Security by Construction

Erik Poll

Digital Security

Radboud University Nijmegen

Security by Construction ?!?

Security by Construction:

Forget about it,

it ain't gonna happen

Some Security by Construction

thanks to LangSec

Overview

1. General observations about security
   - Why we need & want security by construction
   - Why security is hard to do by construction

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

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

Background & motivation

For the past decade I've been

- trying to apply formal methods to security
- teaching software security

Much security research & teaching is the polar opposite of constructive

- Security research is often post-hoc and destructive:
  - Vulnerability research looks for clever ways to attack systems
- Teaching by counterexample:
  - ie. showing students entertaining examples of security flaws

Can we take a more systematic approach?

Why security by construction would be great!

Cyber security is huge & still growing problem

NotPetya: World's First $10 Billion Malware

Pharmaceutical giant Merck confirmed NotPetya attack disrupted operations worldwide

Shipping company Maersk says June cyberattack could cost it up to $300 million

What's new with security:
skilled people with lots of resources are actively looking for bugs to exploit.

https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world

Root cause: SOFTWARE

- Systems can be hacked because there is software in them

- Software is *the* main root cause of security problems
  - Only other cause of problems: humans ('*social engineering*')

- Cyber security is *our* problem, as software engineering community
  - Don't count on security researchers, network security people, cryptographers, … to solve this

software security ≠ security software

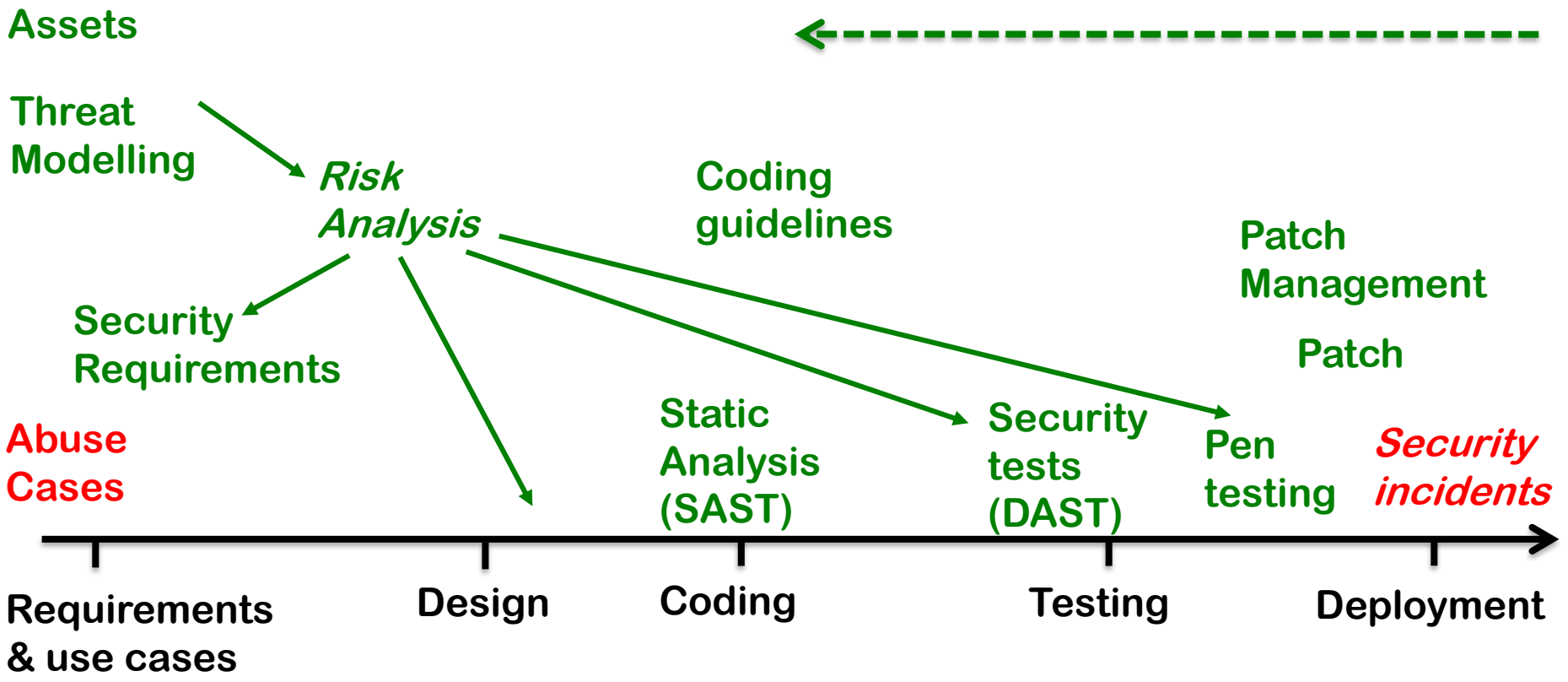
- Security software = software implementing security controls or functionality
  - such as security protocols (eg TLS), access control mechanisms, login procedures, disk encryption, …
- Obviously, security software needs to be correct & secure, and we could try and specify it & get it right by construction
- However, *ALL* software needs to be secure, not just the security software
  - eg device drivers, PDF viewers, MS Office, FaceTime…
  - Of course, we can & should use compartmentalisation to reduce the TCB

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

- Woody Allen

Security in Software Development Lifecycle

Holy grail: Security-by-Design



Why security (by design) is tricky…

Specifying security

- specification = WHAT

  program = HOW

  security specification = WHAT NOT

- WHAT NOT is not so useful during construction…

- Should the specification ('correctness') subsume / imply security?
  - For security software, it might?
  - If the spec allows refinement, then it will not?

Specifying security - WHAT NOT

- The good news: WHAT NOT can be orthogonal to functionality
  - So maybe we can re-use security specs for multiple systems?
  - Indeed, there are useful generic lists of common security flaws
    - OWASP Top 10
    - CWE/SANS Top 25

      This is top 25 out of 702 (!) common security weaknesses

- The bad news: WHAT NOT is hard to specify exhaustively

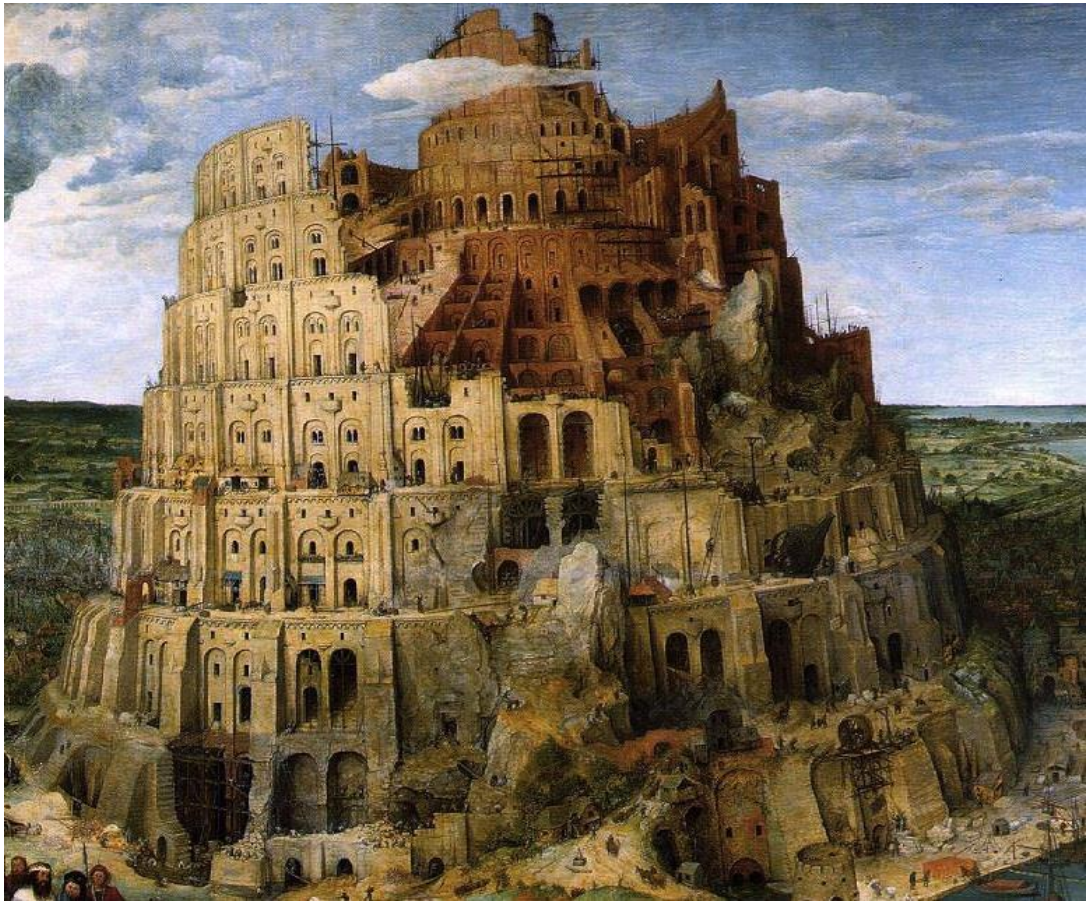
  'There are unknown unknowns' – Donald Rumsfeld

