Improving software security by improving input handling

Erik Poll

Digital Security

Radboud University Nijmegen

Overview

1. General observations about security
   - Why software is what matters, and esp. input handling in software

2. Preventing a large class of input security problems by construction
   - using LangSec approach, esp. parser generation [http://langsec.org]

3. Our own additions to the LangSec approach
   - protocol state machines [LangSec 2015]
   - also tackling forwarding flaws (aka injection flaws) [LangSec 2018]

Root cause of security problems: SOFTWARE

- Systems (laptops, servers, phones, cars, planes, industrial plants, …) can be hacked because there is software in them

- Software is *the* main root cause of security problems
  - The only other important cause of problems: the human factor

- Cyber security is a software engineering problem
  - Don’t count on security researchers, network security people, cryptographers, … to solve this

secure functionality $\neq$ security functionality

- **Some software implements security controls or functionality**
  - **e.g. security protocols, access control mechanisms, login procedures, …**
- **Obviously, such software needs to be correct & secure.
  We could try to specify & verify it.**
  - **e.g. NICTA's L4.verified microkernel, INRIA's miTLS**
- **However, *ALL* software needs to be secure, not just the security software**
  - **incl. device drivers, browsers, Microsoft Office, PDF viewers, mp3 players, Bluetooth interface, …**

'Achilles only had an Achilles heel, I have an entire Achilles body'

- Woody Allen

LangSec

(language-theoretic security)

LangSec (Language-Theoretic Security)

- Interesting look at root causes of large class of security problems, namely problems with input
- Useful suggestions for dos and don'ts

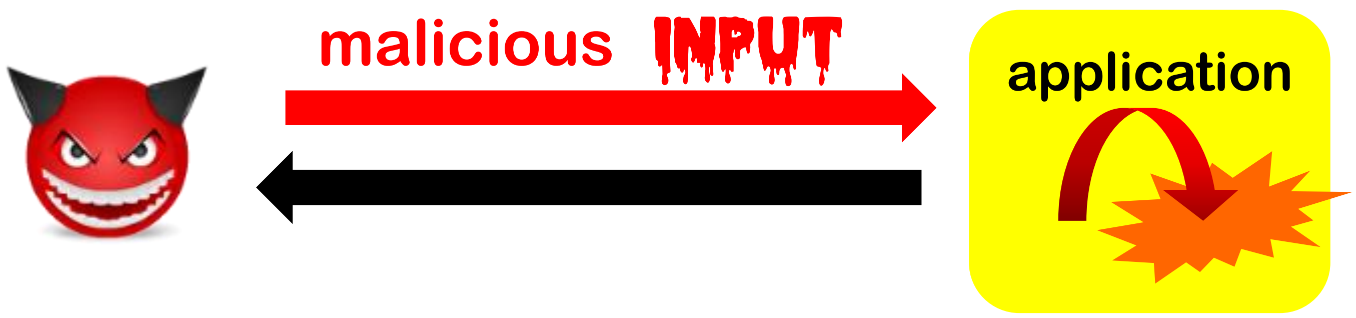
Sergey Bratus & Meredith Patterson
'The science of insecurity'
CCC 2012

- The 'Lang' in 'LangSec' refers to *input* languages,
  not *modelling* or *programming* languages.

Common theme in security flaws: INPUT

Mishandling malicious input is *the* common theme in many attacks

buffer overflows, integer overflows, command injection, path traversal, SQL injection, XSS, CSRF, Word macros, XML injection, deserialization attacks, …



- Garbage In, Garbage Out

    leads to

  Malicious Garbage In, *Security Incident* Out

Example INPUT problem: PDF

Security Update for Foxit PDF Reader Fixes 118 Vulnerabilities

By Lawrence Abrams October 2, 2018 02:49 AM

- Root cause: PDF spec is horrendously complex
- These Foxit bugs are mainly memory corruption flaws that allow remote code execution
    - so high impact, and easy to exploit with email attachments
- *All* PDF viewers suffer from such problems

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

Example INPUT problem: X.509 certificates

X.509 spec is horribly complex. Example attacks:

- **Multiple names, comma-separated, in a certificate Common Name**

  `paypal.com,mafia.org`

  Different browsers and CAs interpret this in different ways; such parser differentials can be critical security flaws.

- **ANS.1 attacks**

  Null terminator in ANS.1 BER-encoded string in a Common Name

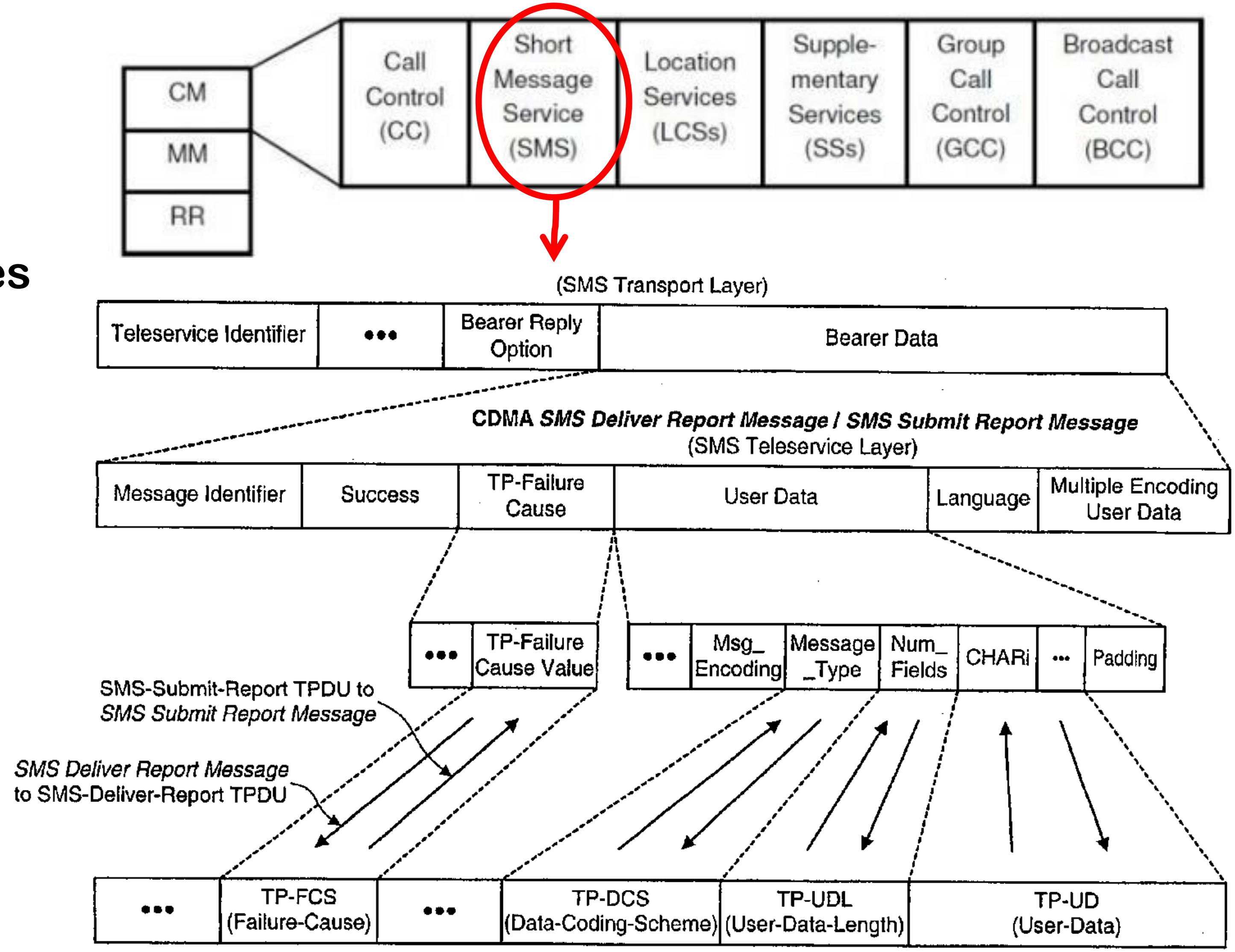
  `paypal.com\00mafia.org`

[Dan Kaminsky, Meredith Patterson, and Len Sassaman, *PKI Layer Cake: New Collision Attacks against the Global X.509 Infrastructure*, Financial Crypto 2010]

Hand-written parsers of complex languages *will* go wrong

Eg GSM specs

for SMS text messages



Unsurprisingly, malformed GSM traffic can trigger lots of problems

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

Even hand-written parsers of simple formats go wrong

```
char buf1[MAX_SIZE], buf2[MAX_SIZE];
// make sure url is valid and fits in buf1 and buf2:
if (!isValid(url)) return;
if (strlen(url) > MAX_SIZE - 1) return;
// Now copy url up to first '/' into buf1
out = buf1;
do {
  // skip spaces
   if (*url != ' ') *out++ = *url;
} while (*url++ != '/');
strcpy(buf2, buf1);
...
```

What if there is no / in the url?

This bug was exploited by the Blaster worm in 2003.

LangSec: root causes of security problems

- Input languages play a central role causing security flaws
  - aka protocols, file formats, encodings, …
- *Any* language *anywhere* in the protocol stack, incl.

