Improving software security by improving input handling

Erik Poll

Digital Security

Radboud University Nijmegen

Radboud Universiteit Nijmegen

TRU/e Master in Cyber Security

Fighting INPUT problems

- Most security problems arise due to INPUT
- We can detect input problems with fuzzing
- We can fix input problems with input validation

  and input or output sanitisation
- Today: can we prevent input problems, in a more structural way, by construction?
  - by dealing with input handling in a methodological way

LangSec

(Language-theoretic Security)

LangSec (Language-Theoretic Security)

LangSec takes a systematic look at how to deal with input languages or formats to avoid typical input security problems

Root causes highlighted by LangSec community

1. **Applications have to handle data in *many* languages & formats**
2. **These languages are often *complex & unclearly defined and combined***
3. **The code handles all these languages & formats in sloppy way,**
   - **as the succes of fuzzing demonstrates**
   - **the prevalence of input attacks (path traversals, SQL injection, XSS, ...) shows**

Tower of Babel

A typical interaction on the web involves *__many__* languages & formats

HTTP(S), HTML5, CSS3, javascript, Flash, cookies & FSOs,

Ajax & XML, ActiveX, jpeg, mpeg, mp4, png, gif, SilverLight,

user names, email addresses, phone numbers,

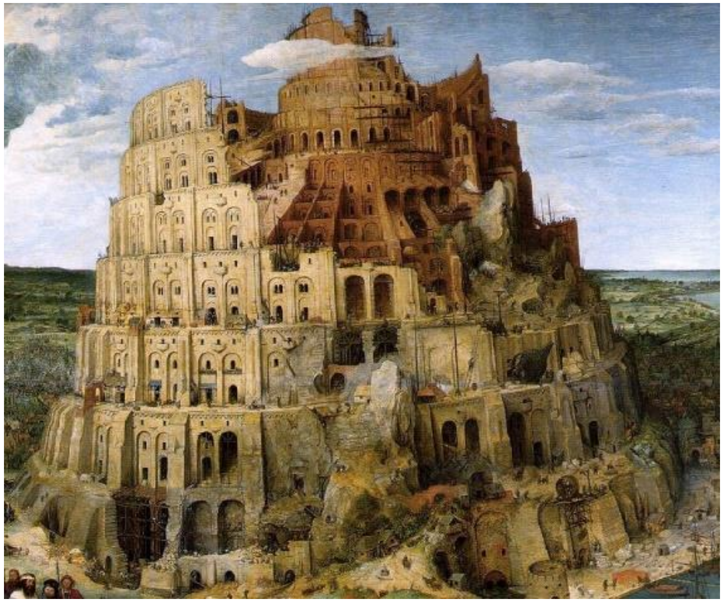
URLs, X509 certificates, TCP/IP (IPv4 or IPv6), DNS,

file names, directories, OS commands, SQL,

database commands, LDAP, JSP, PHP,

ASCII, Unicode, UTF-8, ...

**Some handled by web application & browser,
some others by lower protocol layers or
by external programs & services**



Input attacks on software

The common pattern in many attacks on software

> **buffer overflows, format string attacks, integer overflow, OS command injection, path traversal attacks, SQL injection, HTML injection, XSS, CSRF, database command injection, database function injection, PHP file name injection, LDAP injection, ..., ShellShock, HeartBleed,...**

is

1. **attacker crafts some *malicious input***
2. **software goes off the rails *processing* this:**
   - **Sometimes it simply crashes, and attacker can do DoS attack.**
   - **Sometimes, this exposes all sort of unintended functionality to attackers.**

Like social engineering or hypnosis as attack vector on humans?

Processing input

Processing involves

1) parsing/lexing

2) interpreting/executing

Eg *interpreting* a string as filename, URL, or email address,
or *executing* a piece of OS command, javascript, SQL statement

This relies on some language or format

Step 1) above relies on syntax of this language

Step 2) above relies also on semantics of this language

Processing input is dangerous!

Different ways for an attacker to abuse input

- *wasting resources* **(e.g. a zip-bomb)**
- *crashing things* **(and causing DoS)**
- *abusing strange functionality that is accidentily exposed*
  - **Existing functionality of say SQL database or the OS, or more bizarre functionality exposed by say a buffer overflow attack,**
  - **Buggy processing of inputs provides a weird machine that the attacker can "program" to abuse the system**

Garbage In, Garbage Out (GIGO)

becomes

Malicious **Garbage In, Security Incident Out**

Fallacy of classical input validation?

Classical input validation:

filter or encode harmful characters

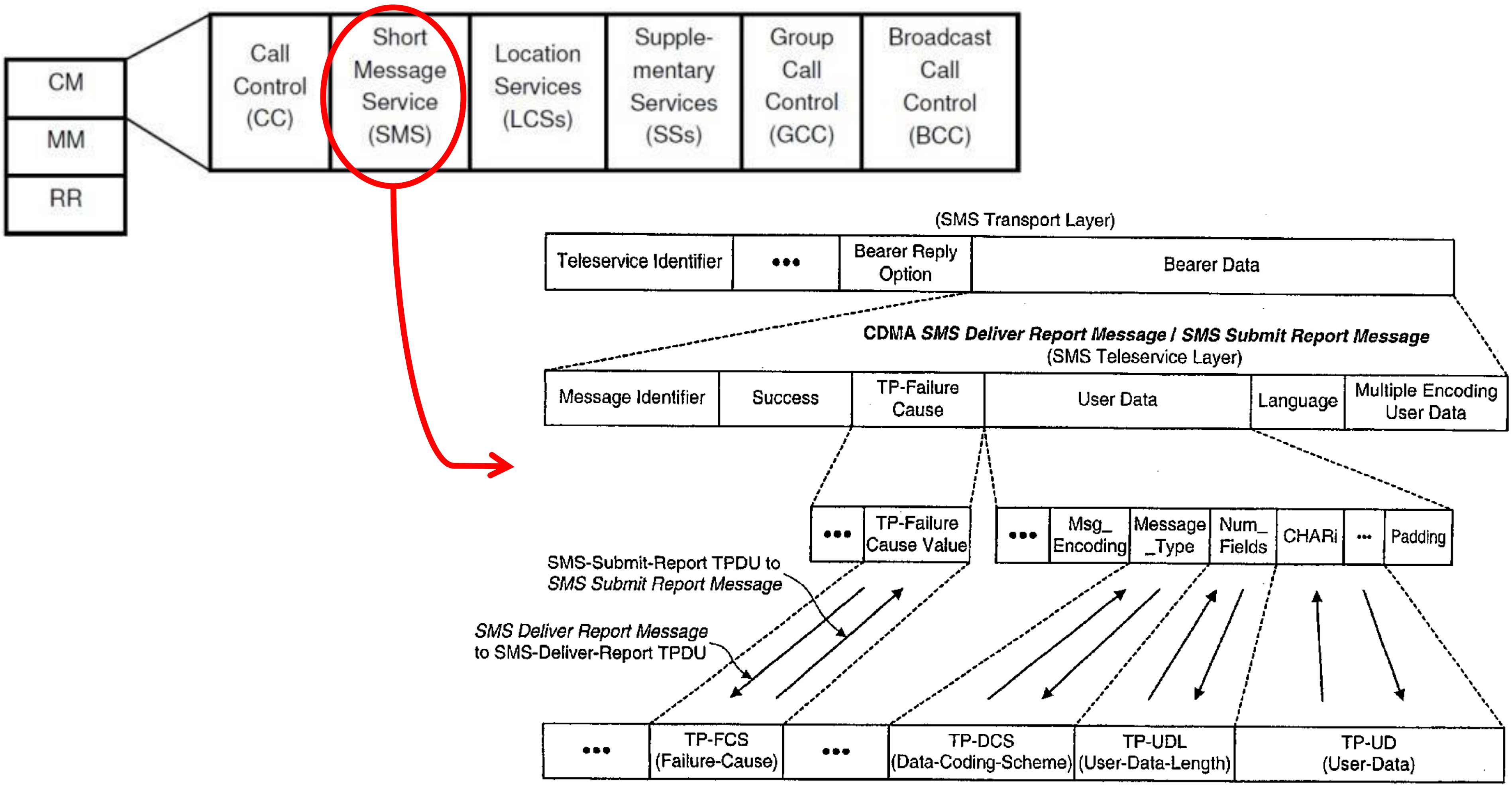
or, slightly better:

only let through harmless characters

But:

- Which characters are harmful (or required!) depends on the language or format. You need *context* to decide which characters are dangerous.
- Not only *presence* of funny characters can cause problems, but als the *absence* of other characters, or input fields that are too long or too short, ...

Recall: GSM as example complex input language



Sample problems (some already mentioned earlier)

- Code Red worm exploiting difference in size (in bytes) between between `char`'s and Unicode chararacters
- Exploits with zero-width fields in JPEG images
- Malformed Flash files exploiting flaws in Abode's flashplayer
- All the GSM problems revealed with fuzzing
- *Correctly formatted* NFC traffic crashing contactless payment terminals [MSc thesis Jordi van den Breekel, 2014]

Sample problems: Combining languages & formats

X509 certificates involve various languages & formats.
Differences in interpretation caused various security flaws:

- **ANS.1 attacks in X509 certificates**
  A null terminator in ANS.1 BER-encoded string in an `CommonName` can cause a CA to emit a certificate for an unauthorized Common Name.