Some ways to specify (some) security

- Temporal logic or security automata
  - eg. action X only possible after entering PIN code
- Information flow properties
  - enforced using typing, static analysis, or deductive verification
- Precondition TRUE in contracts for public interfaces
  - Not just `{P} S {Q}`

    but also `{not P} S {nothing 'bad' happened}`
  - Eg. prove safety conditions

    `{true} S {no RuntimeException}`



LangSec

(language-theoretic security)

LangSec

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

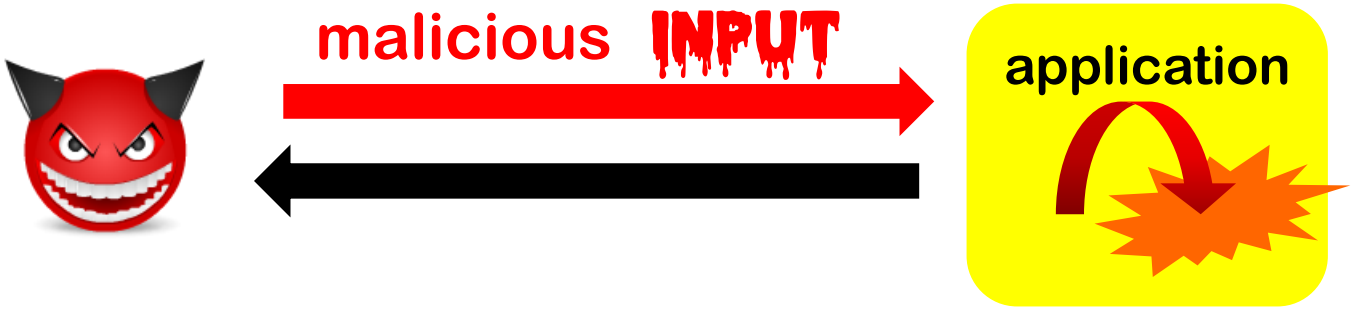
Sergey Bratus & Meredith Patterson
'The science of insecurity'
CCC 2012

- The language in Language-theoretic Security refers to input languages, not modelling or programming languages.

Common theme: INPUT

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

eg buffer overflow, format string attack, command injection, path traversal, SQL injection, XSS, CSRF, Word macros, XML injection, LDAP injection, zip bombs, deserialization attacks, …



- Garbage In, Garbage Out

  leads to

  Malicious Garbage In, *Security Incident* Out

Fallacy of classic input validation?

Classical input validation aka input sanitisation

> remove or encode harmful characters (eg ; ' ")
> before processing inputs

But...

- Which characters are harmful depends on the language/format, and a typical application handles *many* languages.

  Eg ' problematic for SQL database, < > for web app, & for LDAP server

- Instead of validating input before feeding it to crappy software that processes it, maybe that software should be more robust?

  - esp. the parsing it performs as part of any processing

SMS of Death

Text message that used to crash iPhones:

- **Should telco filter SMS to remove these dangerous Unicode combinations?**
- **Should the baseband chip in an iPhone filter out these combinations?**
- **Or should iPhone software be robust in dealing with arbitrary combinations of Unicode?**

So, is input validation always the right way to prevent input problems?

LangSec: root causes

- 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/UMTS/LTE, Ethernet,

  OpenVPN, SSH,

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

  URLs, email addresses, S/MIME,

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

  Bluetooth,

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

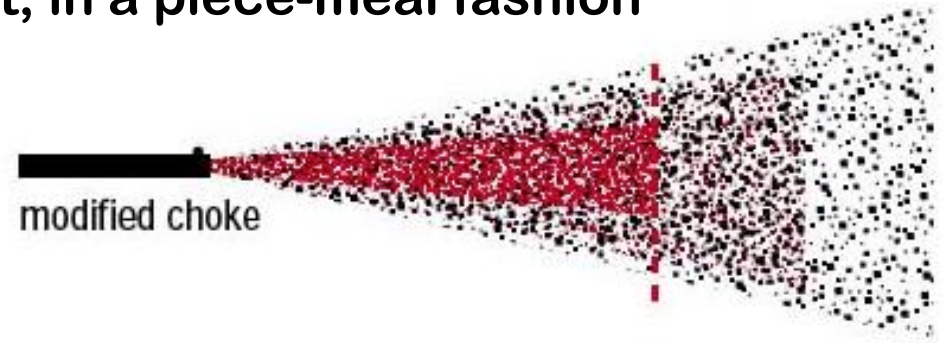
LangSec: root causes of security problems

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

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

- *Mixing input recognition & processing*

  esp. in shotgun parsers, handwritten 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

   eg with regular expression or EBNF grammar

2. *Generated* parser code

3. *Complete parsing before processing*

   So don't substitute strings & then parse,

   but parse & then substitute in parse tree

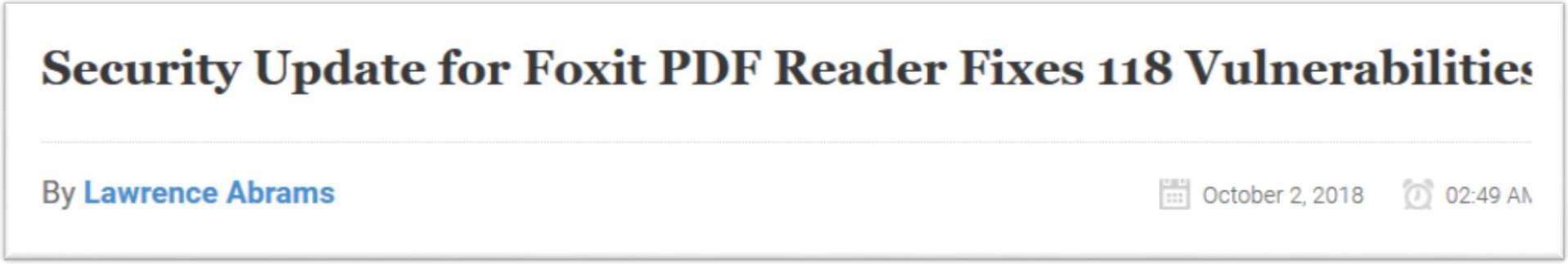
   (eg. parameterised queries instead of dynamic SQL)

4. *Keep the input language simple & clear*

   So that equivalence of parsers is ideally decidable

   So that you give minimal processing power to attackers

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
  - Multiple versions, some include JavaScript, some include Abode's proprietary ActionScript....
- These Foxit bugs are mainly memory 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 horrendously 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

- ANS.1 attacks in X.509 certificates

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

  `paypal.com\00mafia.org`

- PKCS#10-tunneled SQL injection

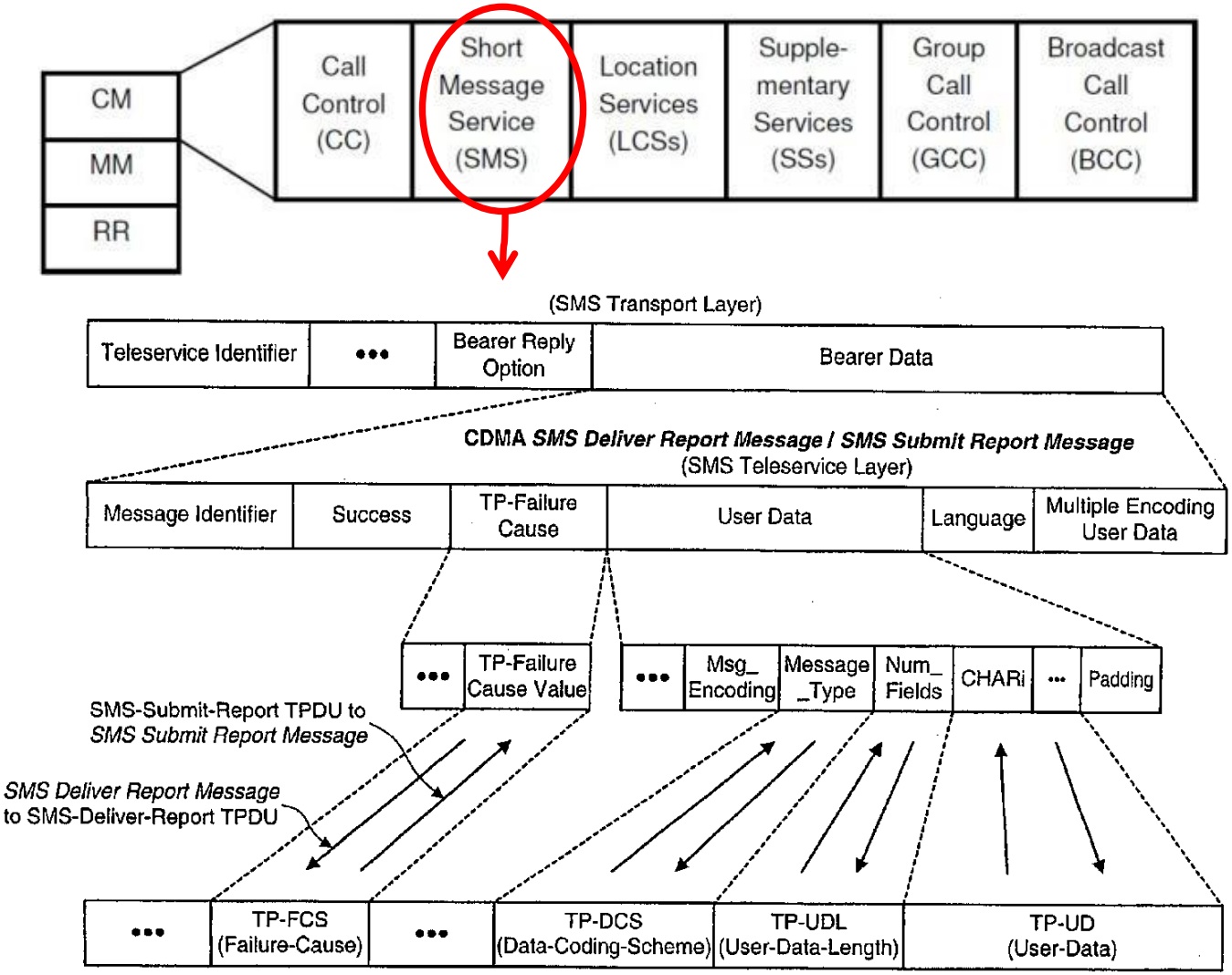
  SQL command inside a BMPString, UTF8String or UniversalString used as PKCS#10 Subject Name

