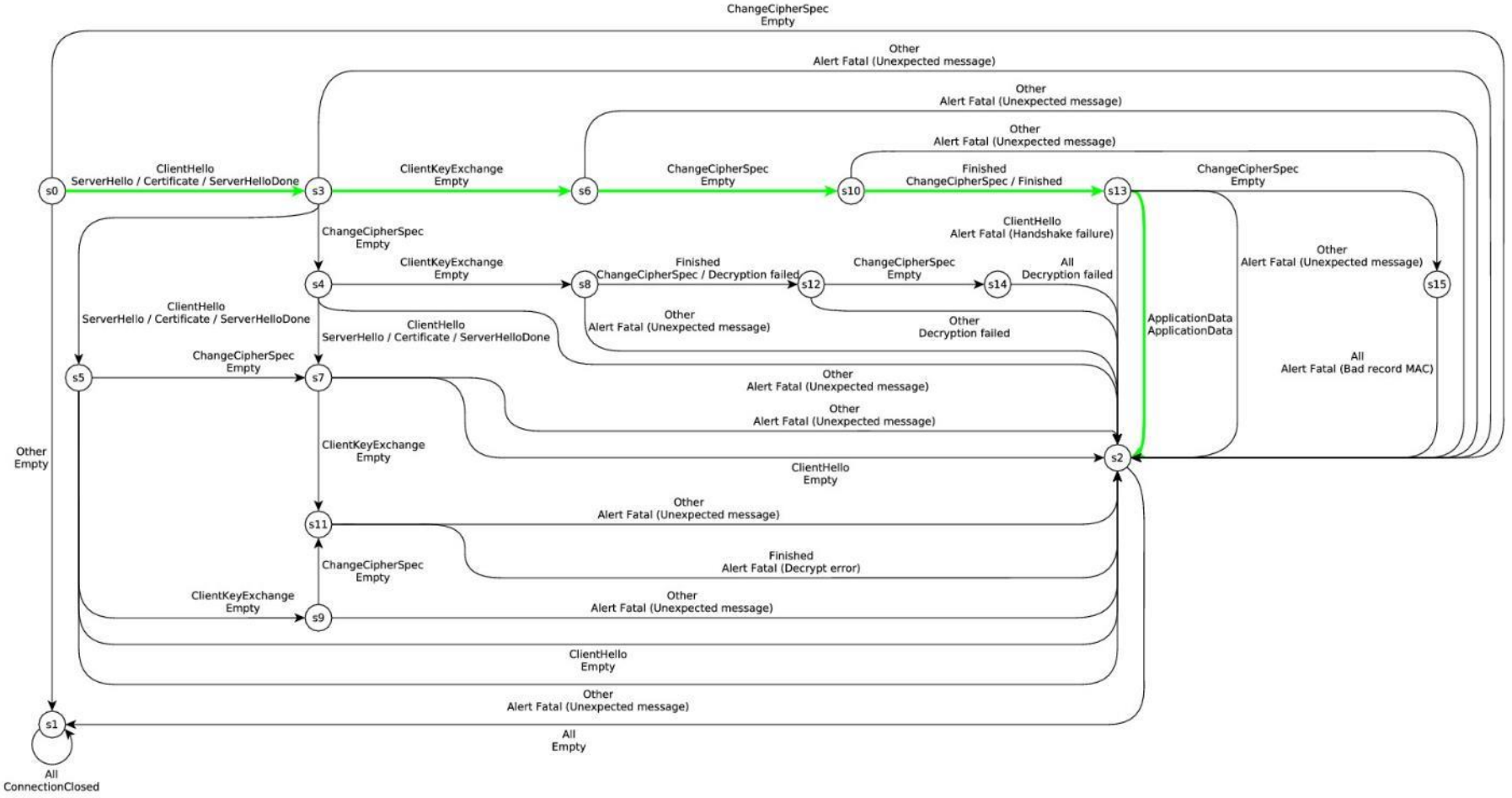
TLS... according to GnuTLS



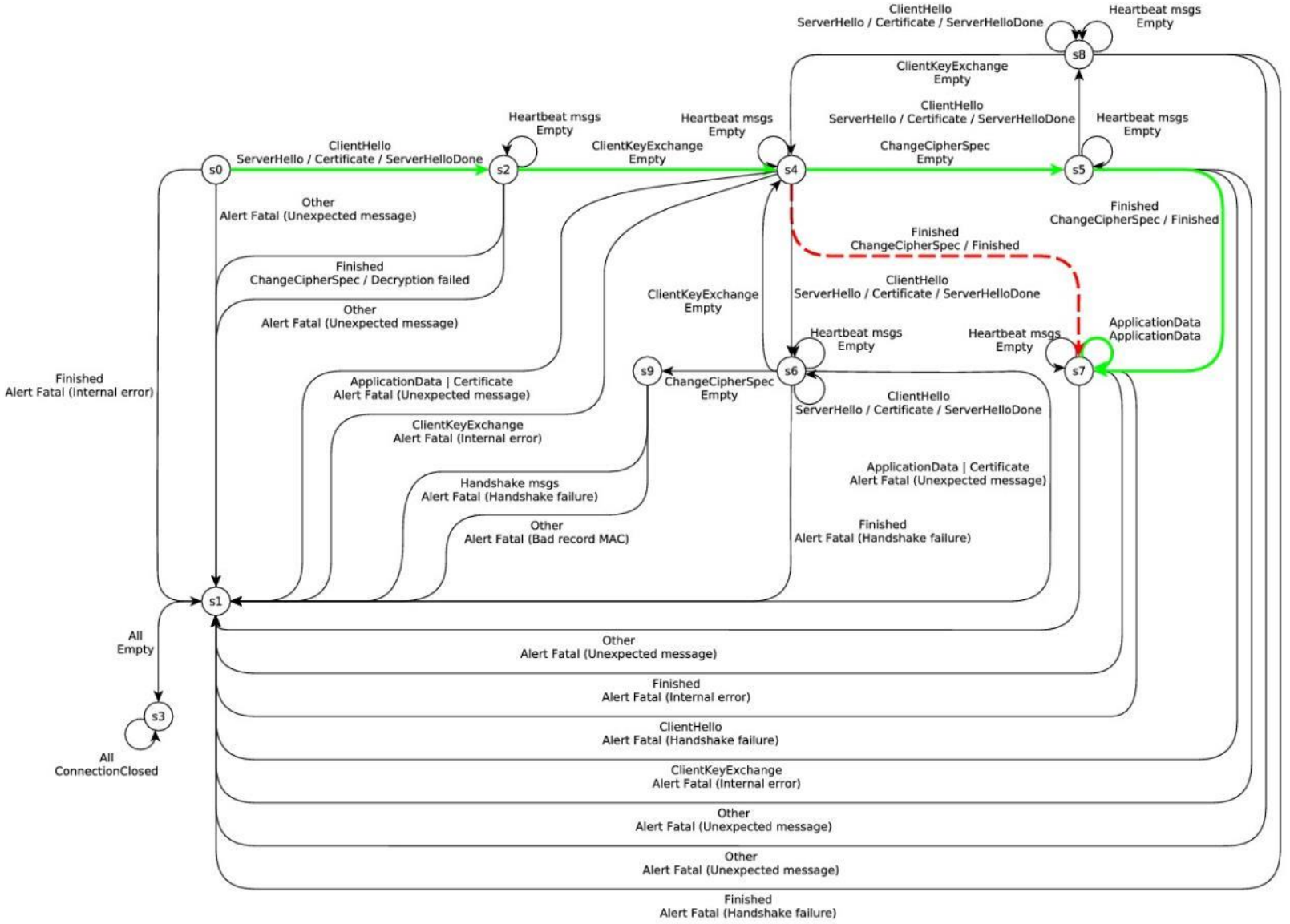
TLS... according to GnuTLS



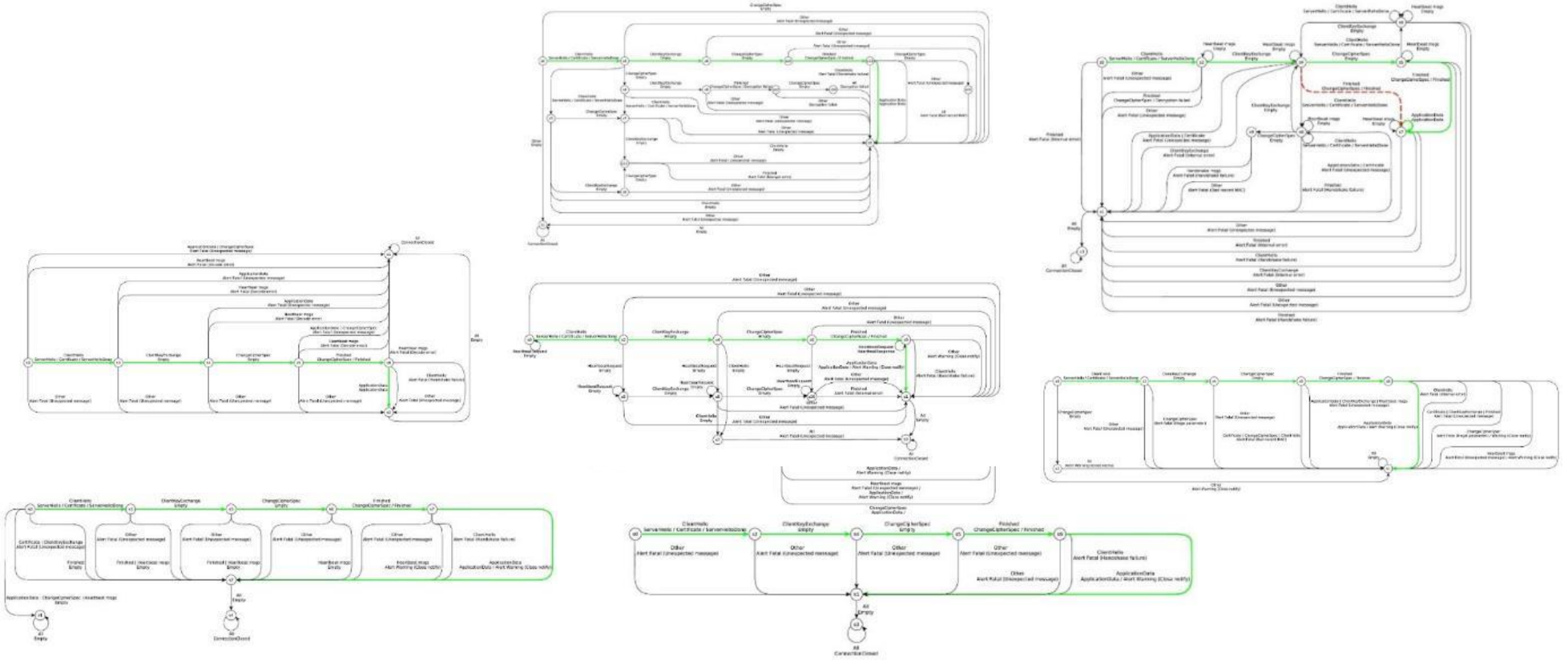
TLS... according to OpenSSL



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]

Wrap-up

Conclusions

- Fuzzing is a great technique to find security flaws
- State machine inference – fuzzing sequences of inputs – is a great technique to find logic flaws in protocols

- Input handling problems are caused by *many*, *complex*, *poorly specified*, *expressive* input languages

- Input handling flaws involve PARSING
  - *insecure* parsing (eg. buffer overflow in PDF viewer)
  - *unintended* parsing (eg. injection attacks: SQLi, XXS, ...)
  - *incorrect* parsing (resulting in parser differentials)

Give a man an 0day and he'll have access for a day,
teach a man to phish and he'll have access for life

-- the grugq