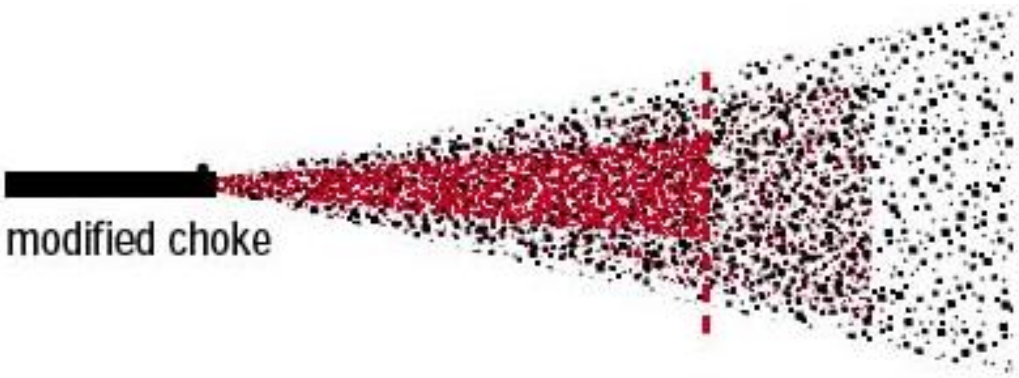
- **Multiple Common Names**
  allowed in X509, but handled diferently in different browsers

- **PKCS#10-tunneled SQL injection**
  SQL command *inside* a `BMPString`, `UTF8String` or `UniversalString` used as PKCS#10 `Subject Name`

[Dan Kaminsky, Meredith L. Patterson, and Len Sassaman,
PKI Layer Cake: New Collision Attacks Against the Global X.509 Infrastructure]

Anti-pattern: shotgun parsers

handwritten code that incrementally parses & interprets input, in a piecemeal fashion



An example shotgun parser

```
char buf1[MAX_SIZE], buf2[MAX_SIZE];

// make sure url is valid URL and fits in buf1 and buf2:

if (!isValid(url)) return;

if (strlen(url) > MAX_SIZE - 1) return;

// copy url up to first separator, ie. first '/', to buf1

out = buf1;

do { // skip spaces

   if (*url != ' ') *out++ = *url;

} while (*url++ != '/');

strcpy(buf2, buf1);

...
```

loop termination flaw (for URLs without /) caused Blaster worm

[Code sample from presentation by Jon Pincus]

Root causes/Anti-patterns

Obstacles in producing code without input vulnerabilities

1. **ad-hoc and imprecise notion of input validity**
2. **parser differentials**

   eg web-browsers parsing X509 certificates in different ways

3. **mixing input recognition & processing**

   aka shotgun parsers

4. **unchecked development of input languages**

   ie always adding new features & continuously evolving standards

All this results in weird machines, ie. systems that an attacker can "program" with malicious input

LangSec principles to prevent input problems

No more hand-coded shotgun parsers, but

1. *precisely defined* input languages

   eg with EBNF grammar

2. *generated* parser

3. *complete parsing before processing*

   So don't substitute strings & then parse,

   but parse & then substitute in parse tree

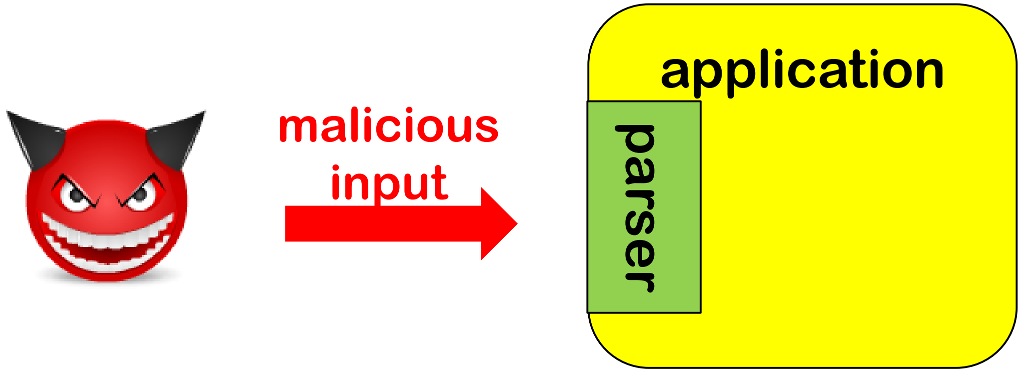
   (eg parameterised query instead of dynamic SQL)

4. *keep the input language simple & clear*

   So that equivalence of parsers is ideally decidable

   So that you give minimal processing power to attackers

Preventing input problems the LangSec way



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

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


FULL RECOGNITION
BEFORE PROCESSING

Weird machine = the strange functionality accidentality exposed by code that (incorrectly) processing input

Attackers can program this weird machine with their malicious input!

Minimise the resources & computing power that input handling gives to attackers


COMPUTATIONAL
EQUIVALENCE
FOR ALL
PROTOCOL
ENDPOINTS

No more length fields?

Proponents of LangSec argue against using length fields in data formats

- Length fields are a common source of trouble
  - incorrect length fields often cause buffer overflows
- They also make acceptor equivalence undecidable
  - because the resulting language is no longer regular or context-free

NB possible confusion in terminology

- Language-*based* security

  Providing safety/security features at programming language level

  Eg memory-safety, type-safety, thread-safety, sandboxing,…

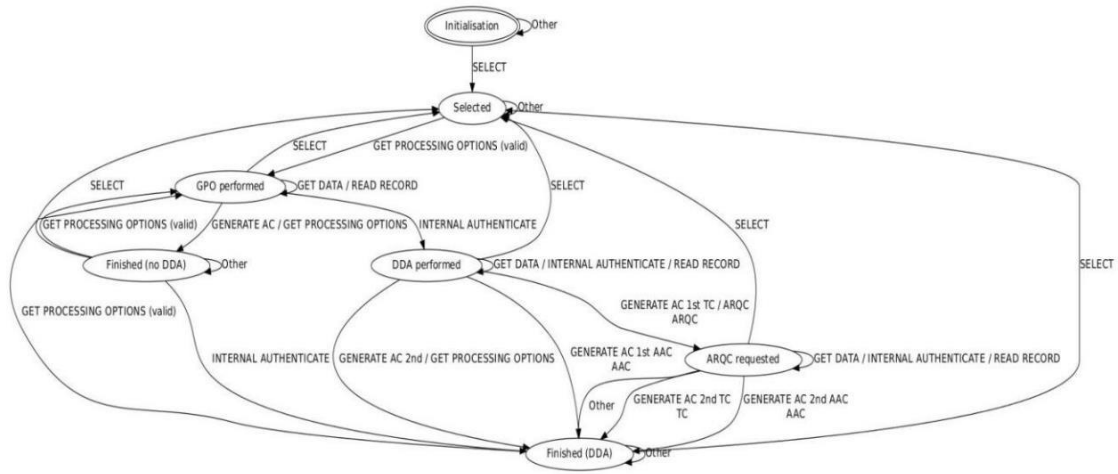
  Making programming less error-prone

  Here language = *programming* language

- Language-*theoretic* security (LangSec)

  Making handling input less error-prone

  Here language = *input* language



State Machines & State Machine Learning

(another level at which input handling goes wrong)

(another from of fuzzing)

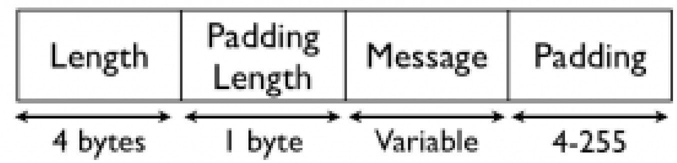
To read: Protocol state machines and session languages, LangSec'15

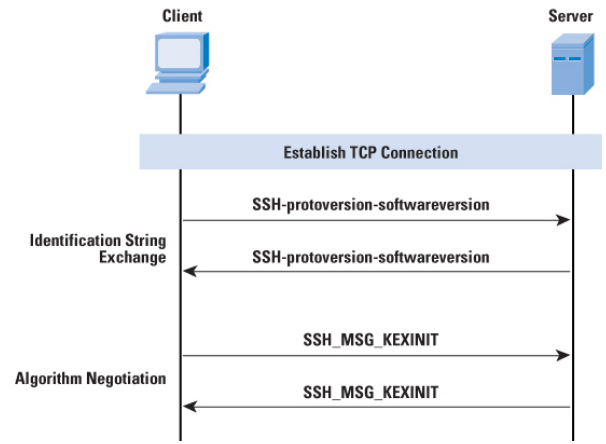
Protocols

Many procotols involve two levels of languages

1) a language of input messages
   or packets

2) a notion of session,
   or *sequence* of messages





How can we develop code for the two levels in a systematic way?

How can we test or fuzz these two levels?

For level 1 we can use fuzzing techniques discussed earlier

For level 2 we can do something different, as we discuss now

Message Sequence Charts (MSCs)

1.	$C \rightarrow S$: CONNECT	protocol identification
2.	$S \rightarrow C$: VERSION_S server version string	
3.	$C \rightarrow S$: VERSION_C client version string	
4.	$S \rightarrow C$: SSH_MSG_KEXINIT I_C	key exchange algorithm negotiation
5.	$C \rightarrow S$: SSH_MSG_KEXINIT I_S	
6.	$C \rightarrow S$: SSH_MSG_KEXDH_INIT e where $e = g^x$ for some client nonce x	key exchange
7.	$S \rightarrow C$: SSH_MSG_KEXDH_REPLY $K_S, f, sign_{K_S}(H)$ where $f = g^y$ for some server nonce y, $K = e^y$ and $H = hash(V_C, V_S, I_C, I_S, K_S, e, f, K)$, K_S is the server key	
8.	$S \rightarrow C$: SSH_MSG_NEWKEYS	
9.	$C \rightarrow S$: SSH_MSG_NEWKEYS	
10.	…	session, incl. SSH authentication and connection protocols

Typical spec given as Message Sequence Chart or in Alice-Bob style.