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

Processing complex input languages *will* go wrong

Eg GSM specs

for SMS text messages



Unsurprisingly, malformed GSM traffic will trigger lots of problems

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

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM quickly crashes many phones!

It also reveals weird functionality in GSM standard and phones

Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM quickly crashes many phones!

It also reveals weird functionality in GSM standard and phones

- **eg warnings about receiving faxes (!?)**

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

LangSec in slogans

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


FULL RECOGNITION
BEFORE PROCESSING


NO MORE
Turing-
Complete
Input
Languages

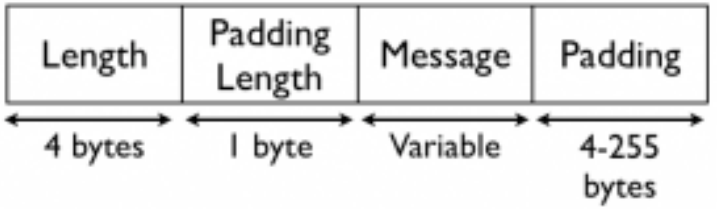
LangSec continued:
Protocol state machines

(Advertisement for LearnLib)

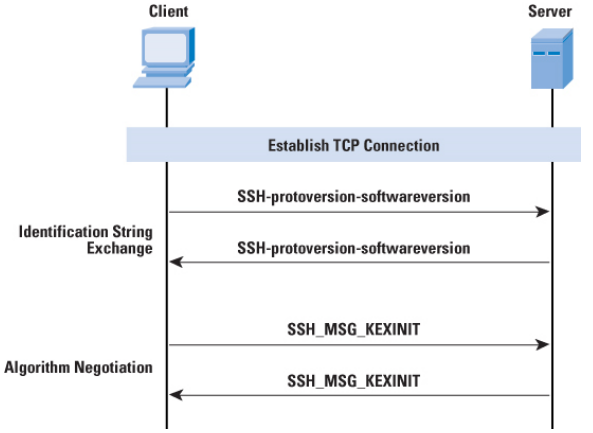
[LangSec 2015 paper]

Sequences of inputs

- Many protocols not only involves a language of input messages



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



- Most specs only describe the happy flow…
- For security protocols, getting unhappy flows correct is crucial
- Fortunately, we can extract these state machines from code - or hardware - using active learning

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

Active model learning of Maestro card

Active model learning of Maestro card

merging arrows with identical response

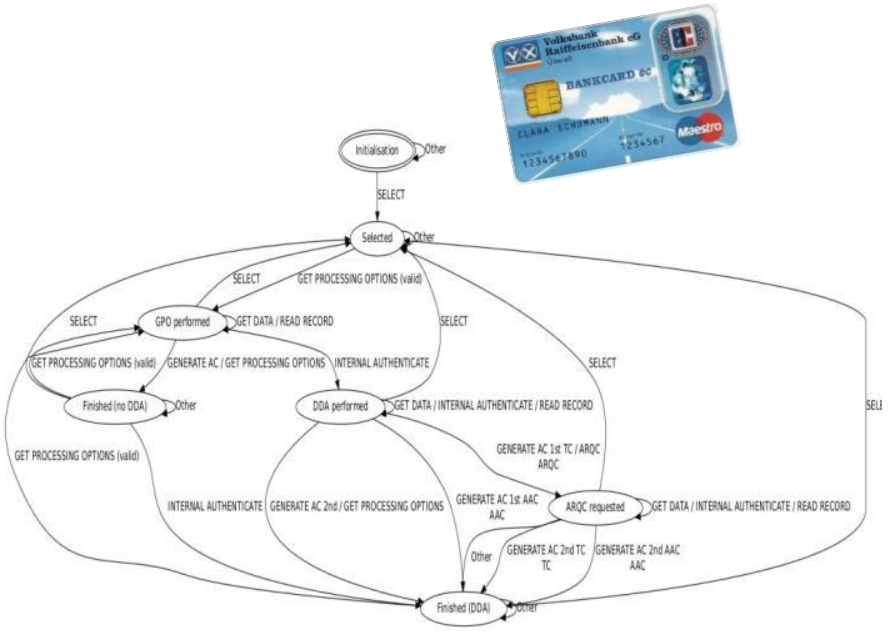
Active model learning of Maestro card

merging arrows with same start & end state

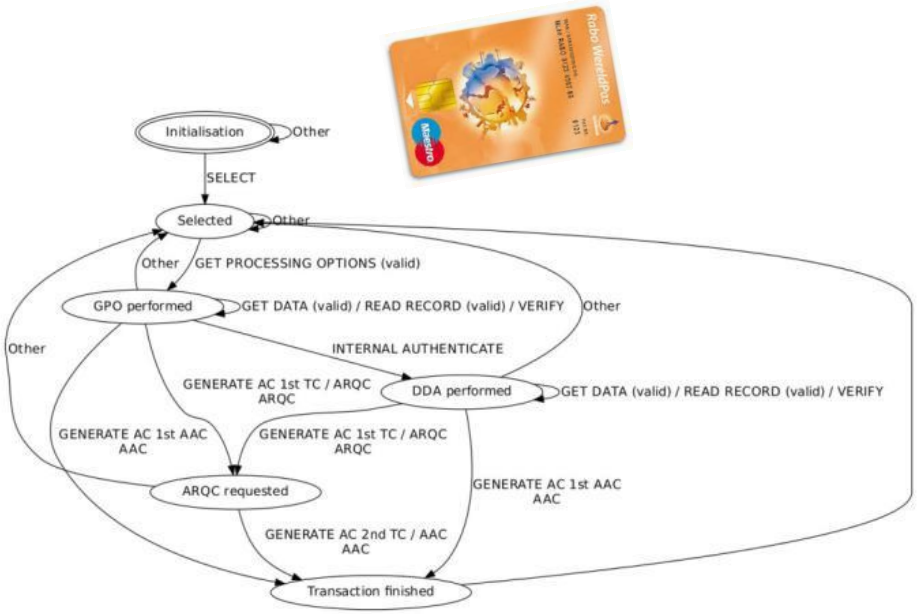
We found no bugs, but lots of variety between cards.