  TCP/IP v4 or v6,

  WiFi, GSM/3G/4G, Ethernet, Bluetooth,

  OpenVPN, SSH,

  HTTP(S), TLS, X.509, HTML5 (incl. JavaScript), XML, JSON,

  URLs, email addresses, S/MIME,

  JPG, doc, PDF, xls, MP3, MPEG, Flash,

  …
- This provides a **huge** attack surface for the attacker

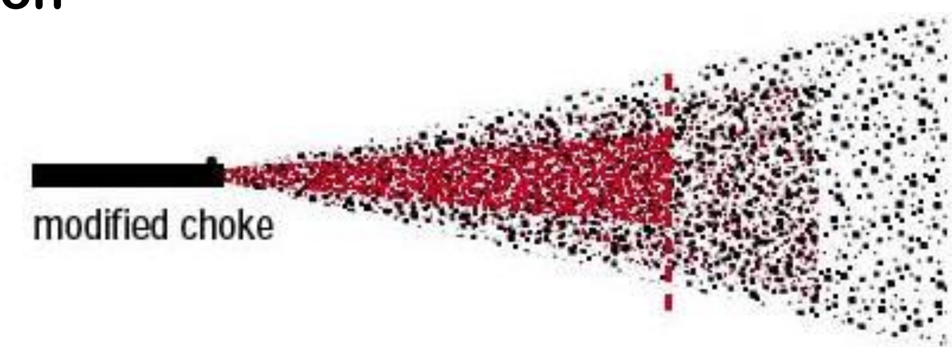
LangSec: root causes of security problems

- *Ad-hoc, imprecise,* or *complex* notions of input validity

  Eg, have you looked at how complex the Flash file format is? Or HTML5? Or X.509 certificates?

- *Handwritten* parsers, which *mix input recognition & processing*

  shotgun parser: code that incrementally parses & interprets input in a piece-meal fashion



The buggy parsing & processing then results in weird behaviour
- a weird machine - for attackers to have fun with

LangSec principles

1. ***Precisely defined*** **input languages**

   Ideally with regular expression or EBNF grammar.

   Common problem: length fields that make format context-sensitive

2. ***Generated*** **parser code**

3. ***Complete*** **parsing** ***before*** **processing**

   So also don't substitute strings & then parse,

   but parse & then substitute in parse tree

   (c.f. parameterised SQL queries instead of dynamic SQL)

4. ***Keep the input language simple & clear***

   So that equivalence of parsers is ideally decidable.

   So that you give minimal processing power to attackers.

LangSec in slogans

[Photoshopped by Kythera of Anevern, see http://langsec.org/occupy]


COMPUTATIONAL
EQUIVALENCE
FOR ALL
PROTOCOL
ENDPOINTS

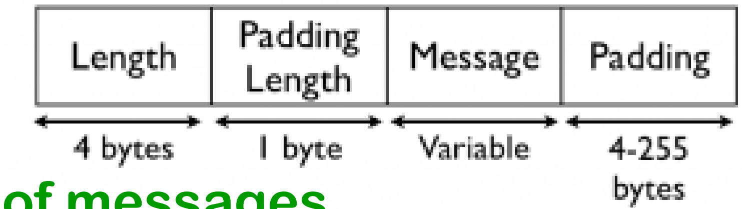

NO MORE
Turing-
Complete
Input
Languages

LangSec continued: protocol state machines

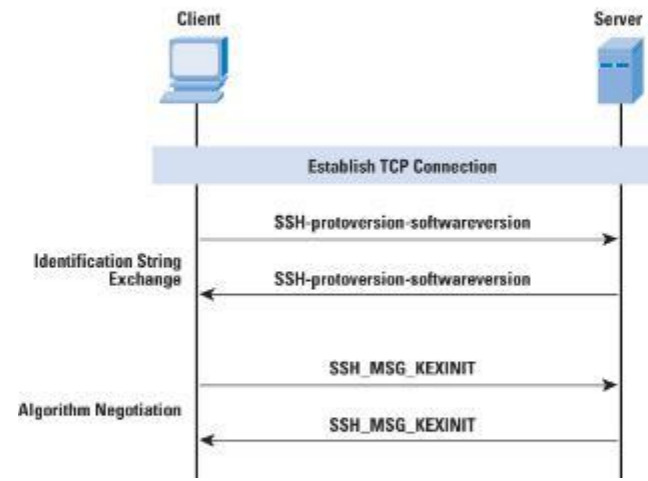
[LangSec 2015 paper]

Sequences of inputs

Many protocols not only involve a language of input messages



but also a notion of session, ie. *sequence* of messages

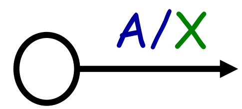


- **Most specs only describe the happy flow.**
  For security, getting unhappy flows correct can be crucial!
- **A specification of all flows could be given by a state machine…**
- **Fortunately, we can extract state machines from systems by black box testing!**

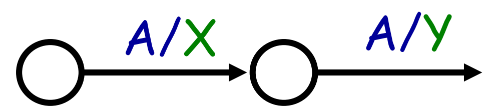
State machine inference, eg using LearnLib tool