NB this *oversimplifies* because it only specifies *one correct run*, *the so-called happy flow*

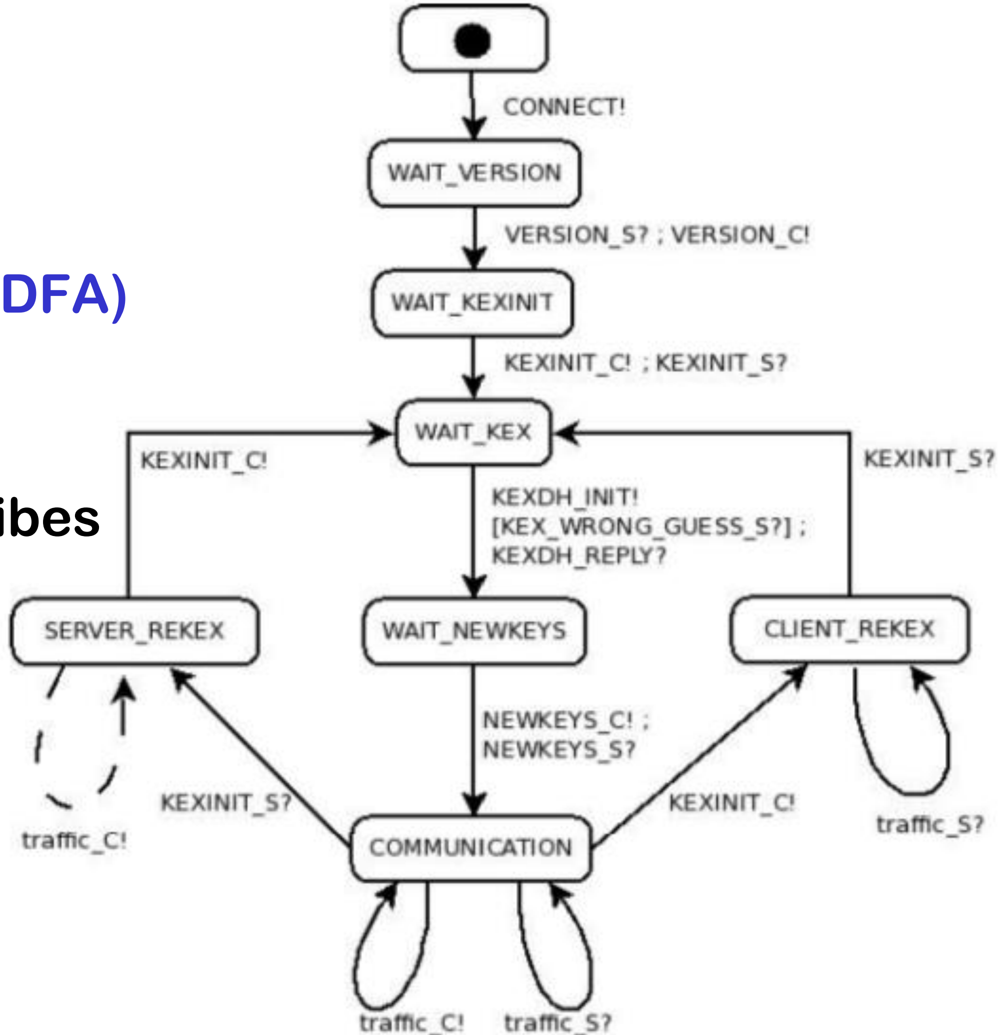
Protocol state machines

A protocol is typically more complicated than a simple sequential flow.

A better spec can be given using a **Finite State Machine (FSM)** aka **Deterministic Finite Automaton (DFA)**

This still oversimplifies: it only describes happy flows, albeit several ones.

The implementation will have to be **input-enabled**



SSH transport layer

input enabled state machines

A state machine is input enabled iff

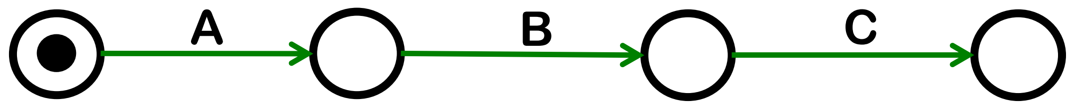
in *every* state

it is able to receive *every* message

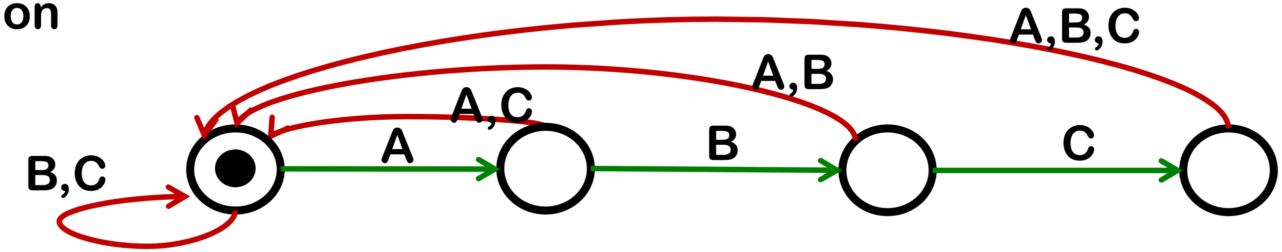
Often, many messages go to 1) some error state, 2) back to the initial state, or 3) are ignored

input enabling

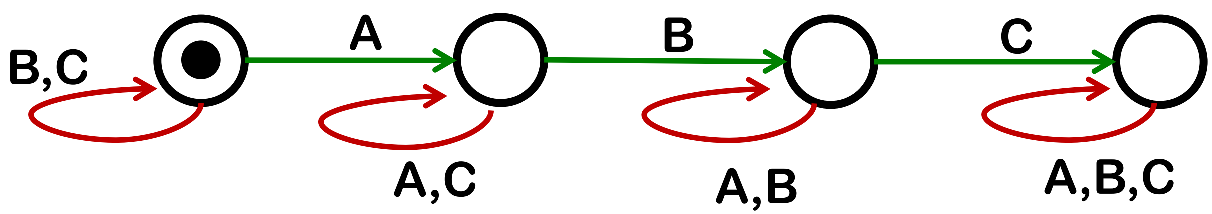
State machine that is not input-enabled



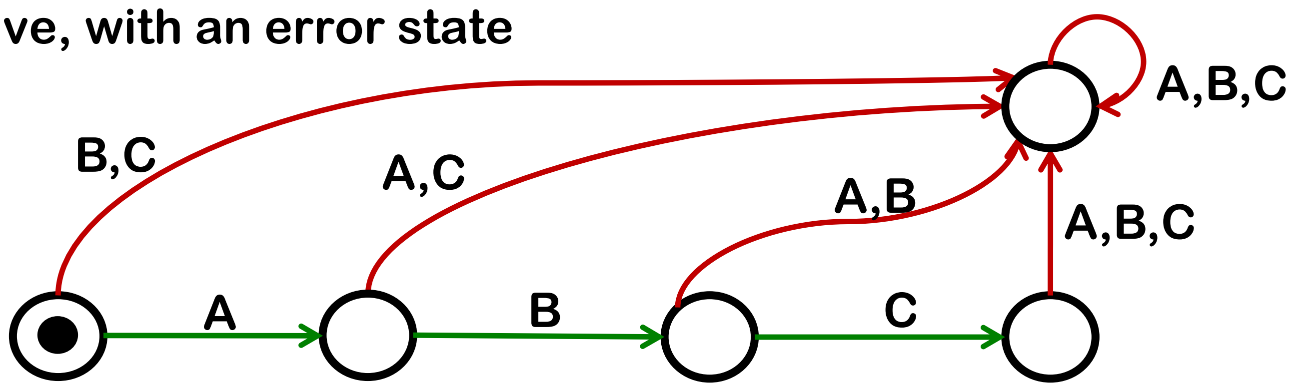
Input enabled version



Alternative input enabled version



Yet another alternative, with an error state



Typical prose specifications: SSH ☹

"Once a party has sent a SSH_MSG_KEXINIT message for key exchange or re-exchange, until it has sent a SSH_MSG_NEWKEYS message, it MUST NOT send any messages other than:

- **Transport layer generic messages (1 to 19) (but SSH_MSG_ SERVICE REQUEST and SSH_MSG_SERVICE_ACCEPT MUST NOT be sent);**
- **Algorithm negotiation messages (20 to 29) (but further SSH_MSG KEXINIT messages MUST NOT be sent);**
- **Specific key exchange method messages (30 to 49).**

The provisions of Section 11 apply to unrecognised messages"

In Section 11:

> **"An implementation MUST respond to all unrecognised messages with an SSH_MSG_UNIMPLEMENTED. Such messages MUST be otherwise ignored. Later protocol versions may define other meanings for these message types."**

Understanding protocol state machine from prose is hard!

Typical prose specifications: EMV ☹☹

Excerpt of the EMV contactless specs

“If the card responds to GPO with SW1 SW2 = x9000 and AIP byte 2 bit 8 set to b0, and if the reader supports qVSDC and contactless VSDC, then if the Application Cryptogram (Tag '9F26') is present in the GPO response, then the reader shall process the transaction as qVSDC, and if Tag '9F26' is not present, then the reader shall process the transaction as VSDC.”

Example security flaws due to broken state machines

CVE-2018-10933 libssh

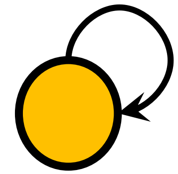
- **libssh versions 0.6 and above have an authentication bypass vulnerability in the server code. By presenting the server an SSH2_MSG_USERAUTH_*SUCCESS* message in place of the SSH2_MSG_USERAUTH_*REQUEST* message which the server would expect to initiate authentication, the attacker could successfully authentciate without any credentials.**

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

Example security flaws due to broken state machines

- **MIDPSSH**

  no state machine implemented at all



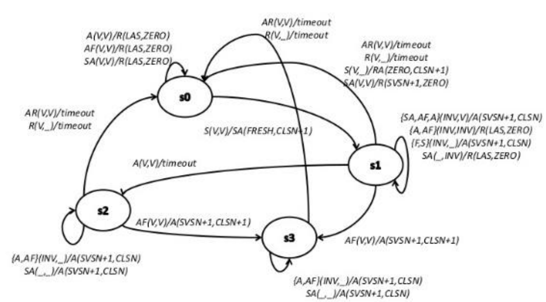
[Verifying an implementation of SSH, WIST 2007]

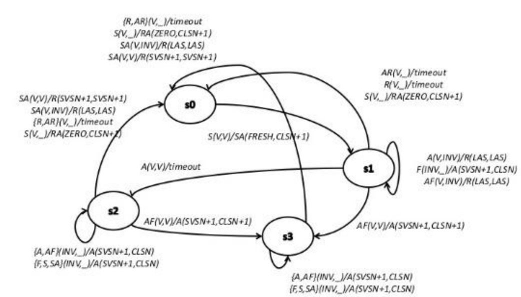
- **e.dentifier2**

  strange sequence of USB commands by-passes OK

[Designed to fail: a USB-connected reader for online banking , NordSec 2012

There can also be fingerprinting possibilities due to differences in implemented protocol state machines, eg in e-passports from different countries or in TCP implementations on Windows/Linux





Extracting protocol state machines from code

We can infer a finite state machine from implementation by black box testing using state machine inference

- using L* algorithm, as implemented in eg. LearnLib

This is effectively a form of 'stateful' fuzzing using a test harness that sends typical protocol messages.

It can also be regarded as a form of automated reverse engineering

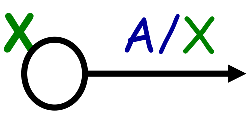
This is a great way to obtain protocol state machine

- without reading specs!
- without reading code!