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

Using state machines for comparison



Volksbank Maestro implementation

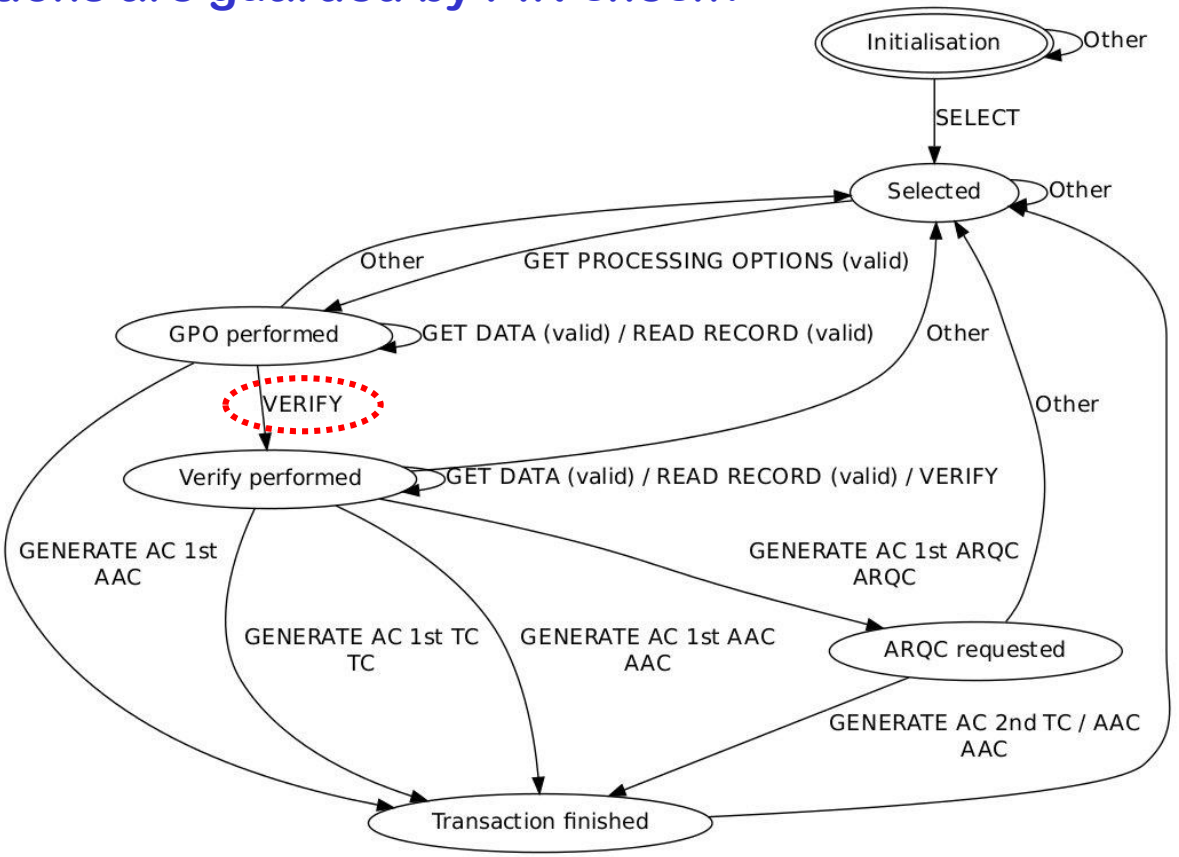


Rabobank Maestro implementation

Are both implementations correct & secure? Or compatible?

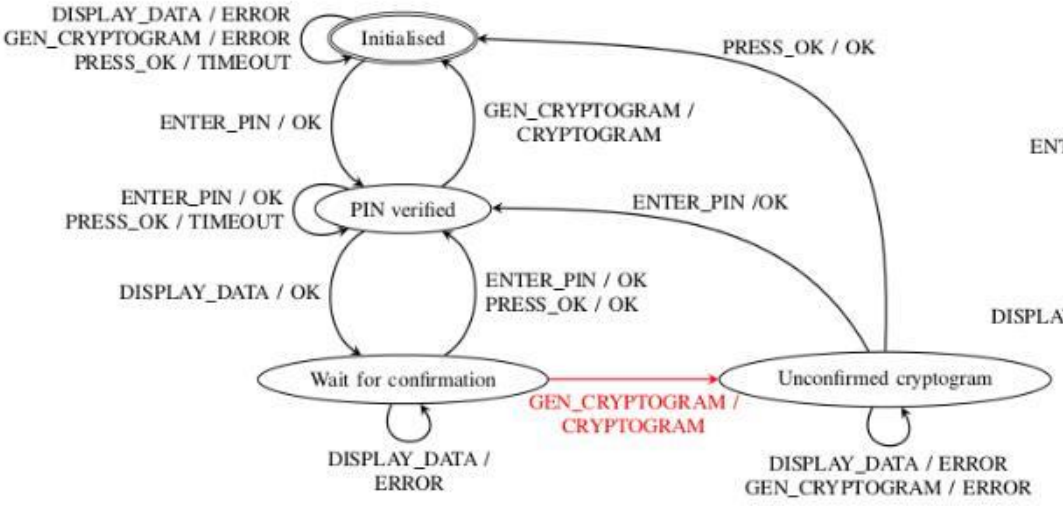
Using state machine for security analysis

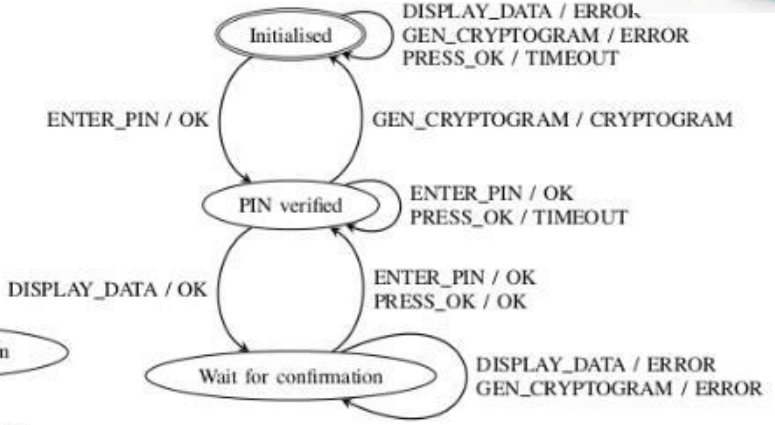
Which actions are guarded by PIN check?