Just try out many sequences of inputs, and observe outputs

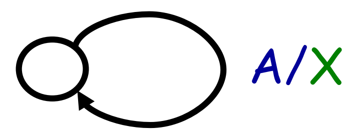
Suppose input A results in output X



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

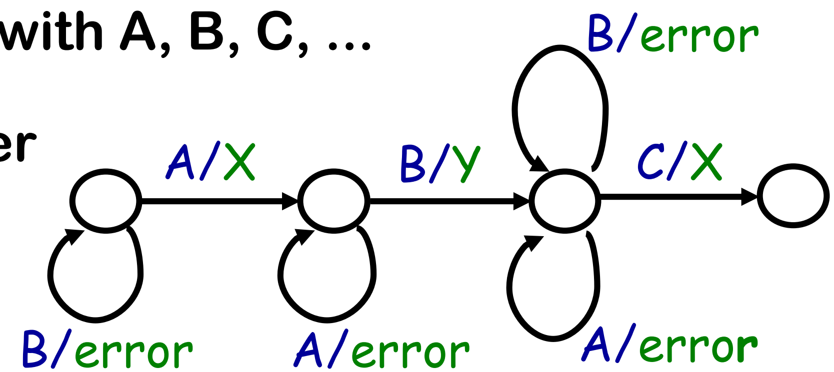


- If second input A results in the *same* output X



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

to e.g. infer

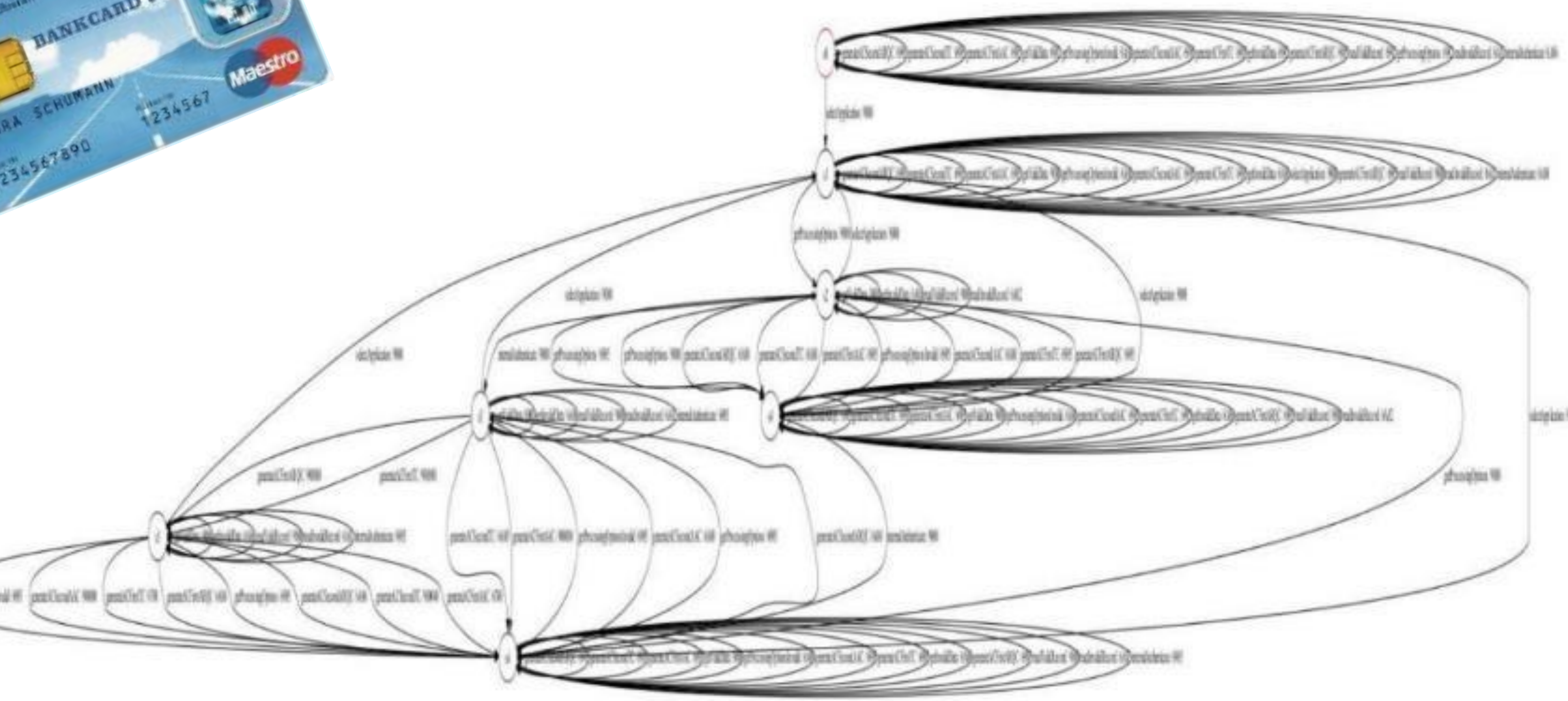


The inferred state machine is an under-approximation of real system

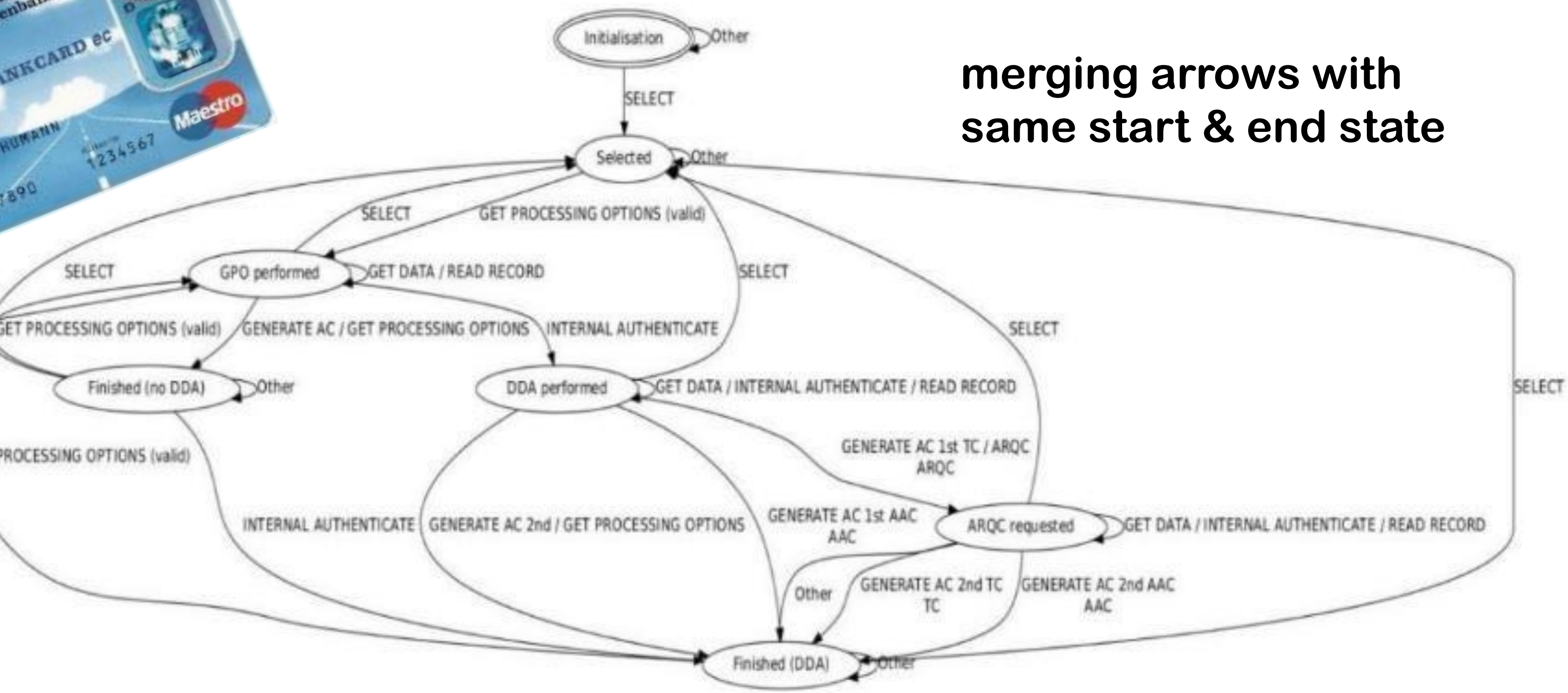
Case study: EMV

- **Most banking smartcards implement a variant of EMV**
  - **EMV = Europay-Mastercard-Visa**
- **Specification in 4 books totalling > 700 pages**
- **Contactless payments: another 7 books with > 2000 pages**

State machine inference of Maestro card



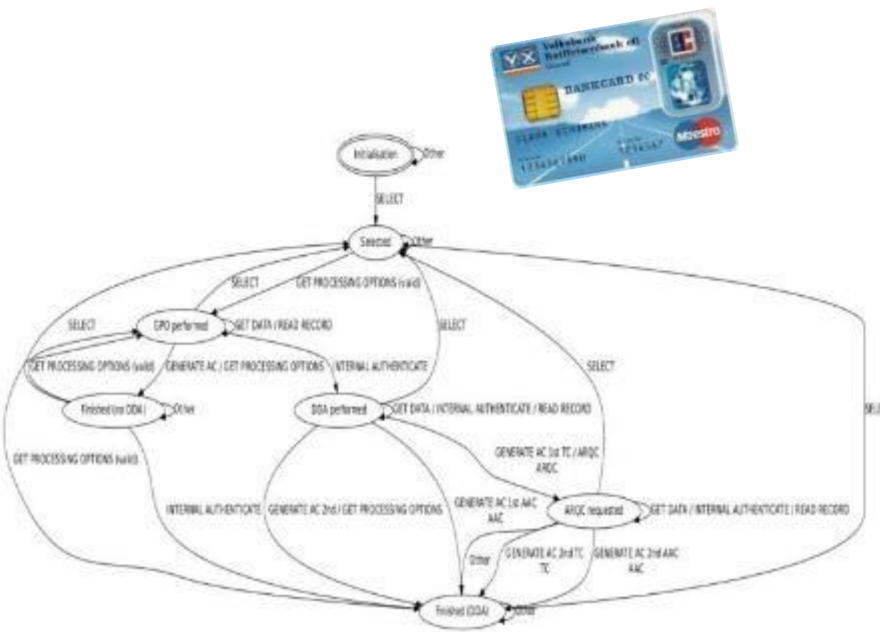
State machine inference of Maestro card



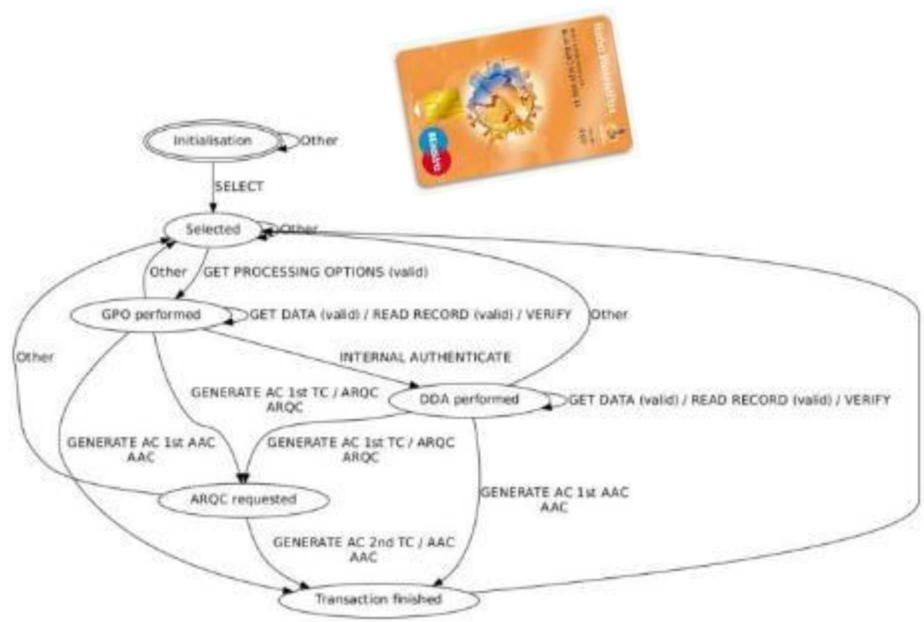
merging arrows with same start & end state