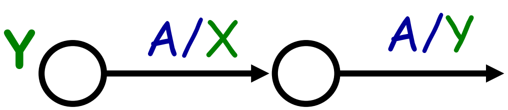
State machine inference, eg using LearnLib tool

Just try out many sequences of inputs, and observe outputs

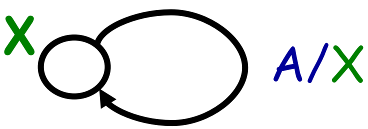
Suppose input A results in output X



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

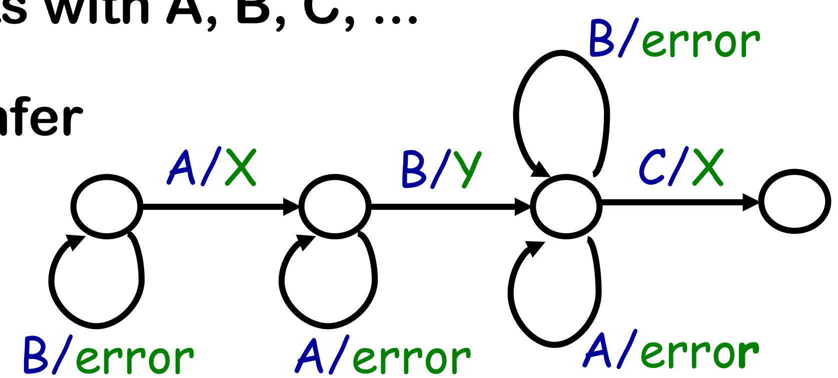


- **If second input A results in the *same* output X**



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

to e.g. infer



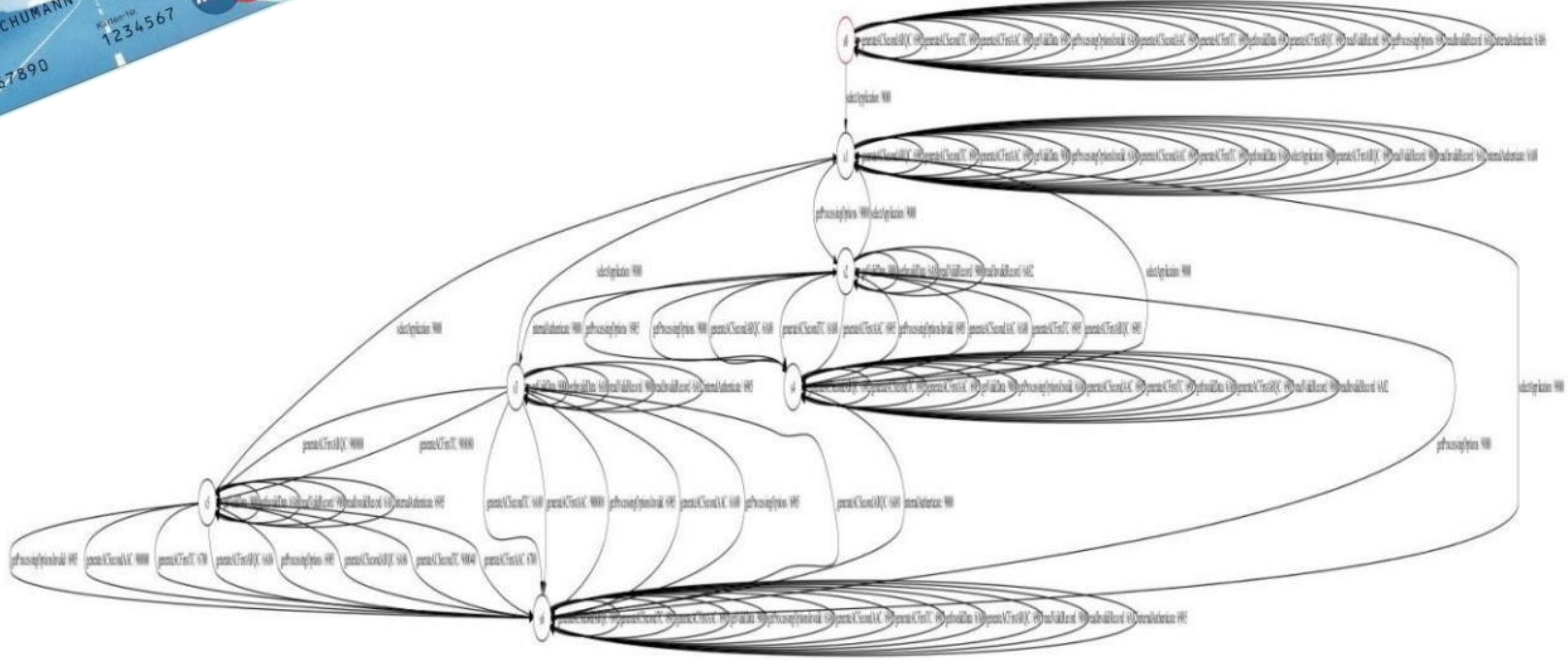
The inferred state machine is an under-approximation of real system

Case study 1: EMV

- **Most banking smartcards implement a variant of EMV**
- **EMV (Europay-Mastercard-Visa) defines set of protocols with *lots* of variants**

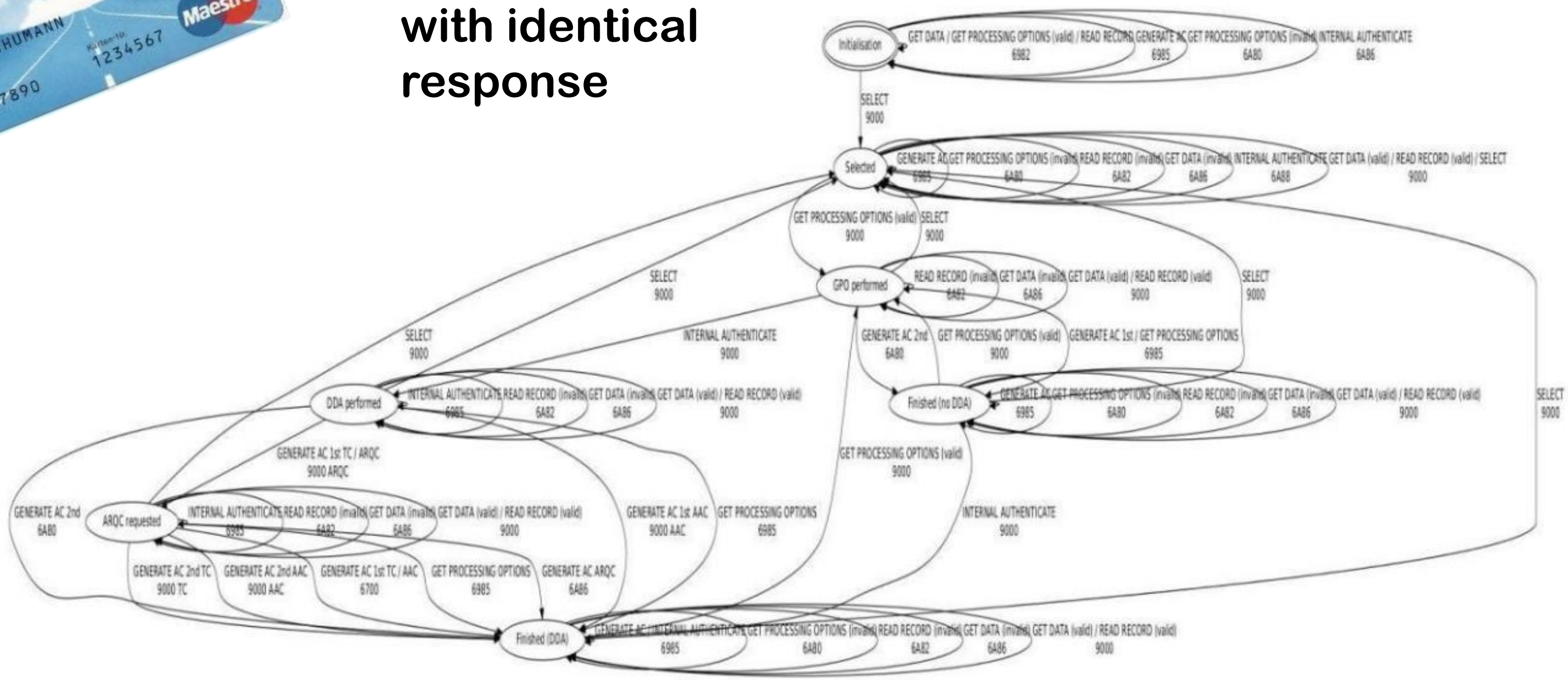
- **Specification in 4 books totalling > 700 pages**
- **EMV contactless specs: 10 more books, > 1500 pages**