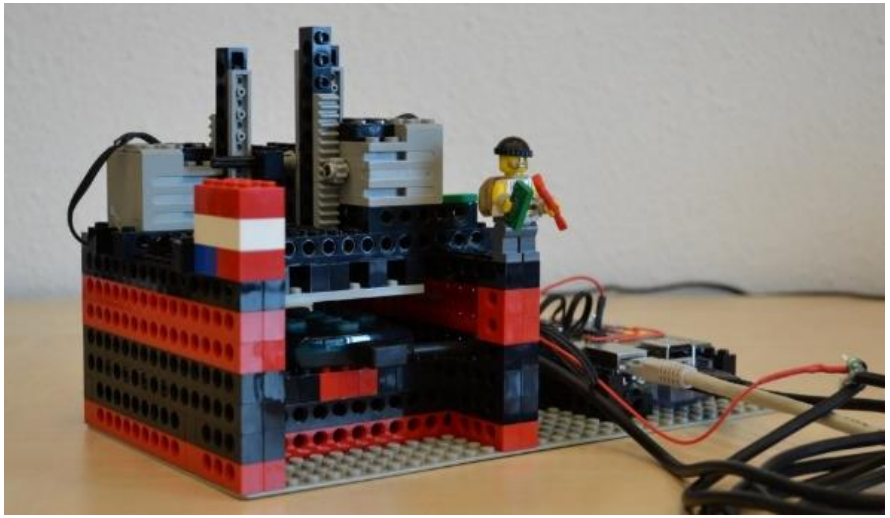


Active learning of internet banking device

State machines inferred for flawed & patched device

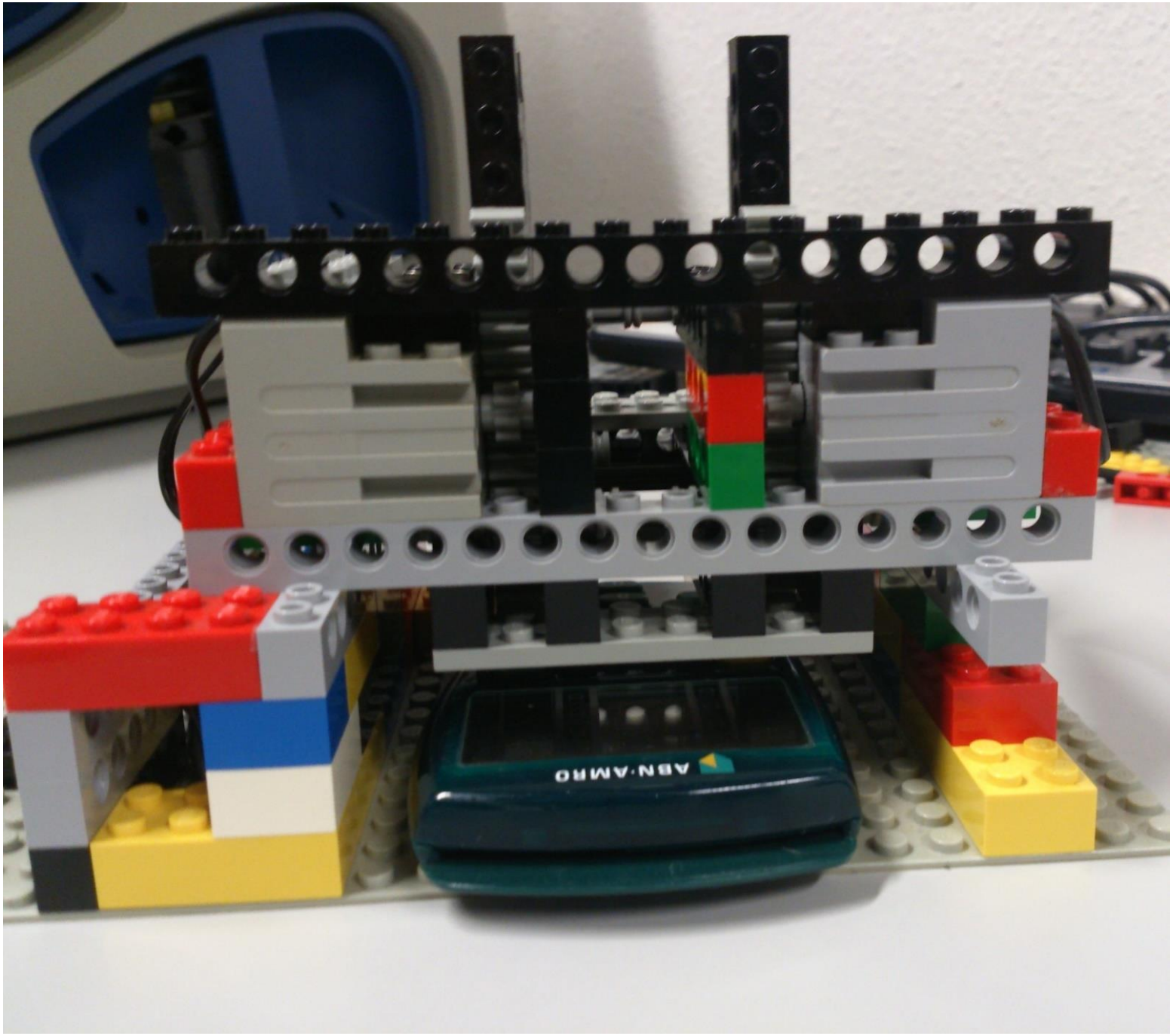


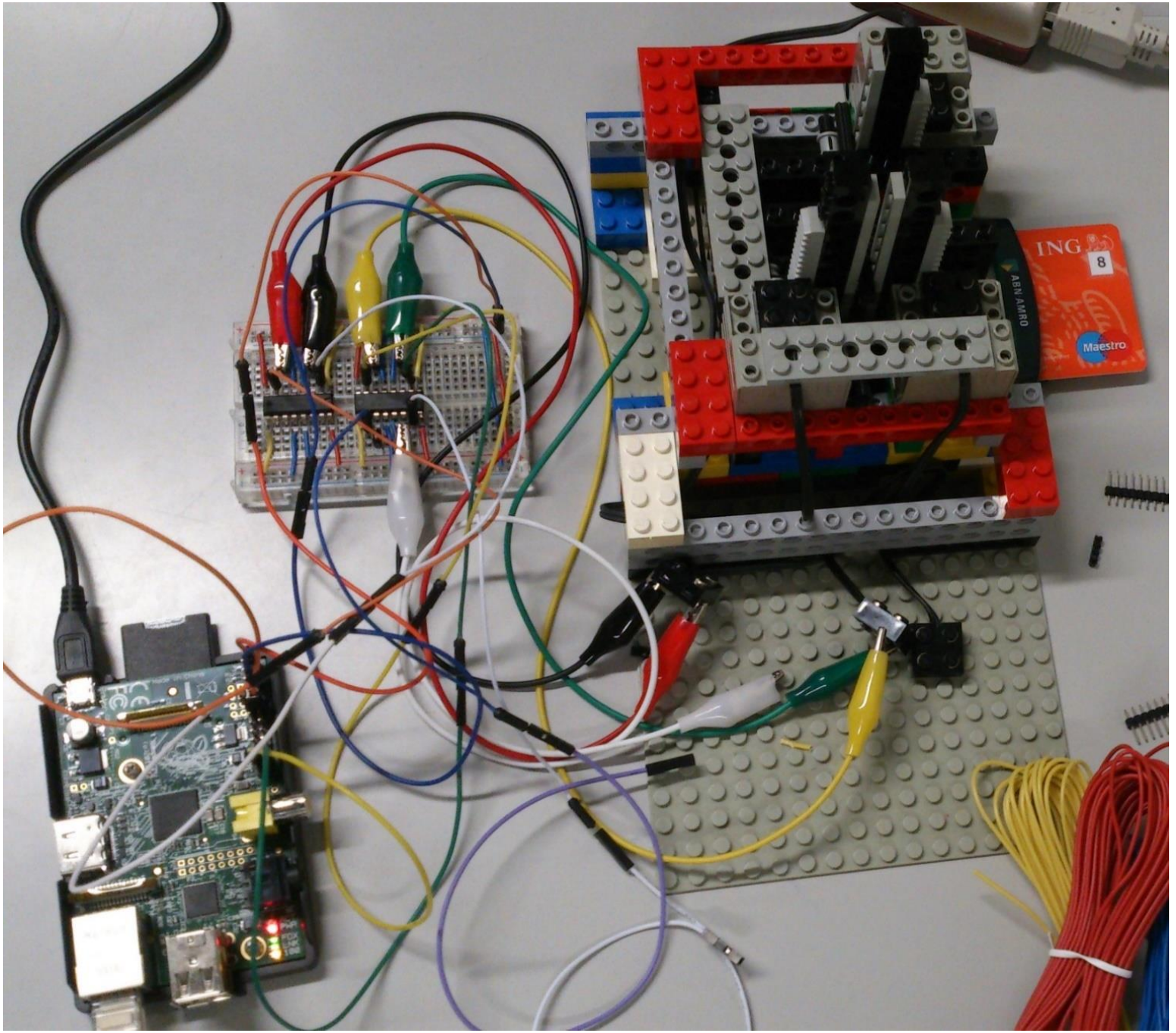




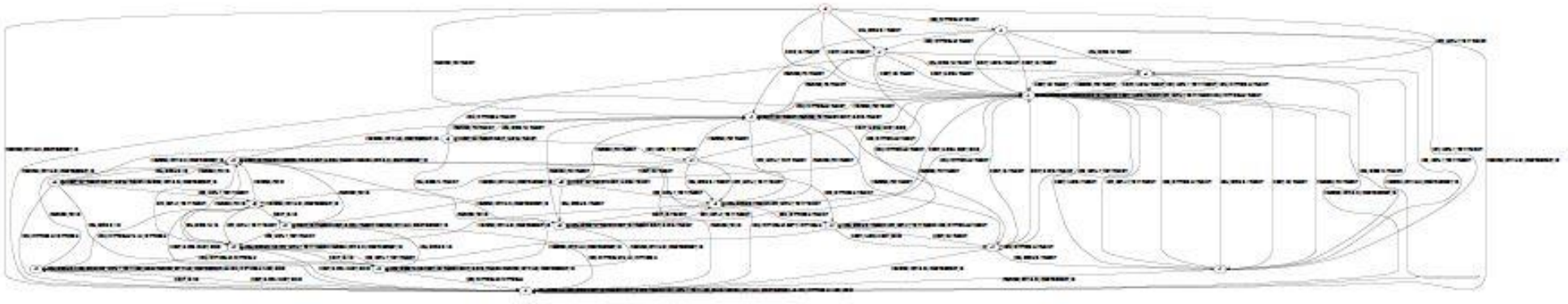
[Georg Chalupar et al.,
Automated reverse engineering using Lego,
WOOT 2014]

Movie at http://tinyurl/legolearn


ABN·AMRO


ING
8
ABN AMRO
Maestro

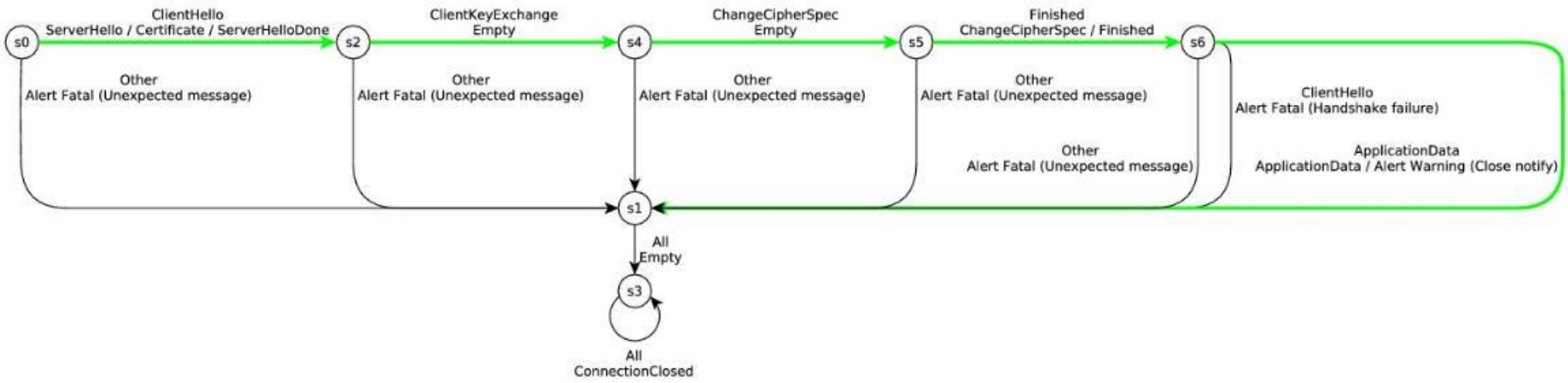
Active learning of internet banking device