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

Using state machines for comparison



Volksbank Maestro implementation

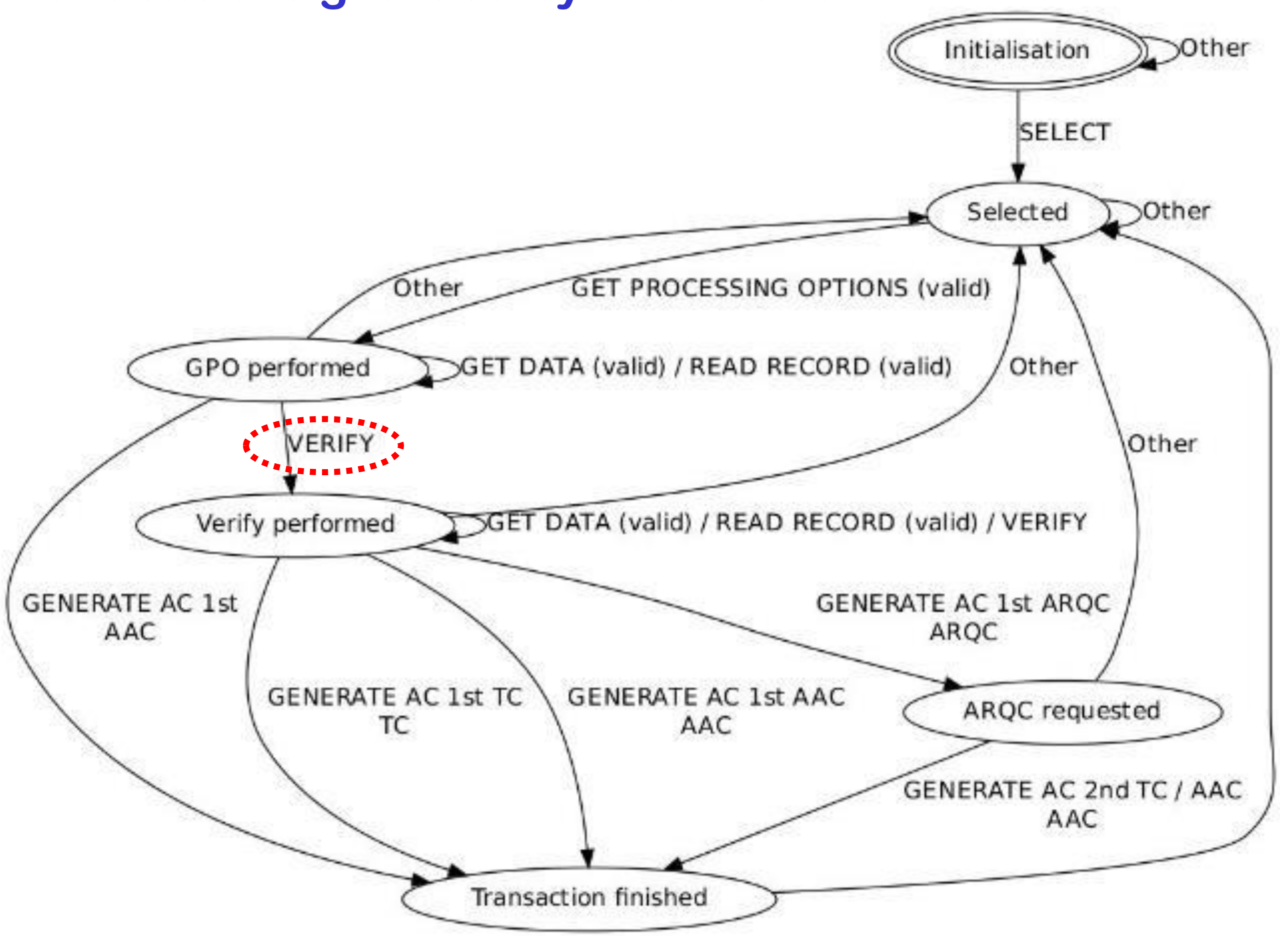


Rabobank Maestro implementation

Are both implementations correct & secure? Or compatible?

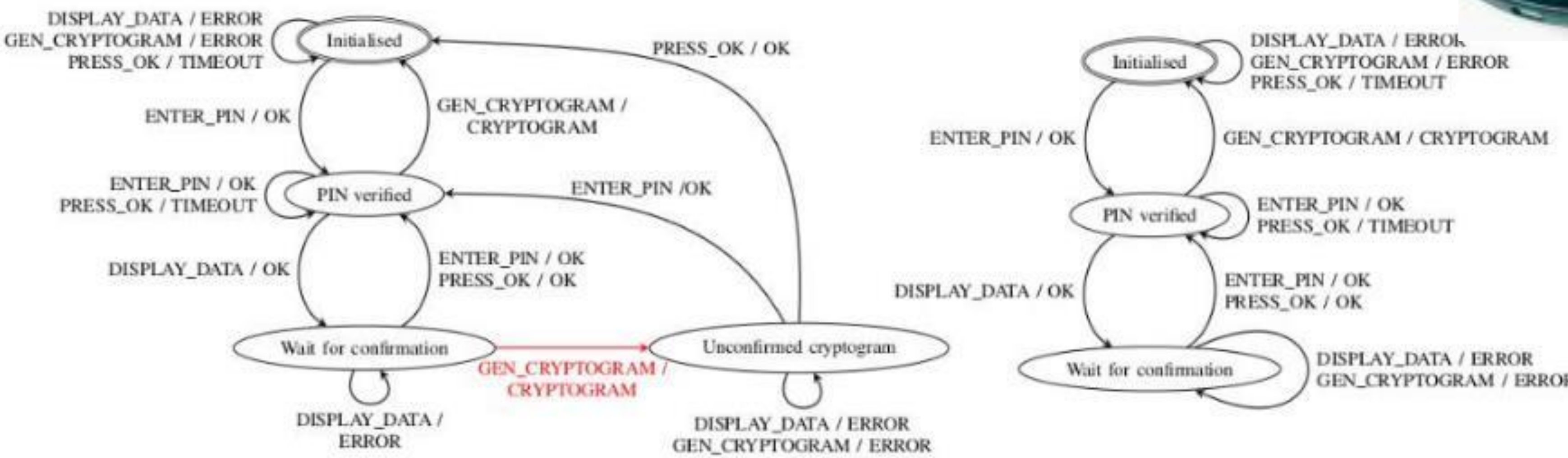
Using state machine for security analysis

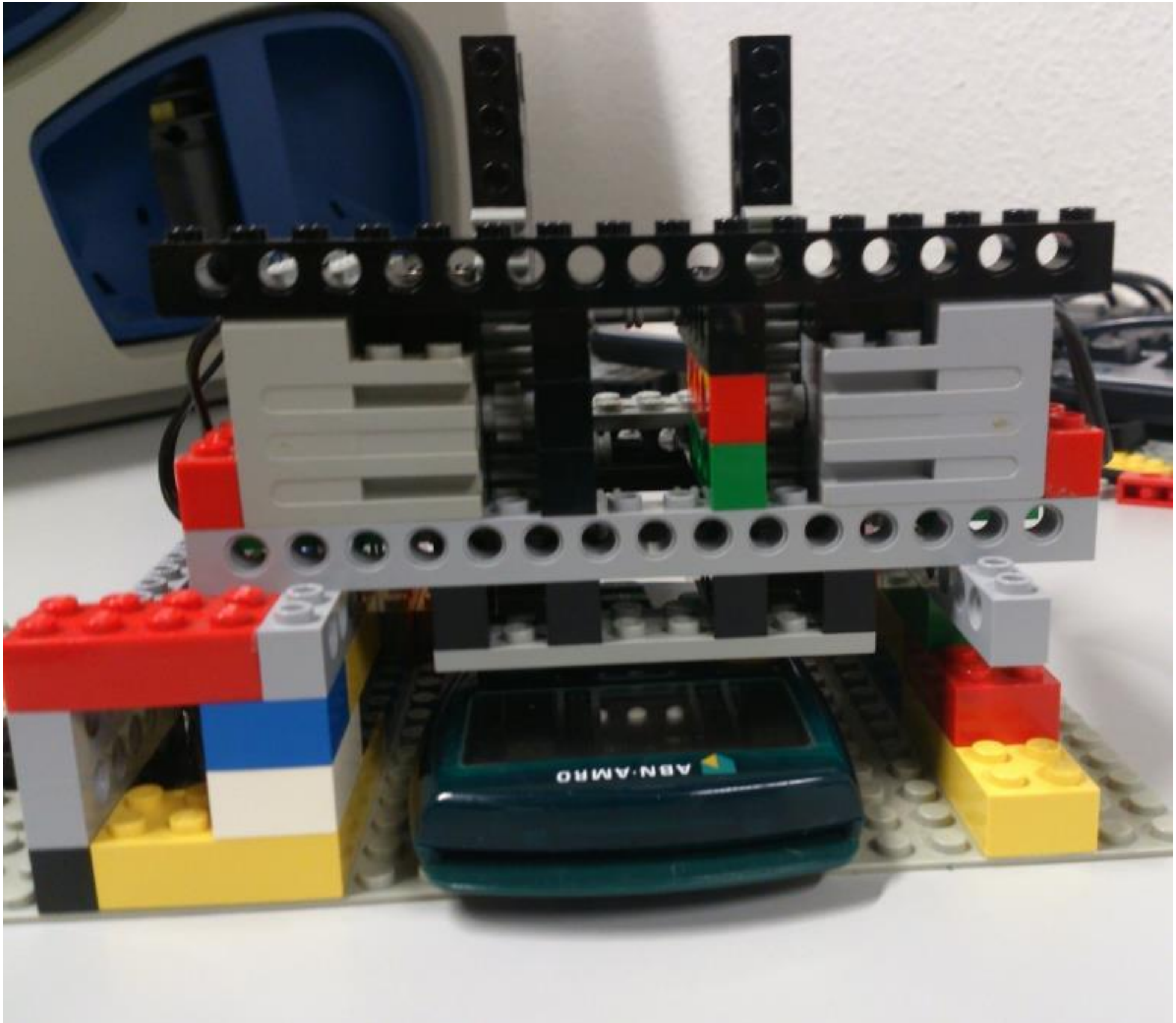
Which actions are guarded by PIN check?