State machine inference of Maestro card

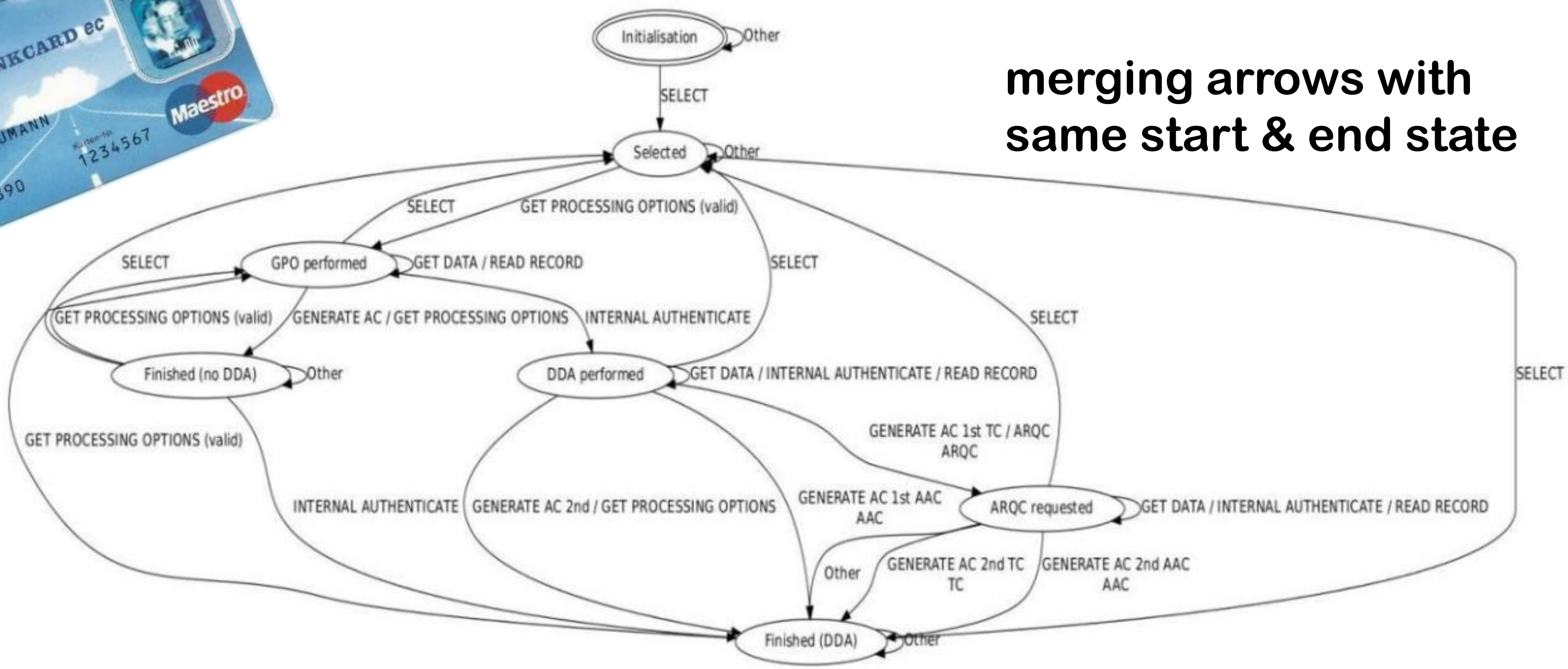


State machine inference of Maestro card

merging arrows with identical response



State machine inference of Maestro card



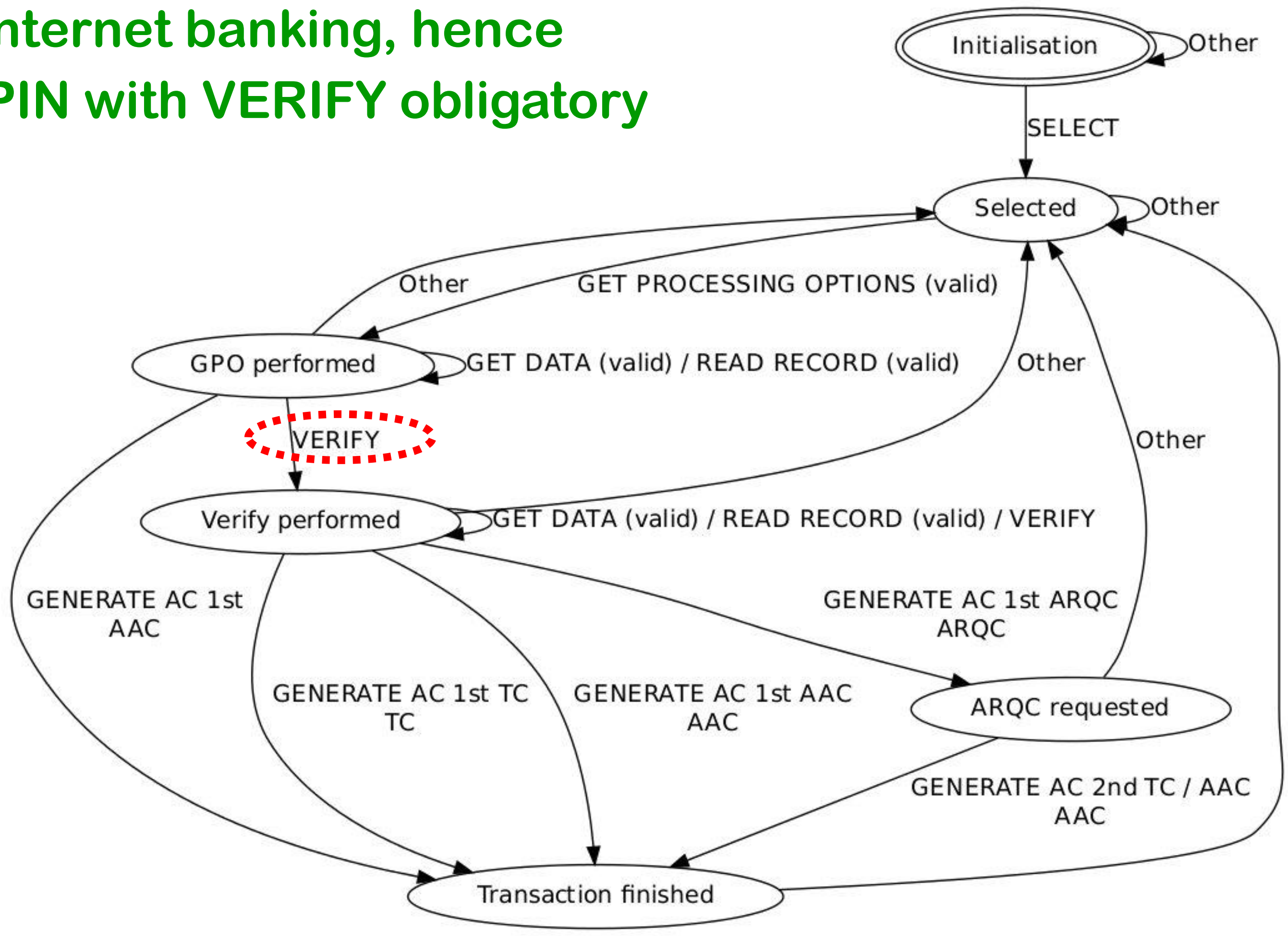
merging arrows with
same start & end state

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

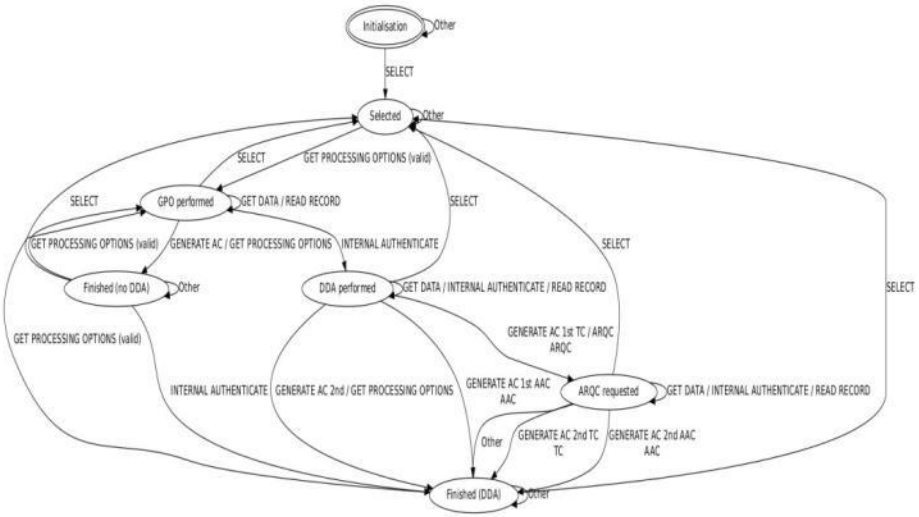
[Fides Aarts et al., Formal models of bank cards for free, SECTEST 2013]

SecureCode application on Rabobank card

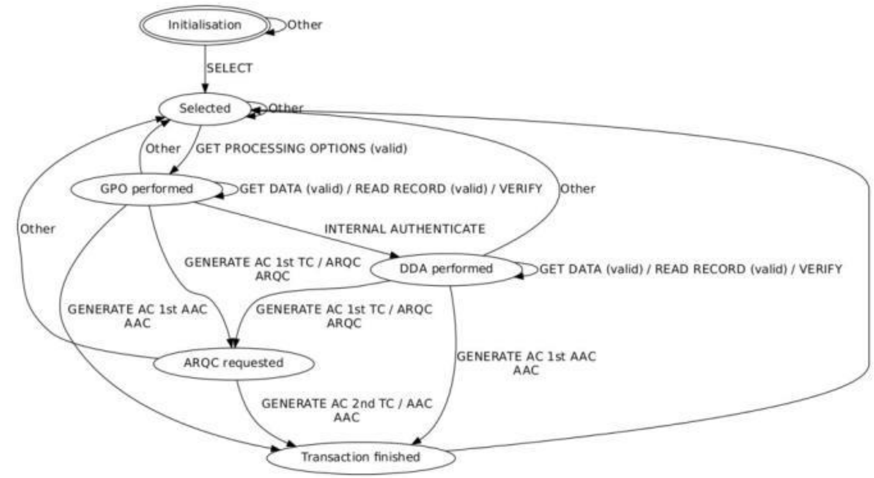
used for internet banking, hence entering PIN with VERIFY obligatory



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?