More complete state machine

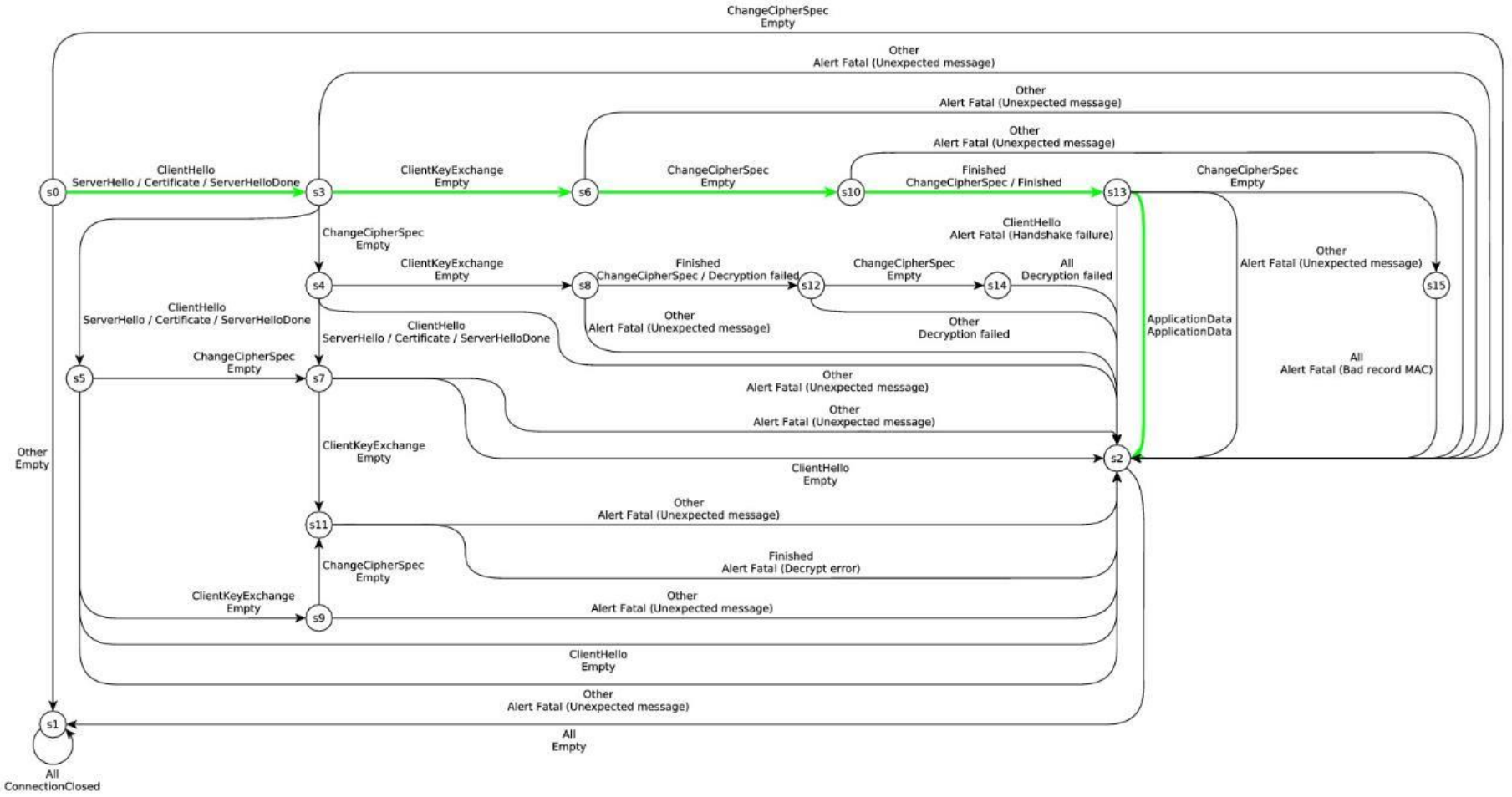
Would you trust this to be secure?

Active learning of TLS

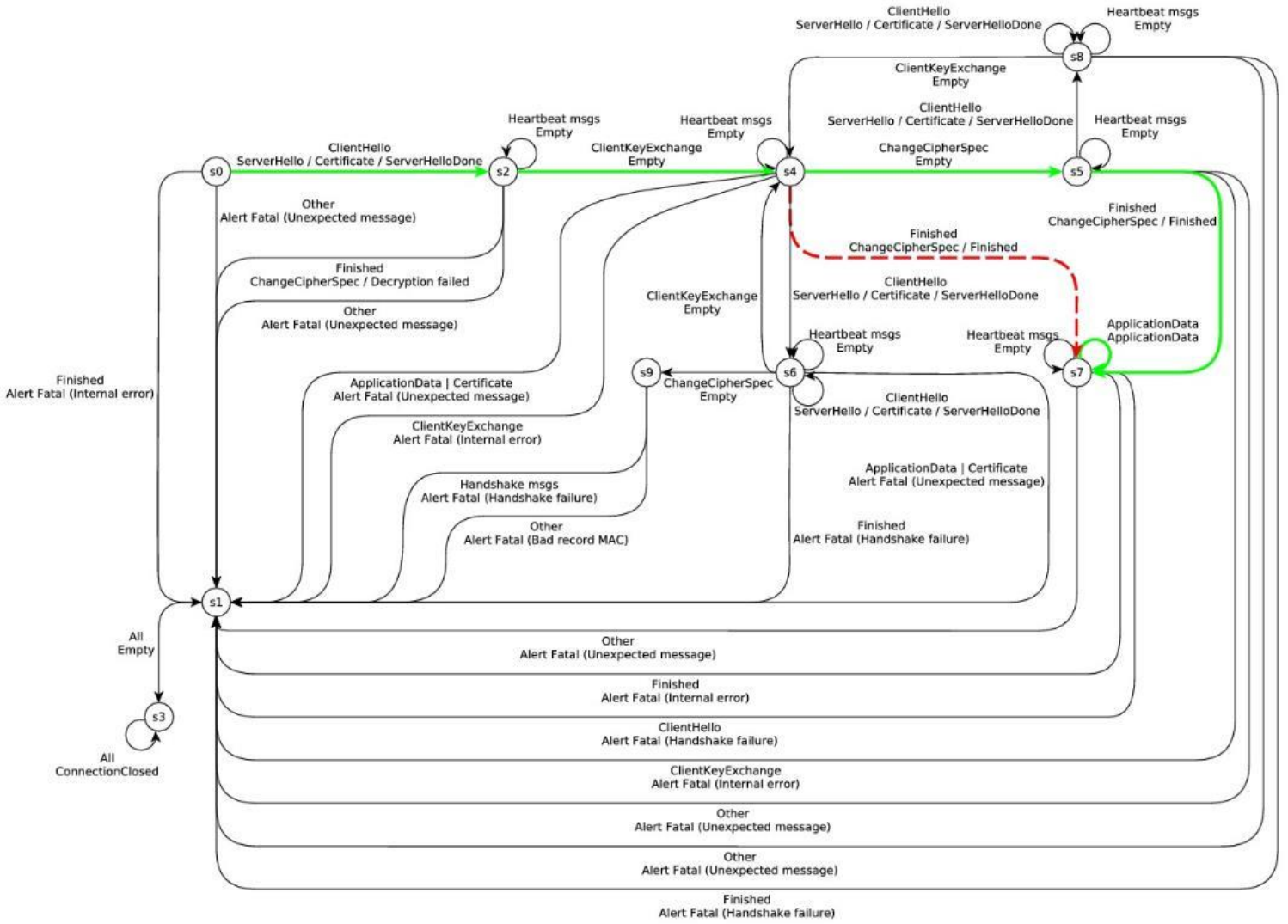


Protocol state machine of the NSS TLS implementation

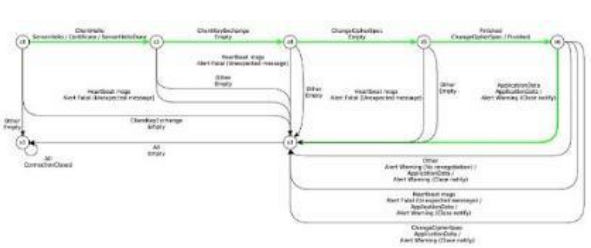
State machine of OpenSSL

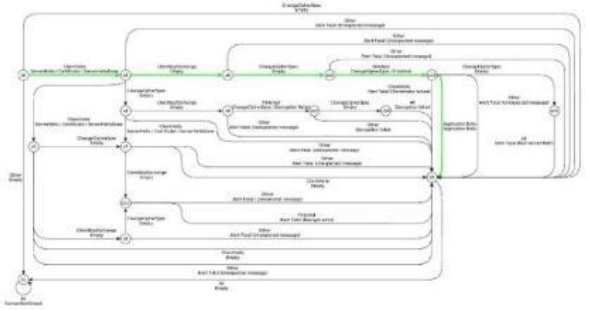


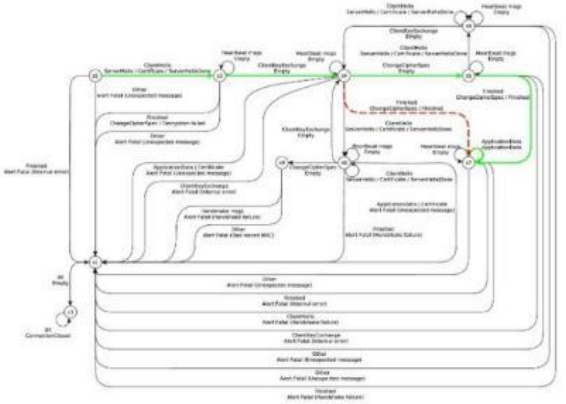
State machine of Java Secure Socket Exchange