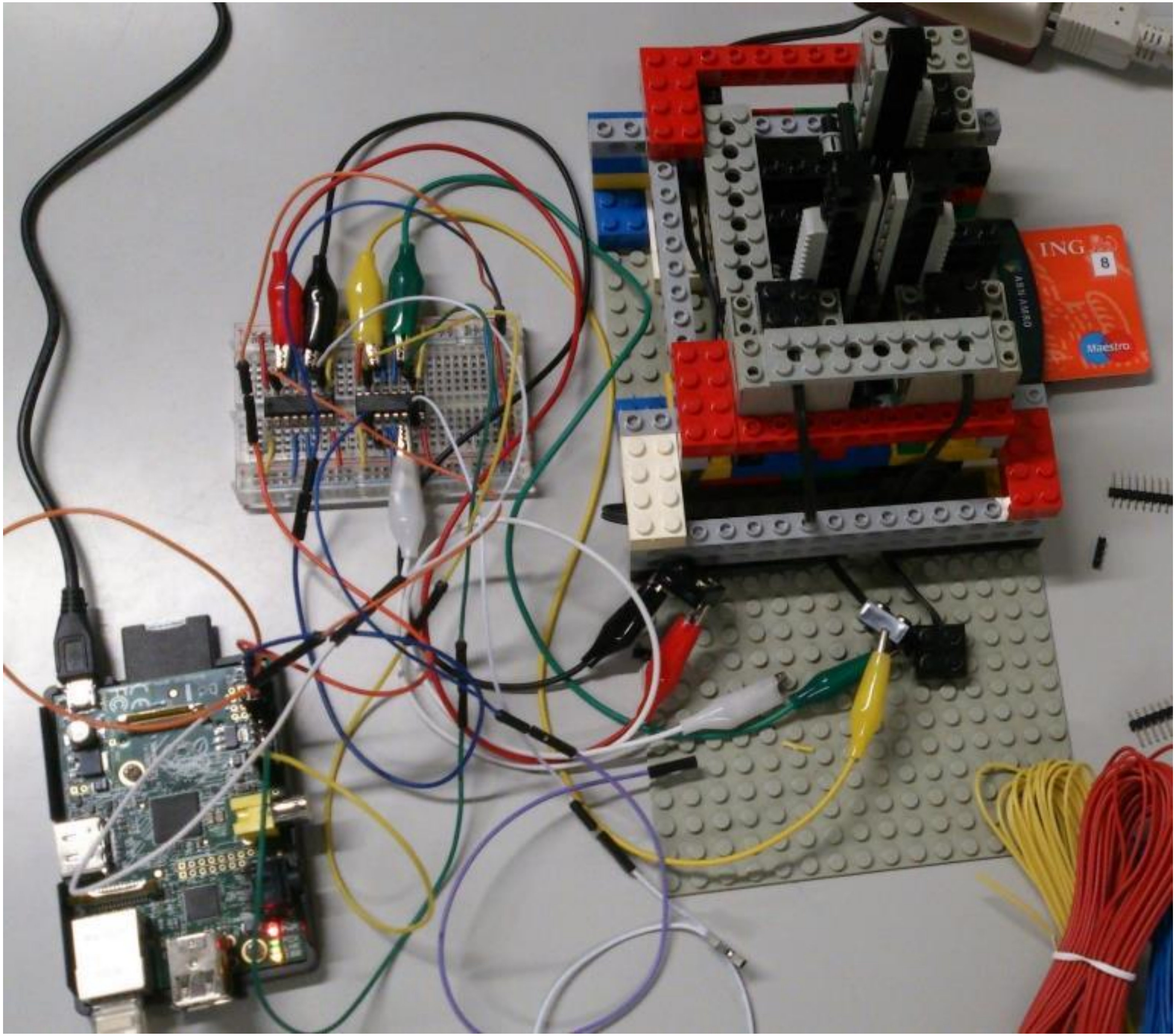


State machine of internet banking device

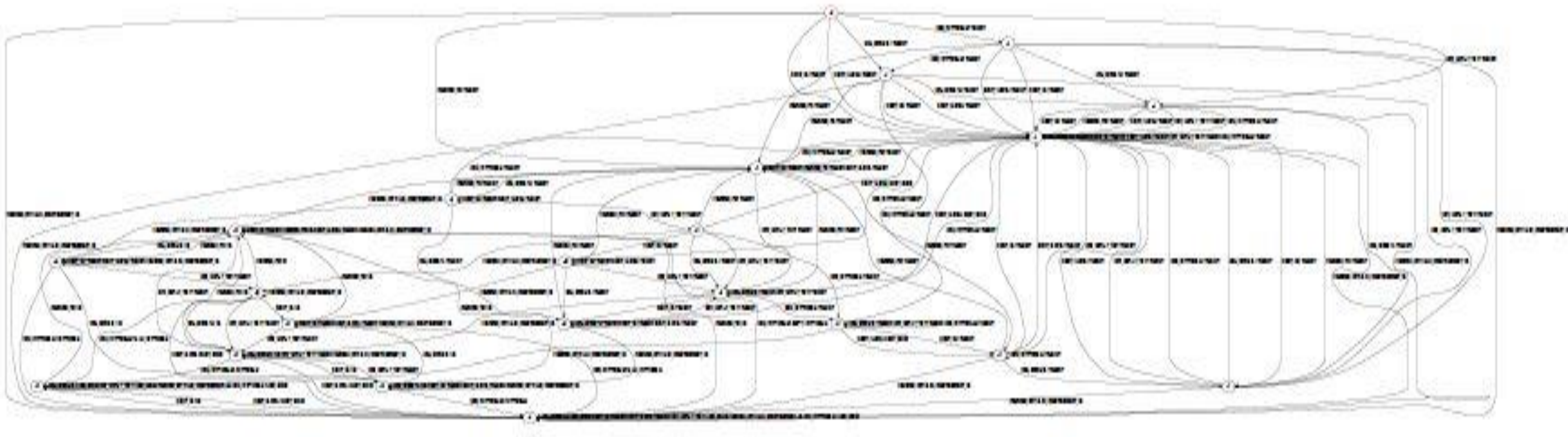
State machines inferred for flawed & patched device




ABN·AMRO


ING
8
Maestro

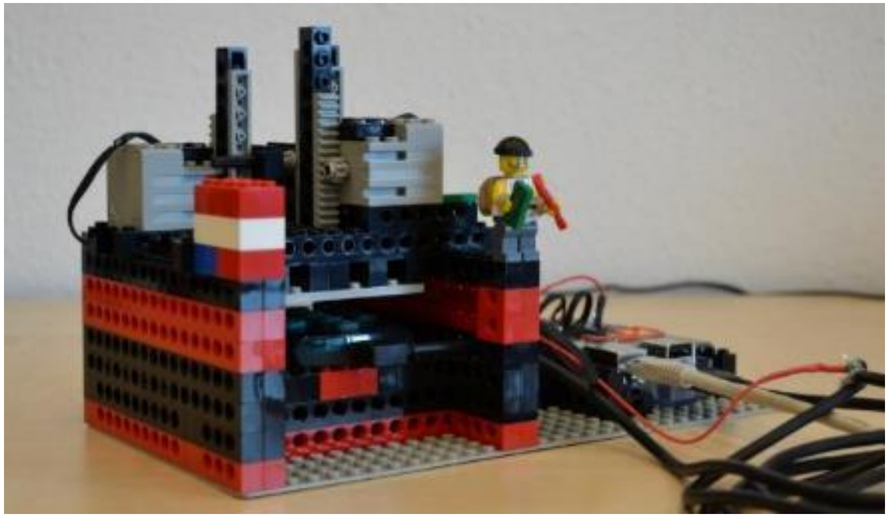
Complete inferred state machine



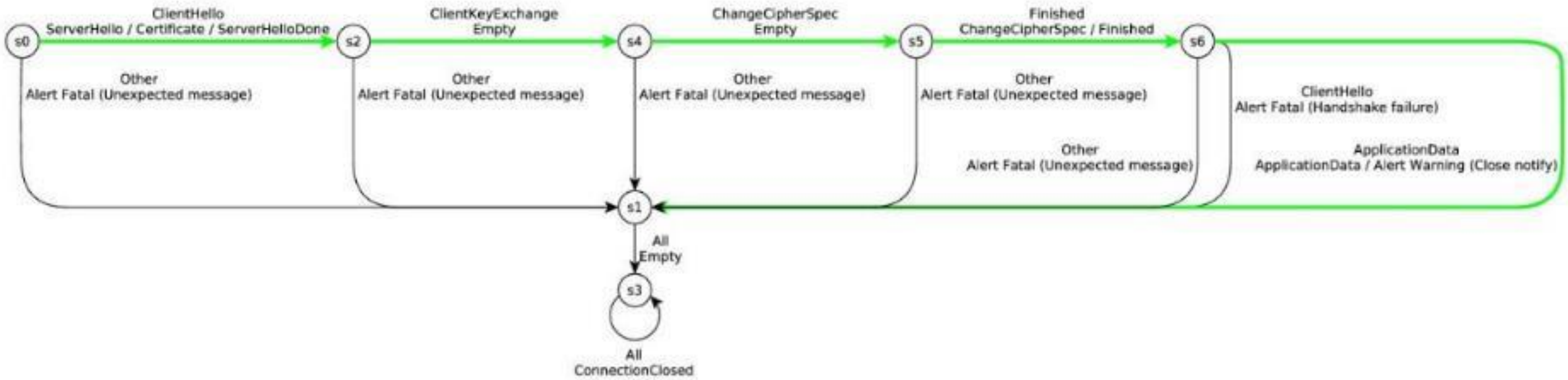
Would you trust this to be secure?