Using such protocol state diagrams

- Analysing the models by hand, or with model checker, for flaws
  - to see if *all paths* are correct & secure
- Fuzzing or model-based testing
  - using the diagram as basis for “deeper” fuzz testing
    - eg fuzzing also parameters of commands
- Program verification
  - *proving* that there is no functionality beyond that in the diagram, which using just testing you can never be sure of
- Using it when doing a manual code review

Case study 2: the USB-connected e.dentifier

Can we fuzz

- USB commands
- user actions via keyboard

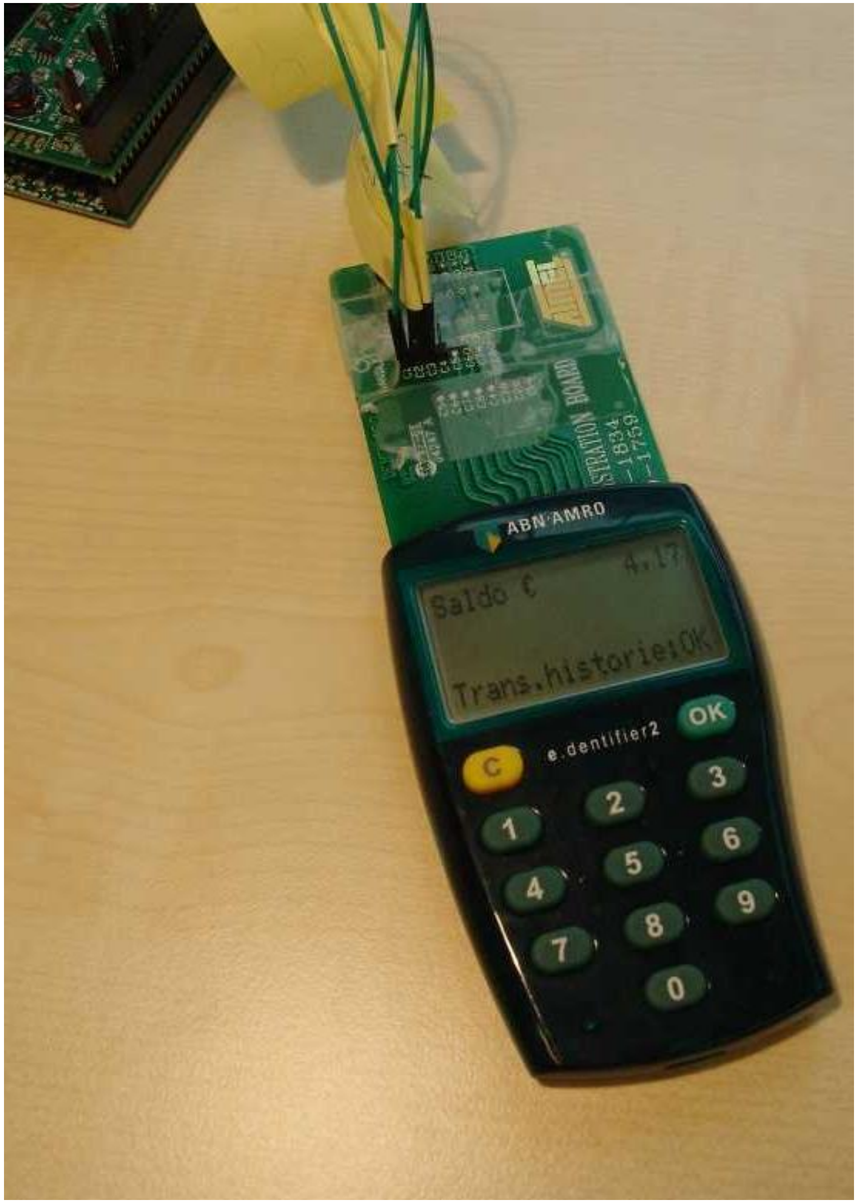
to automatically reverse engineer

the ABN-AMRO e.dentifier2?

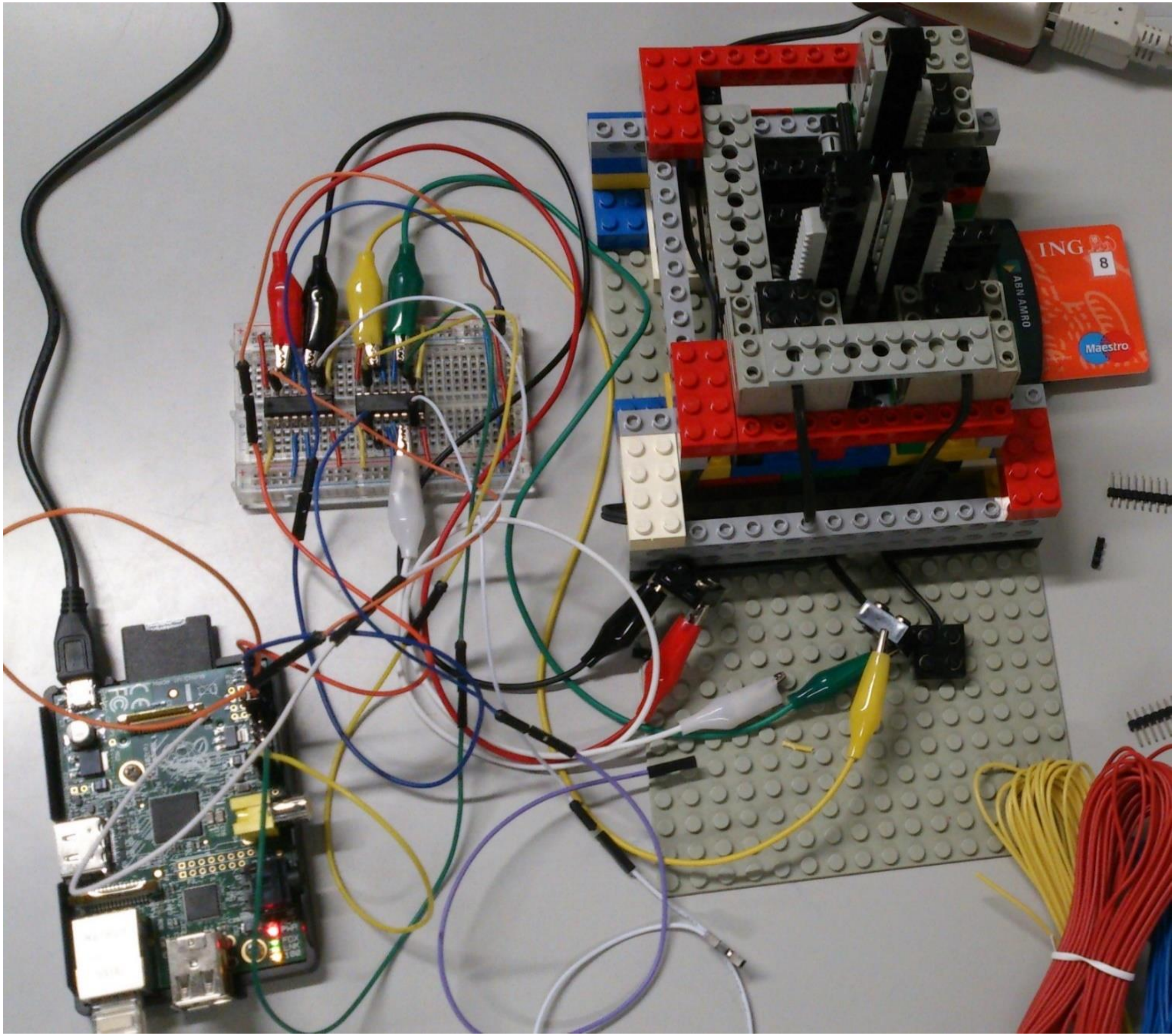
[Arjan Blom et al,

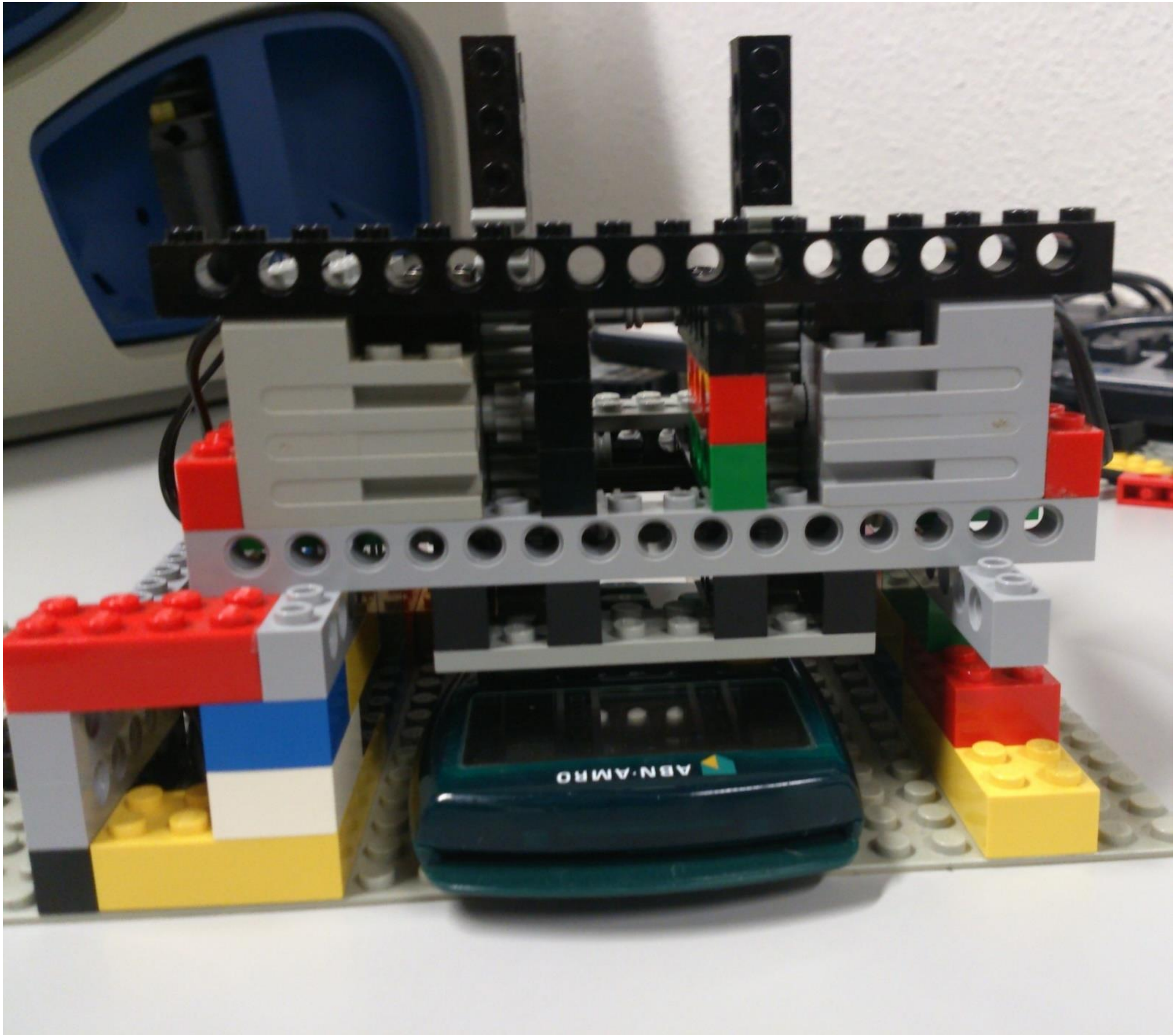
Designed to Fail: a USB-connected reader

for online banking, NORDSEC 2012]