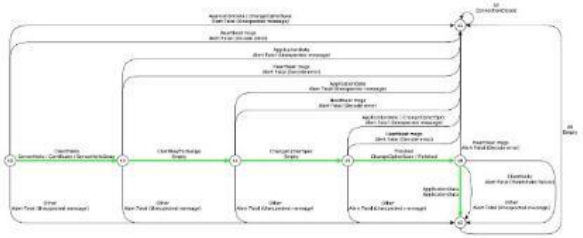


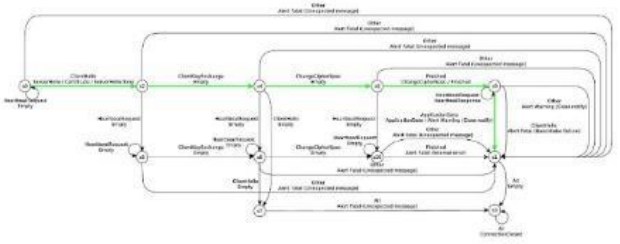
Active learning of TLS

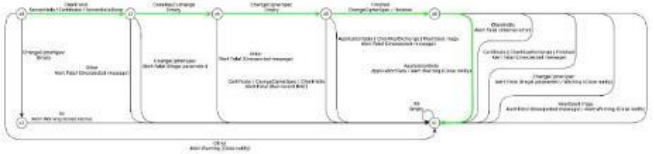


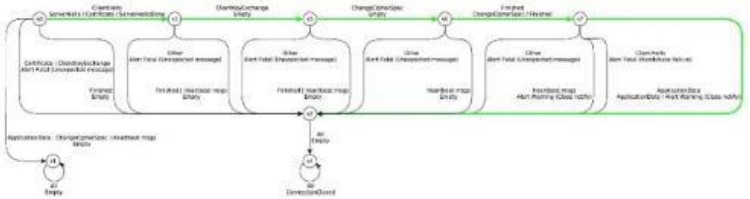


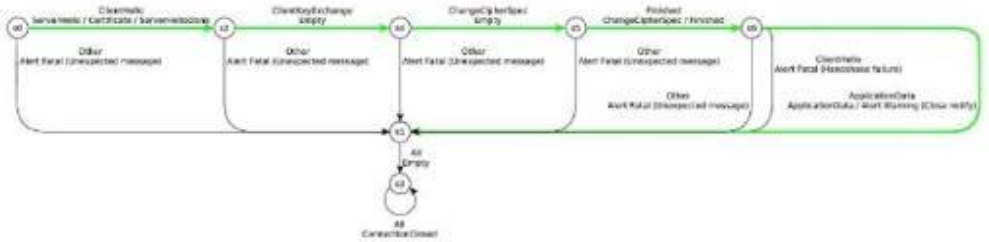












All implementations we analysed are different!
Why doesn't the TLS spec include a state machine?

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


NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES
USA

Forwarding flaws

[LangSec 2018]

[Strings considered harmful, Usenix ;login: , to appear]

(At least) two types of INPUT problems

1. Buggy processing & parsing

   - 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*

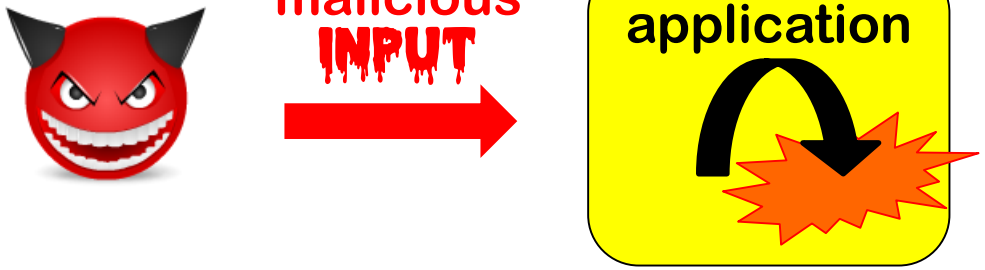
2. Flawed forwarding (aka injection attacks)

   - Input is forwarded to *back-end* service/system/API, to cause damage there
   - Classic examples: SQL injection, XSS, format string attack, Word macros

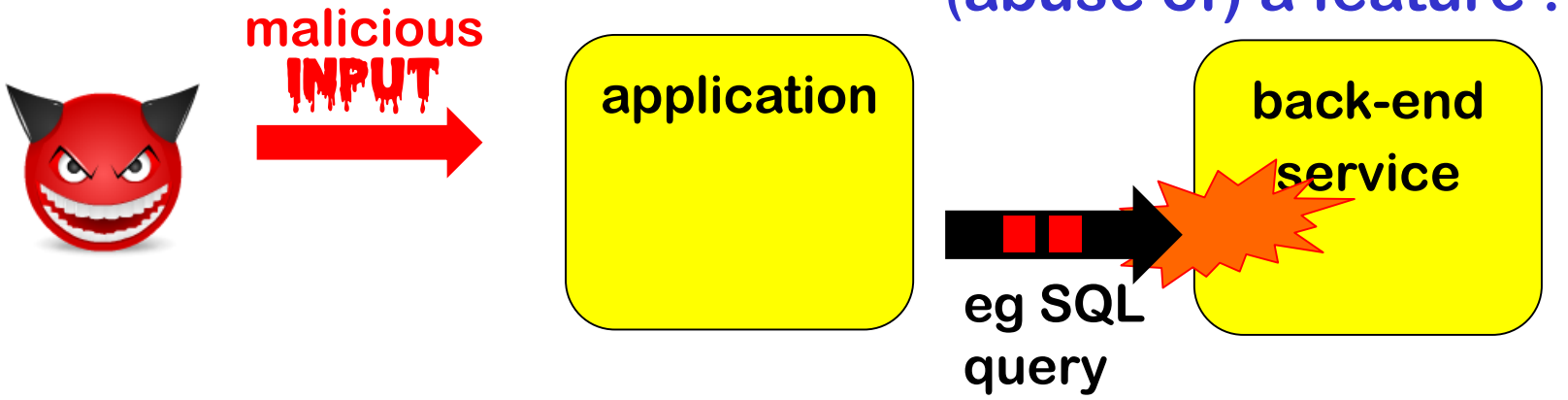
   This is *intended* behaviour of the back-end, introduced *deliberately*, but *exposed by mistake* by the front-end