[Georg Chalupar et al.,
Automated reverse engineering using Lego,
WOOT 2014]

Movie at http://tinyurl/legolearn

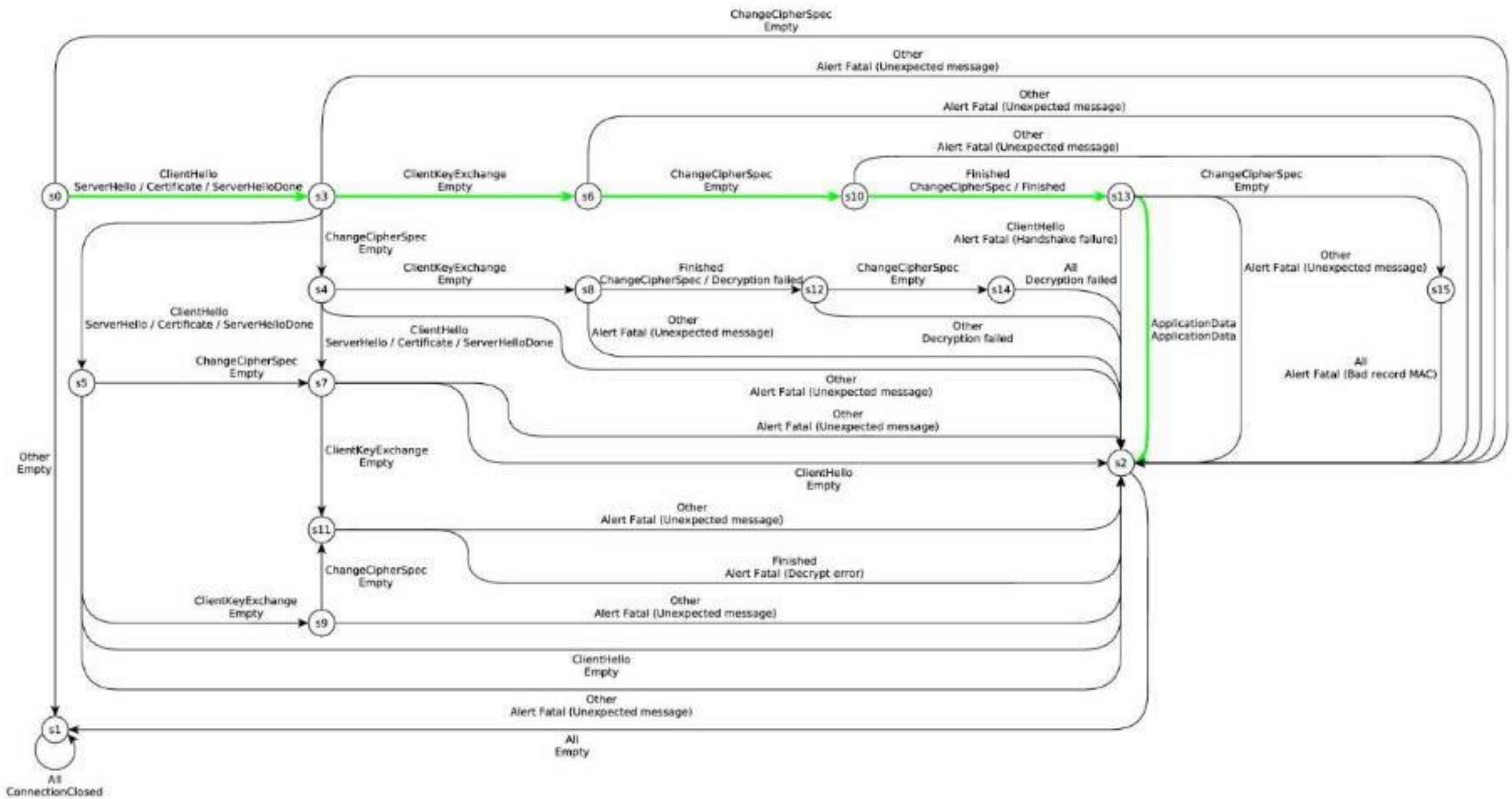


State machine of TLS

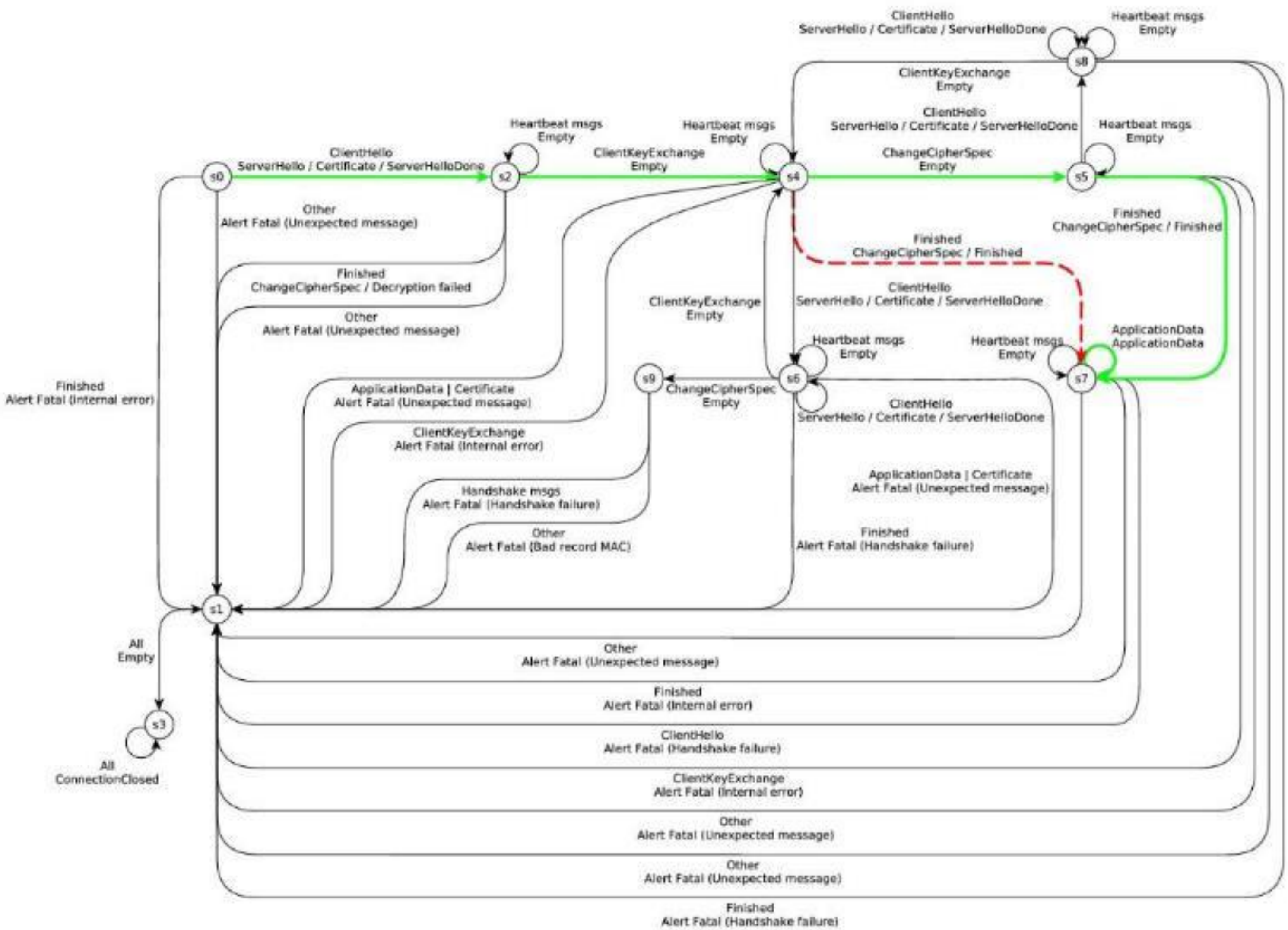


Protocol state machine of the NSS TLS implementation

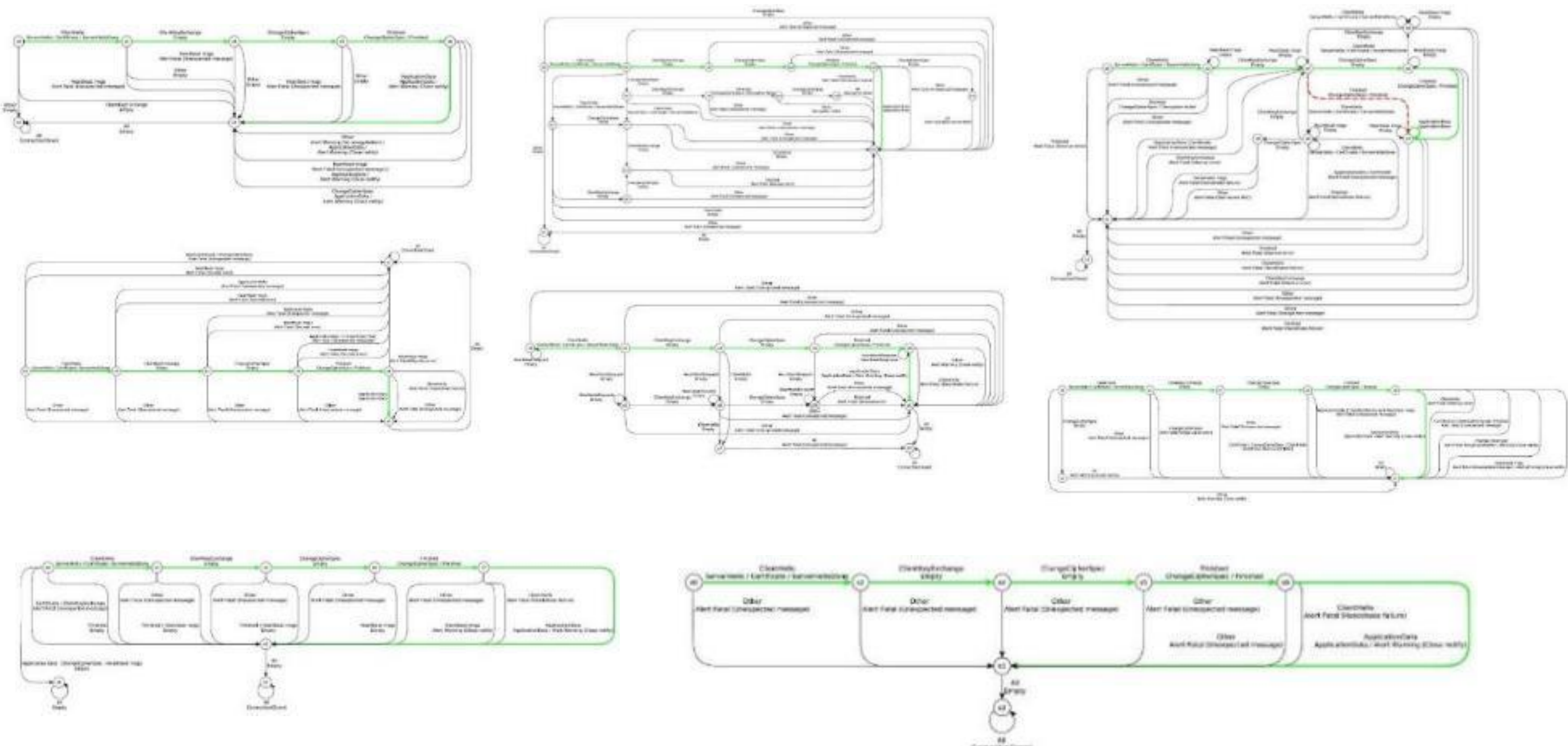
State machine of OpenSSL



State machine of Java Secure Socket Exchange



State machine inference of TLS implementations



All TLS implementations we analysed were different!
Why doesn’t the TLS spec include a state machine?