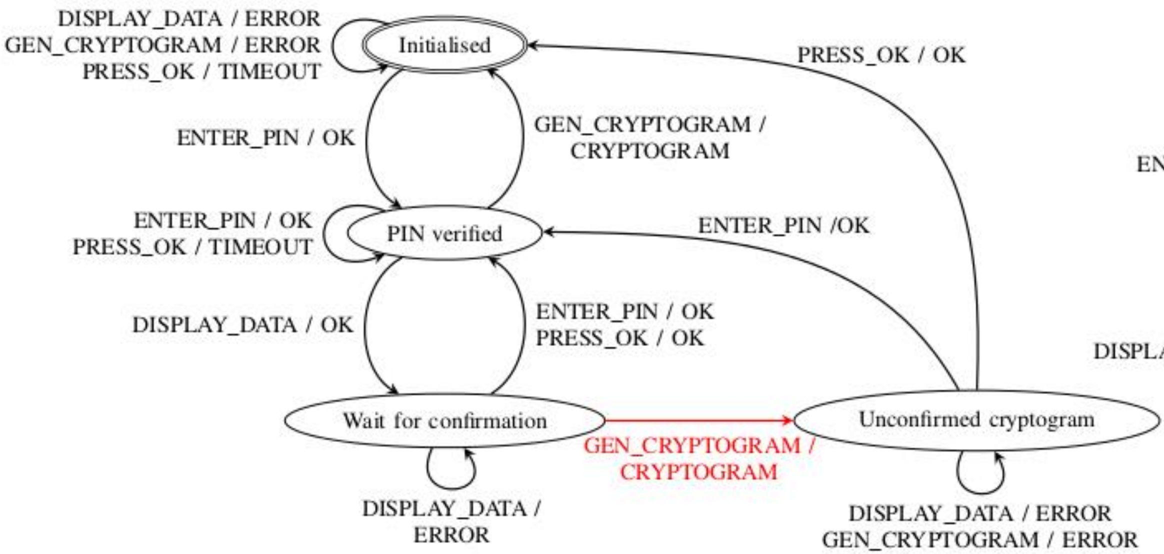


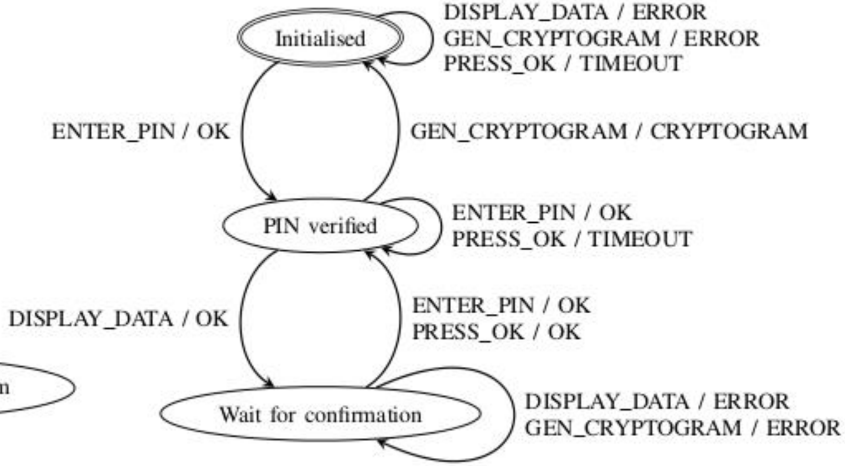
Operating the keyboard using LEGO

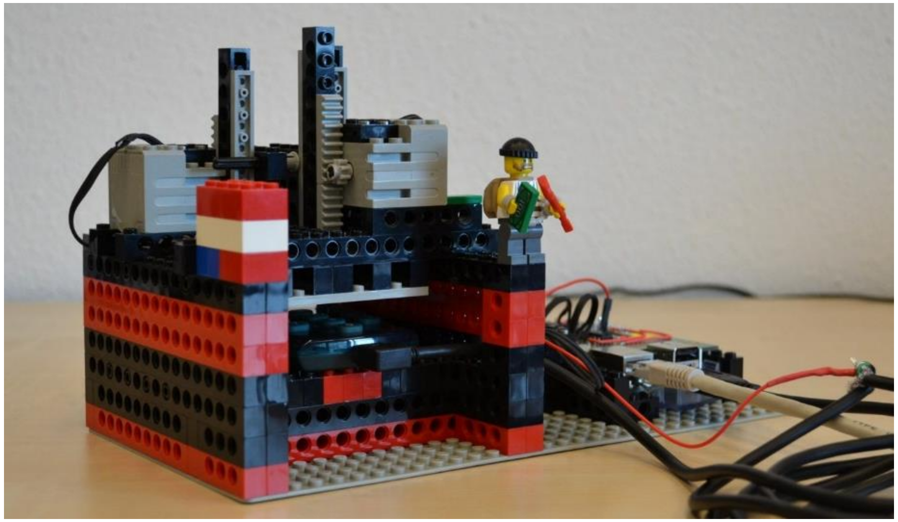

ING
8
ABN AMRO
Maestro


ABN·AMRO

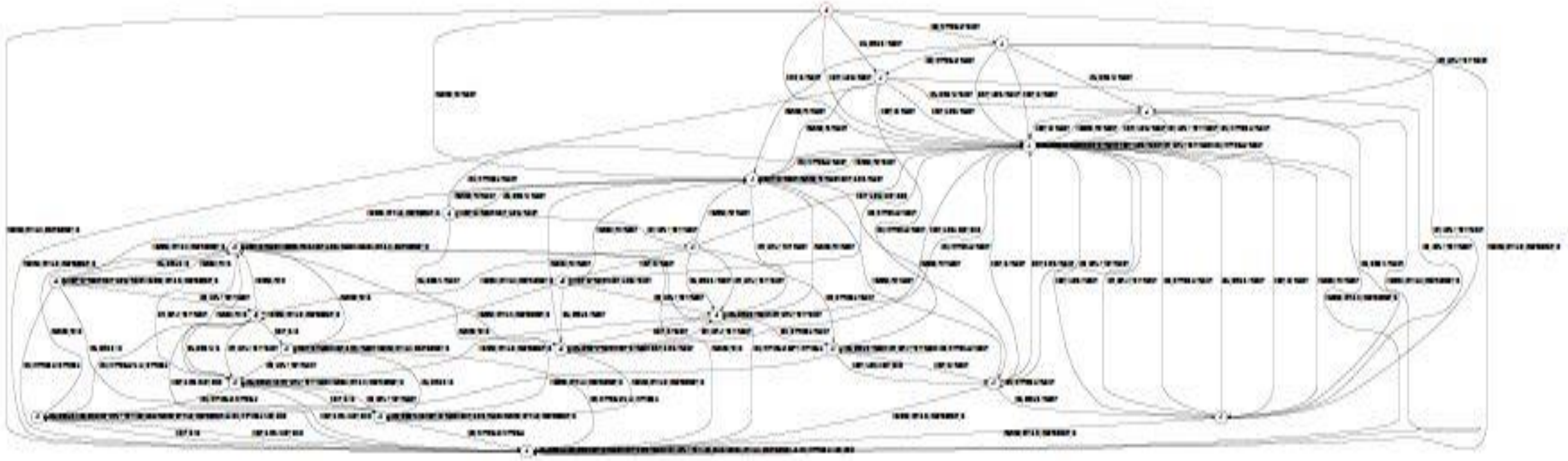
State machines of old vs new e.dentifier2







Would you trust this to be secure?



More detailed inferred state machine, using richer input alphabet.

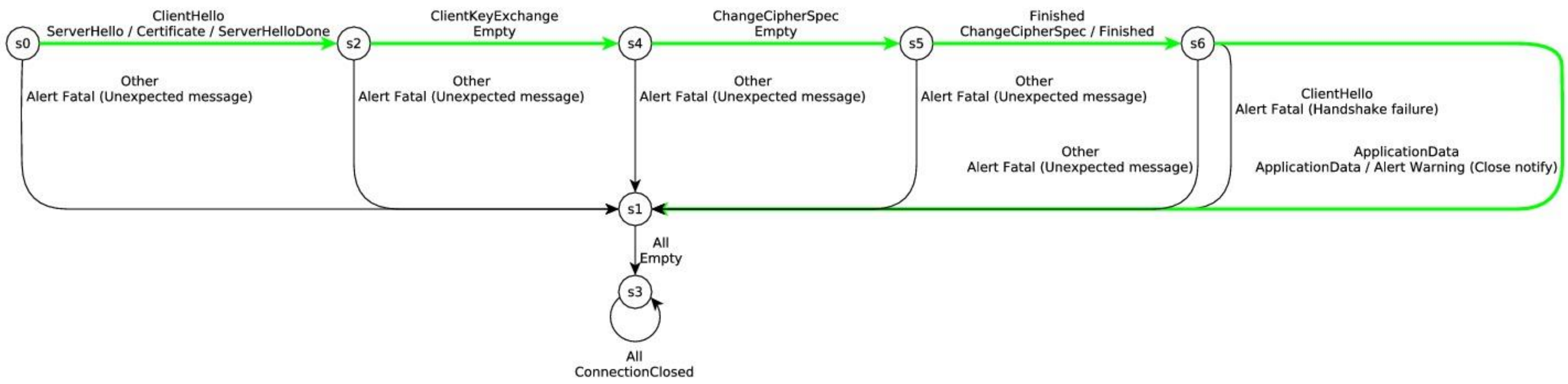
Do you think whoever designed or implemented this is confident that this is secure?
Or that all this behaviour is necessary?