Processing vs Forwarding Flaws

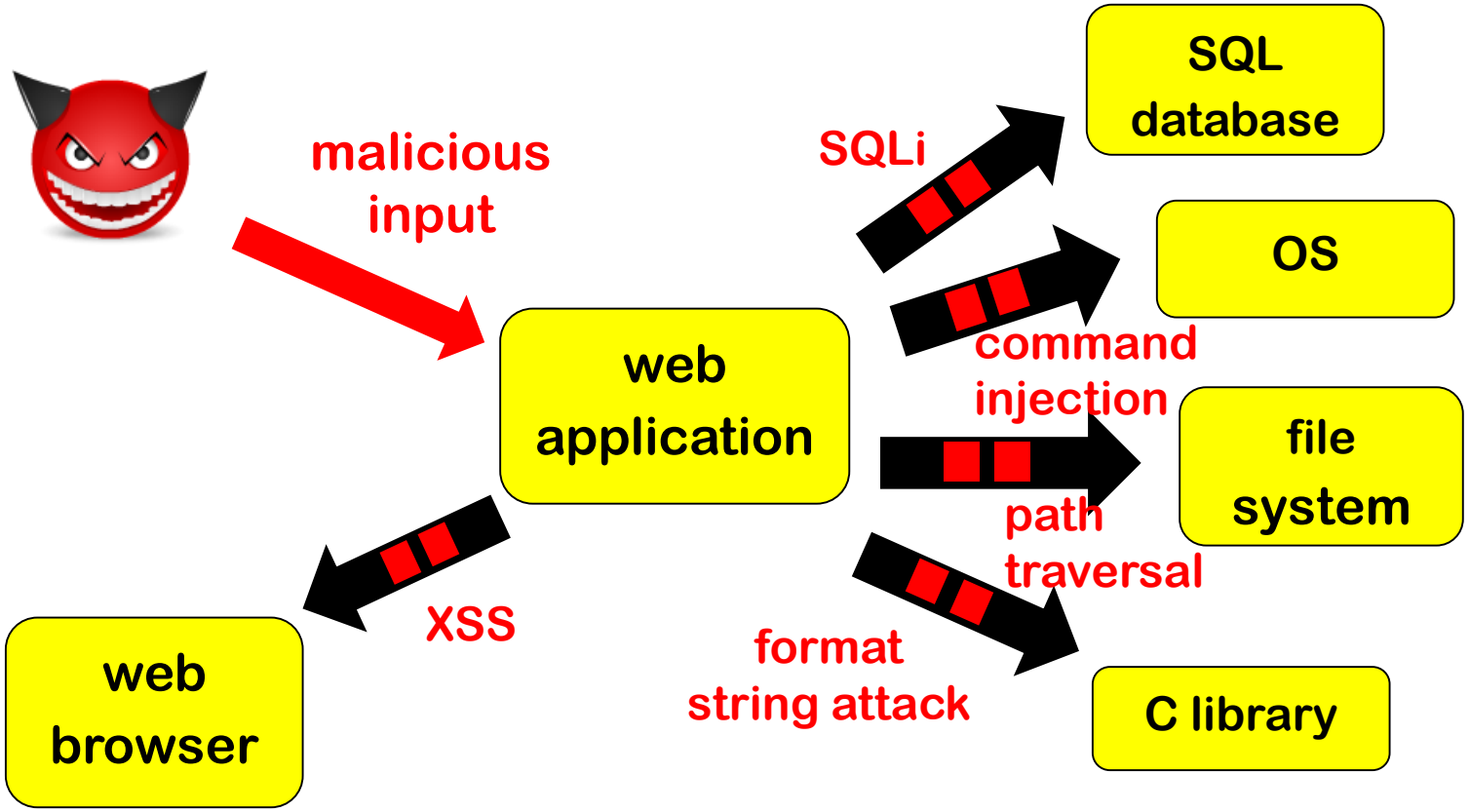
Processing Flaws



Forwarding Flaws



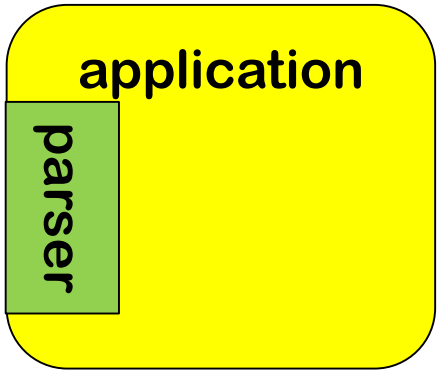
More back-ends, more languages, more problems



How & where to tackle input problems?

Tackling processing flaws

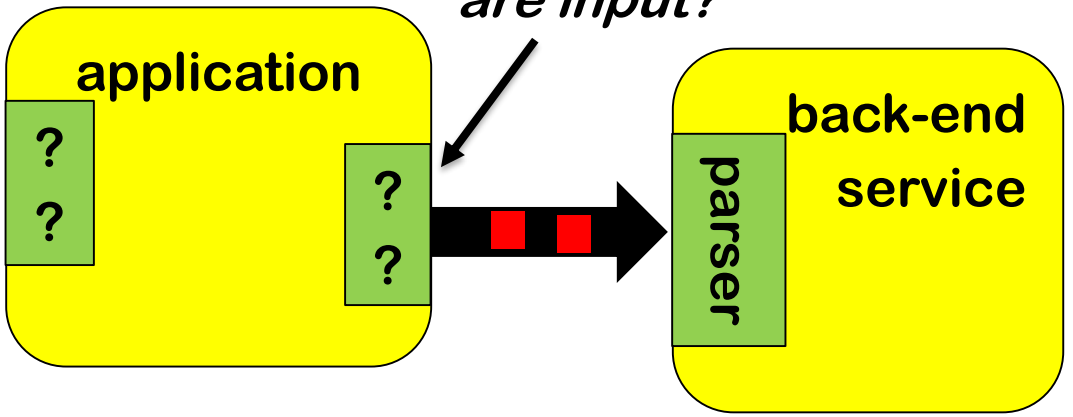
malicious input



LangSec approach:
Simple & clear language spec;
generated parser code;
complete parsing before
any further processing
(no shotgun parsing)

Tackling forwarding flaws?

malicious input



Which bits are input?

Where will this input end up?

validation/sanitisation:
filtering and/or escaping?

Anti-patterns
in tackling forwarding flaws

Anti-pattern: STRING CONCATENATION

- Standard recipe for security disaster: concatenating several pieces of data, some of them user input, and passing the result on to some API
  - Classic example: SQL injection
- Note: string concatenation is inverse of parsing

Avoiding SQL injection with prepared statement

Instead of a raw string as single input (aka dynamic SQL)

```
"SELECT * FROM Account WHERE Username = " + $username
                          + "AND Password = " + $password;
```

give a string with placeholders and the parameters as separate inputs

```
"SELECT * FROM Account WHERE Username = ? AND Password = ?" ,
 $username ,
 $password
```

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)**
     - **If you have a shotgun parser, your code will use strings**
  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**

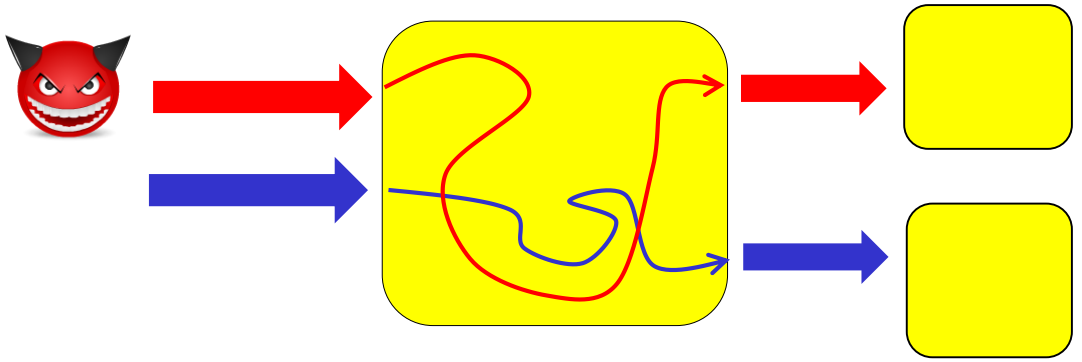
Remedies
to tackle forwarding flaws

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

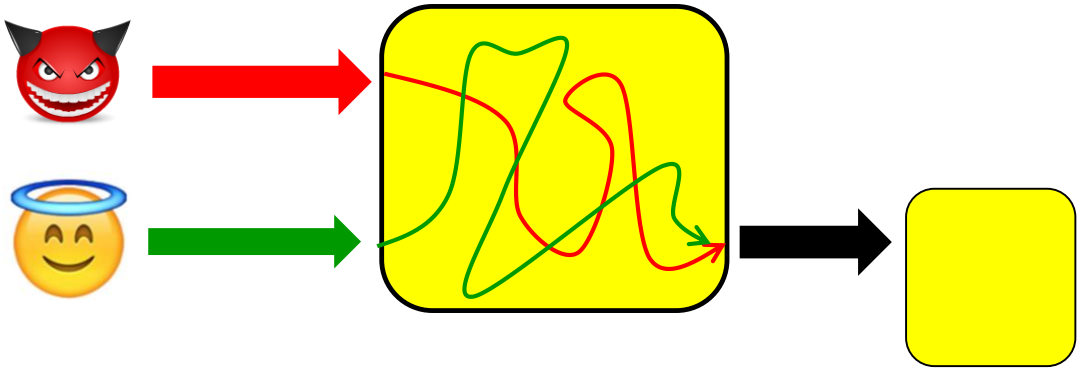


Remedy: Types (2) to distinguish *trust levels*

- 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

- **as latest attacks using script gadgets demonstrate**

  [Lekies et al., *Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets*, CCS'17]

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

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

Beyond types: extending programming language

Wyvern programming language by Jonathan Aldrich et al.
allows domain-specific extensions, eg

```
let authorName : String = user_input
 let webpage : HTML = ~
   <html>
     <body>
       <h1>Search results:</h1>
       <ul id="results">
         {query_results(db, ~)
           SELECT author, bookTitle FROM books
           WHERE author = {authorName}}
       </ul></body></html>
```

where HTML and SQL are 'built-in' types of the programming language

Added advantage over types: more convenient syntax

[D. Kurilova et al, *Wyvern: Impacting Software Security via Programming Language Design*, PLATEAU 2014, ACM]

Conclusions

Security is about software!

- Software plays *the* central role in cyber (in)security
- Hence: it's an important challenge to the software engineering community – incl. the ISoLA community - to improve software security

Conclusions

- 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
- Tools for test case generation can be very useful for security testing!
  - There's been an upsurge in interest in fuzzing over the past years

Thanks for your attention

Submit your papers to LangSec 2019!