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


NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES

Forwarding flaws

[LangSec 2018]

[Strings considered harmful, Usenix login magazine, 2018]

Two types of INPUT problems

1. **Buggy parsing & processing**
   - Bug in processing input causes application to go of the rails
   - Classic example: buffer overflow in a PDF viewer, leading to remote code execution

   This is *unintended* behaviour, introduced by *mistake*

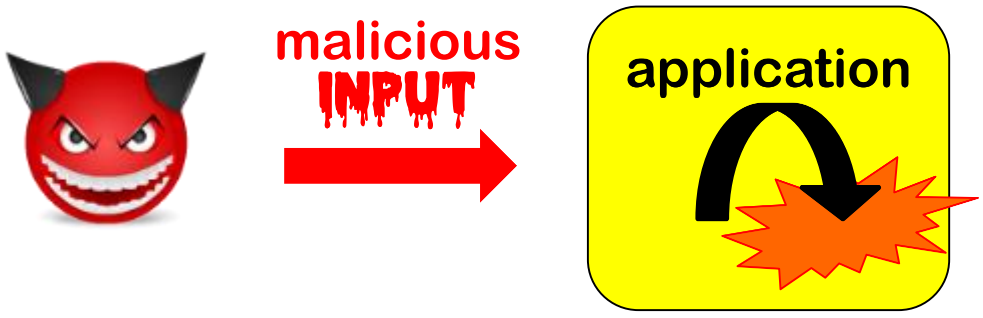
2. **Flawed forwarding** (aka injection attacks)
   - Input is forwarded to *back-end* service/system/API, to cause damage there
   - Classic examples: SQL injection, path traversal, XSS, Word macros

   This is *intended* behaviour of the back-end, introduced *deliberately*, but *exposed by mistake* by the front-end

Processing vs Forwarding Flaws

Processing Flaws

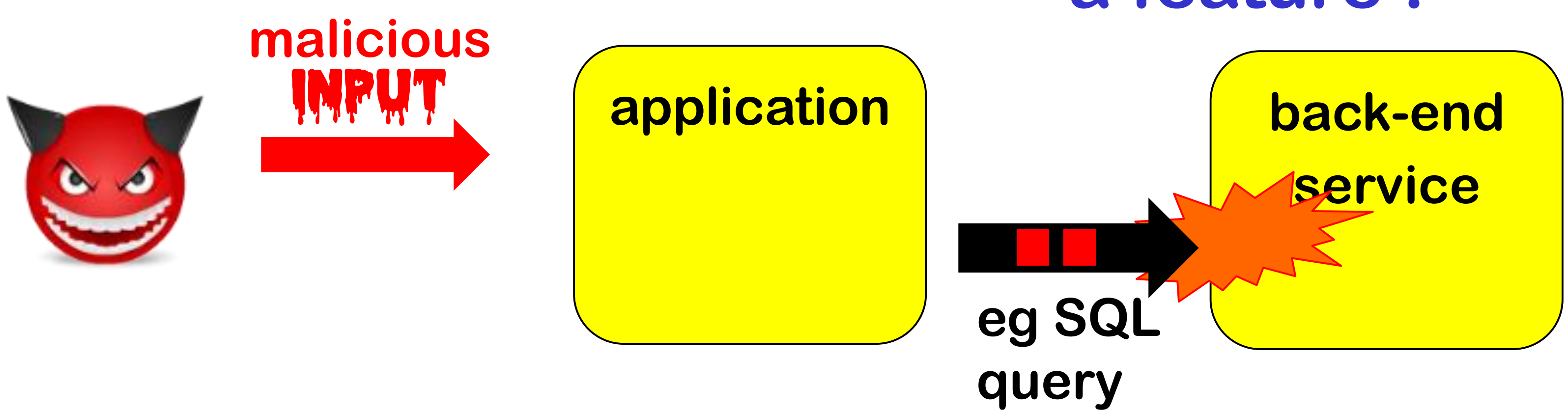
a bug !



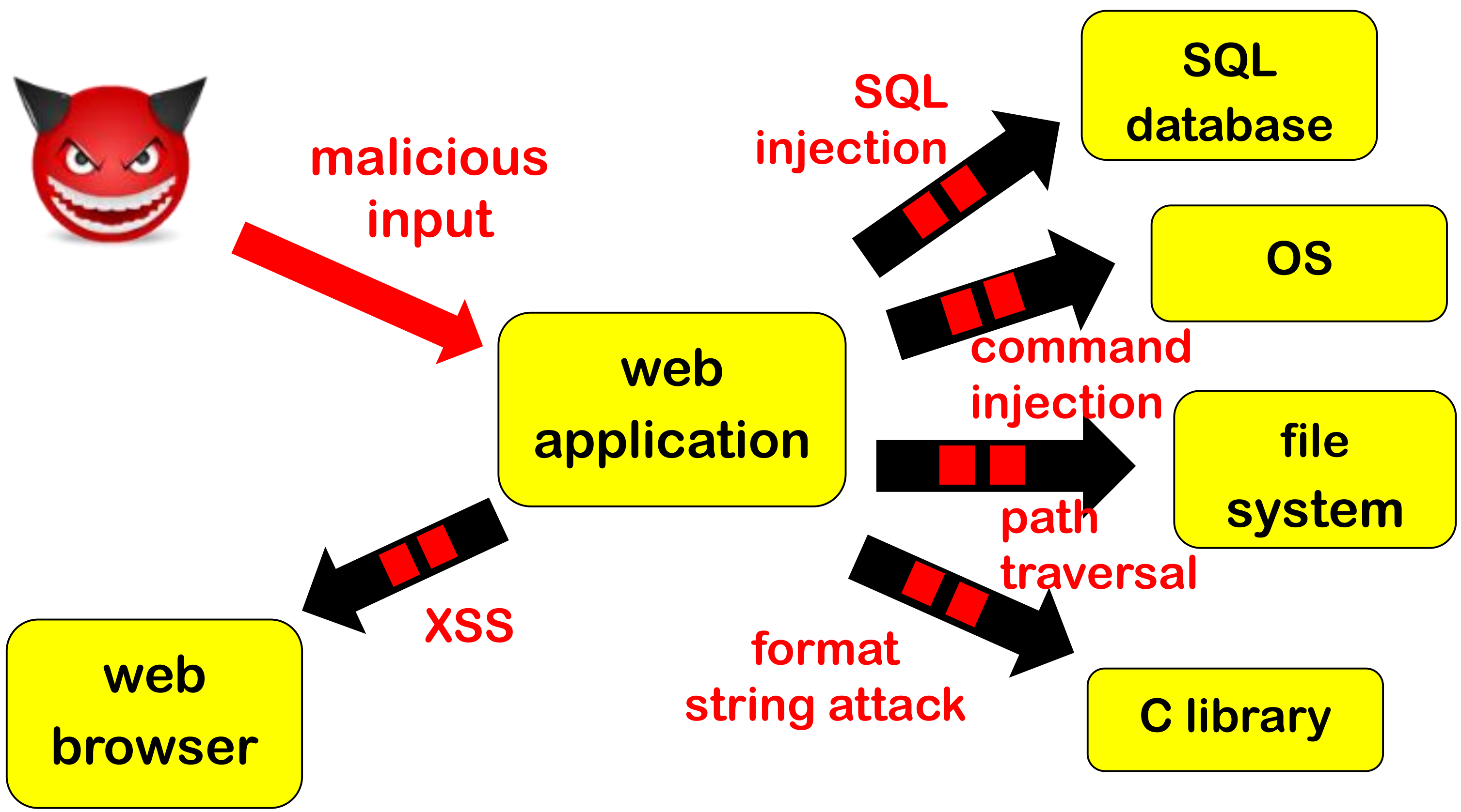
eg buffer overflow
in PDF viewer

Forwarding Flaws

(abuse of)
a feature !

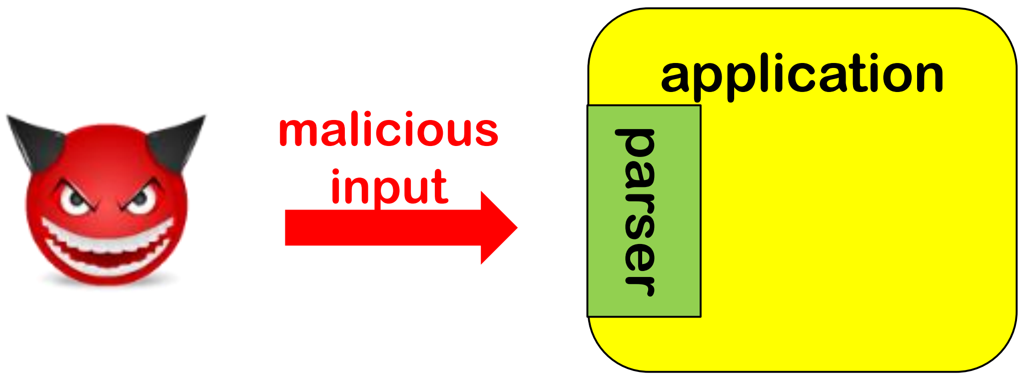


More back-ends, more languages, more problems



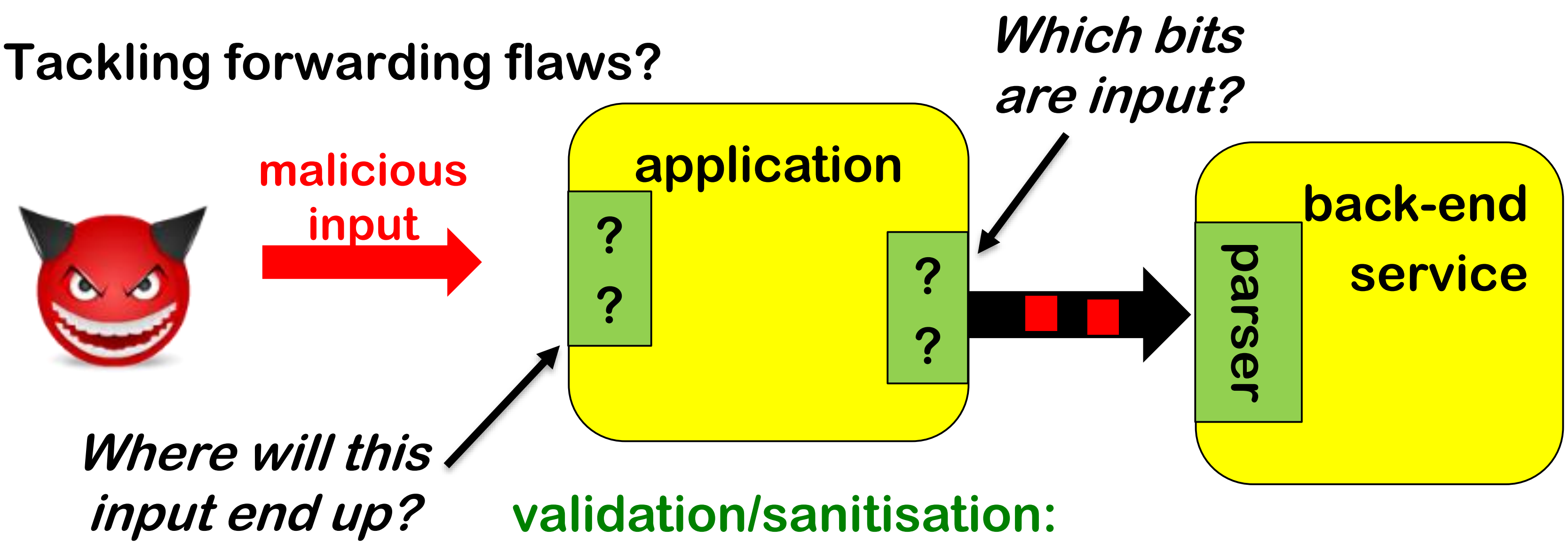
How & where to tackle input problems?

Tackling processing flaws



LangSec approach:
Simple & clear language spec;
generated parser code;
complete parsing before
processing

Tackling forwarding flaws?



validation/sanitisation:
filtering and/or escaping?

Anti-patterns
in tackling forwarding flaws

Anti-pattern: STRING CONCATENATION

- Standard recipe for security disaster:
  1. concatenate several pieces of data, some of them user input,
  2. pass the result to some API

- Classic example: SQL injection

- Note: string concatenation is *inverse* of parsing

Anti-pattern: STRINGS

The use of strings in itself is already troublesome

- be it `char*`, `char[]`, `String`, `string`, `StringBuilder`, ...

- Strings are *useful*, because you use them to represent many things:
  eg. name, file name, email address, URL, shell command, bit of SQL, HTML,...

- This also make strings *dangerous:*

  1. Strings are unstructured & unparsed data, and processing often involve some interpretation (incl. parsing)
  2. The same string may be handled & interpreted in many
     – possibly unexpected – ways
  3. A string parameter in an API call can – and often does – hide a very expressive & powerful language

Remedies
to tackle forwarding flaws

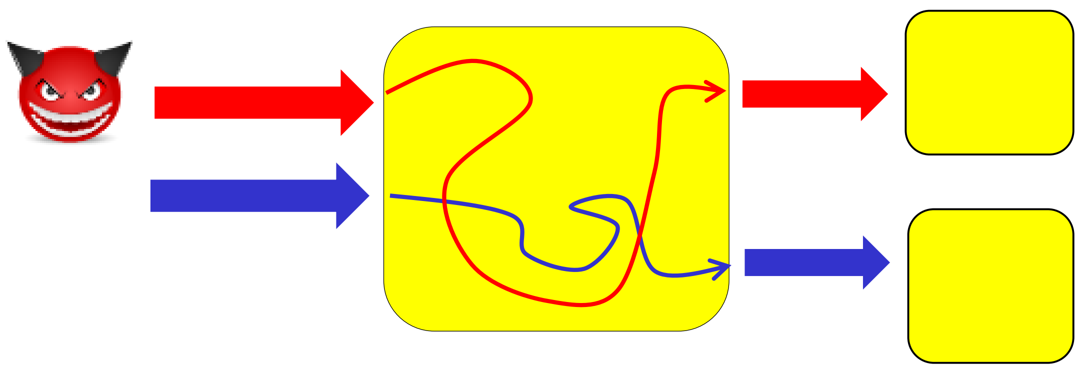
Types to the rescue!

Remedy: Types (1) to distinguish *languages*

- Instead of using strings for everything,
  use different types to distinguish different kinds of data

  Eg different types for HTML, URLs, file names, user names, paths, …

- Advantages
  - Types provide structured data
  - No ambiguity about the intended use of data

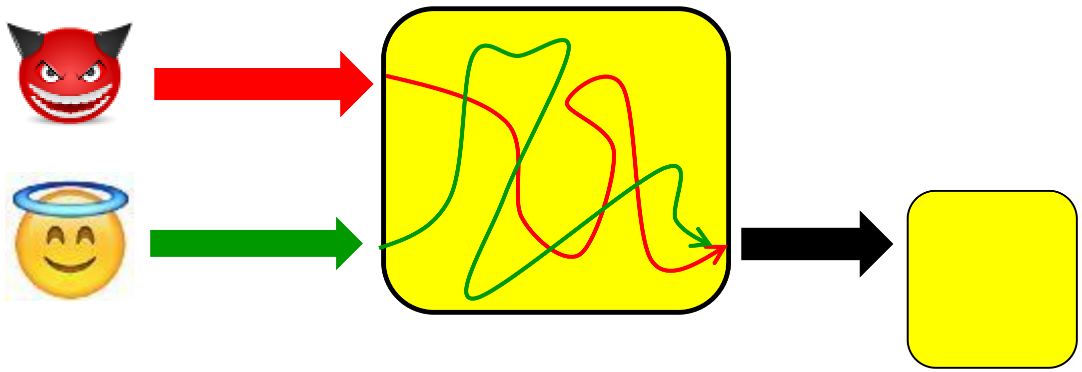


Remedy: Types (2) to distinguish *trust levels*

- Use information flow types to track the origins of data

  and/or to control destinations

  - Eg distinguish untrusted user input vs compile-time constants



The two uses of types, to distinguish (1) languages or (2) trust levels, are orthogonal and can be combined.

Example: Trusted Types for DOM Manipulation

DOM-based XSS flaws are proving difficult to root out.

The DOM API is string-based, where strings can be HTML snippets, pieces of javascript, URLs, …

Google's Trusted Types initiative [https://github.com/WICG/trusted-types] replaces string-based DOM API with a typed API

- **using `TrustedHtml, TrustedUrl, TrustedScriptUrl, TrustedJavaScript,…`**
- **'safe' APIs for back-ends which auto-escape or reject untrusted inputs**

Now released as a Chrome browser feature

[https://developers.google.com/web/updates/2019/02/trusted-types]

Conclusions

Conclusions

- Software play central role in cyber security
- Many security problems arise in INPUT handling
  - buggy parsing
  - buggy protocol state machines
  - unintended parsing due to forwarding

  Ironically, parsing is a well-understood area of computer science…
- LangSec provides some constructive remedies to tackle this
  - Have clear, simple & well-specified input languages
  - Generate parser code
  - Don't use STRINGS
  - Do use types, to distinguish languages & trust levels

Postel's Law

'Be liberal in what you expect, be strict in what you send'

- aka Robustness Principle, originates from the RFC for TCP
- In the short run:

  a great way to quickly get implementations to work together
- In the long run:

  a recipe for lots of security headaches

Thanks for your attention

References

On LangSec

- **Lots of papers at http://langsec.org,**
  e.g. the LangSec manifesto http://langsec.org/bof-handout.pdf

On state machine inference:

- **Georg Chalupar, Stefan Peherstorfer, Erik Poll and Joeri de Ruiter,**
  ***Automated Reverse Engineering using LEGO*, WOOT 2014**
- **Joeri de Ruiter and Erik Poll,**
  ***Protocol state fuzzing of TLS implementations*, Usenix Security 2015**
- **Erik Poll, Joeri de Ruiter and Aleksy Schubert,**
  ***Protocol state machines and session languages*, LangSec 2015**

On forwarding attacks

- **Erik Poll, *LangSec revisited: input security flaws of the 2nd kind*, LangSec 2018**
- **Erik Poll, *Strings considered harmful*, Usenix login magazine, 2018**