Results with learning state machines for e.dentifier2

- Coarse models, with a limited alphabet, can be learnt in a few hours
  - these models are detailed enough to show presence of the known security flaw in the old e.dentifier, and absence of this flaw in the new one
- The most detailed models required 8 hours or more
- The complexity of the more detailed models suggest there was no clear protocol design that was used as the basis for the implementation

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

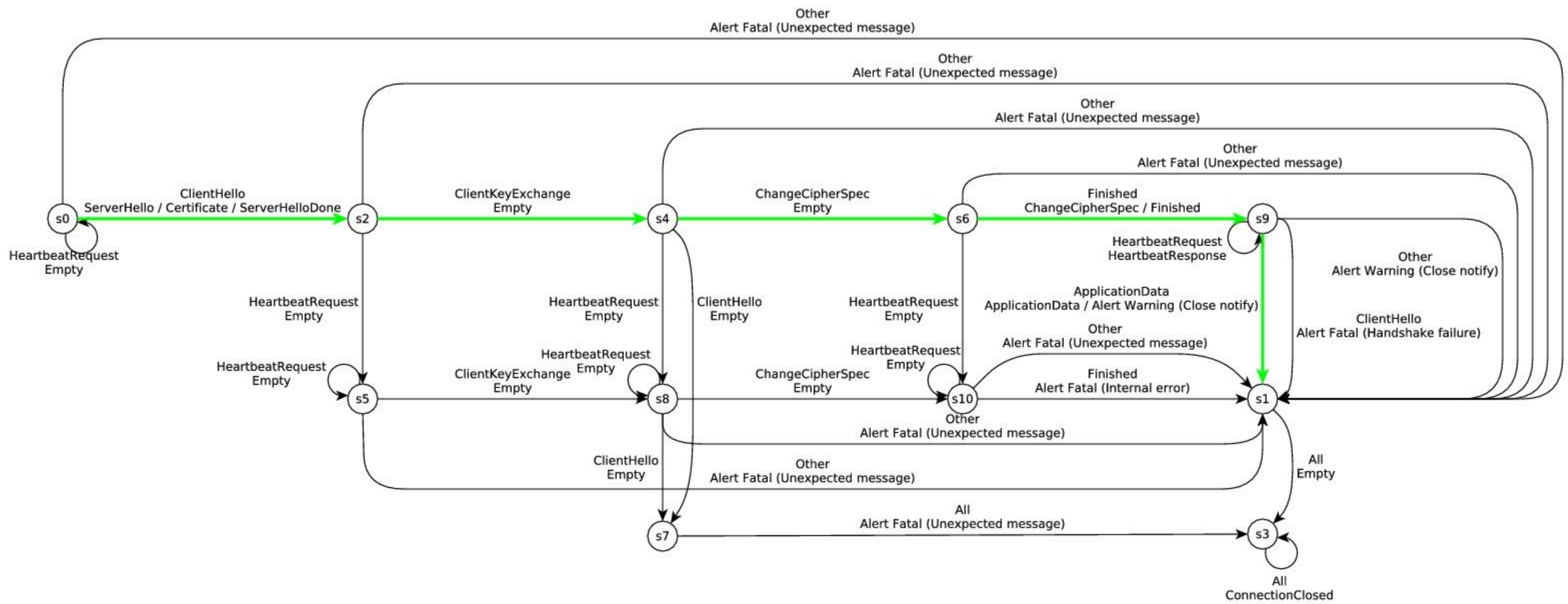
Case study 3: TLS



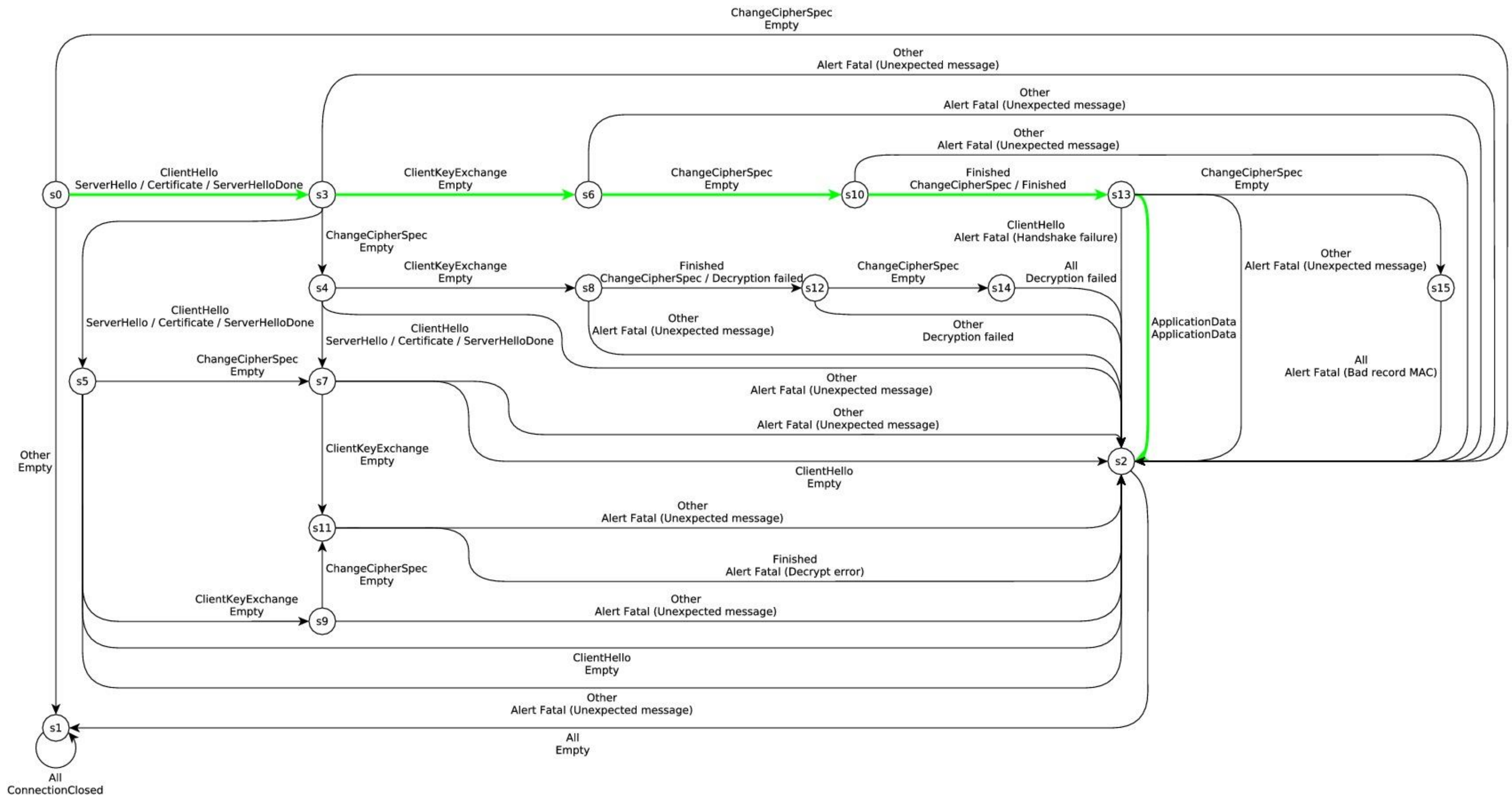
State machine inferred from NSS implementation

Comforting to see this is so simple!

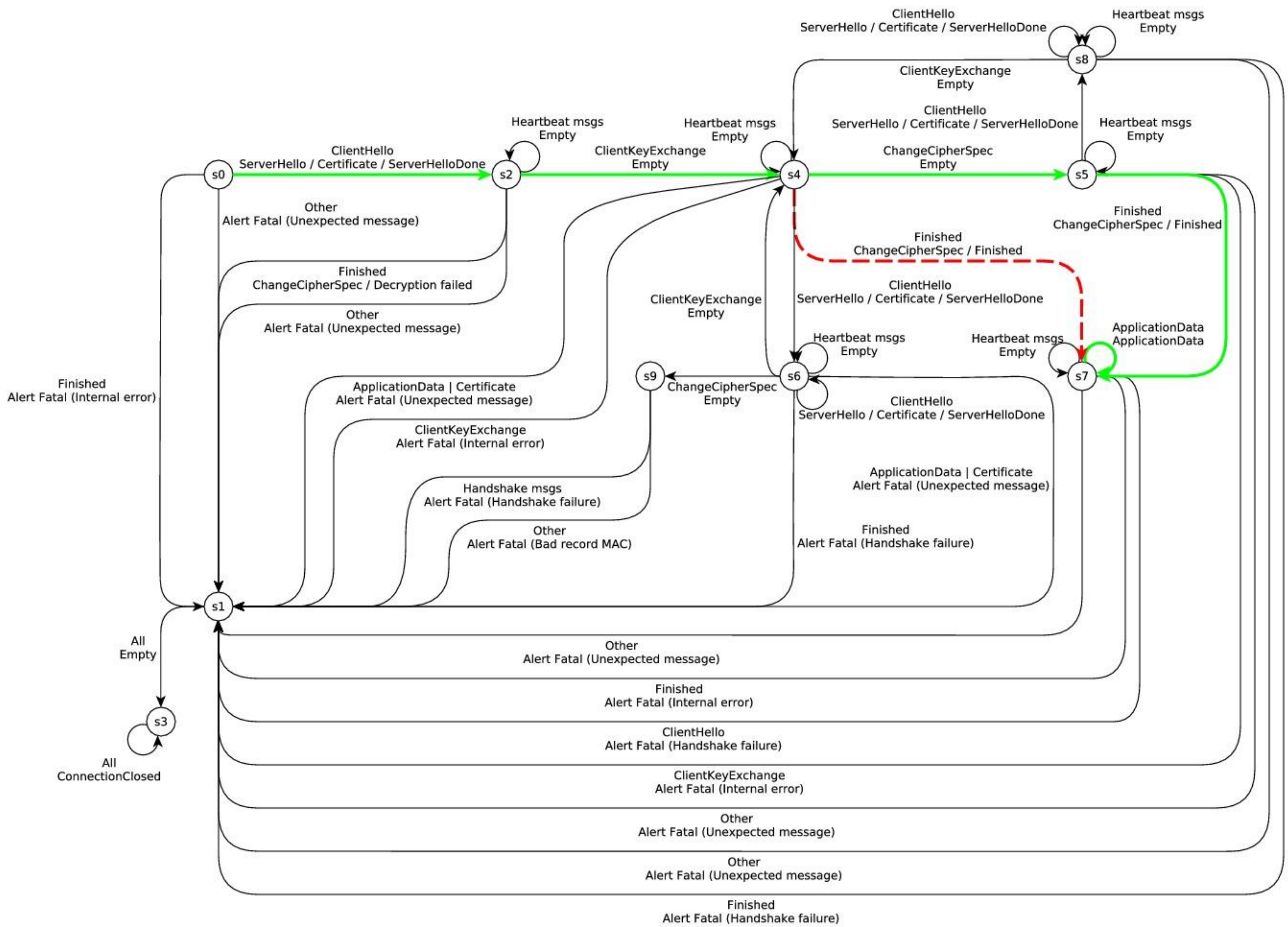
TLS... according to GnuTLS



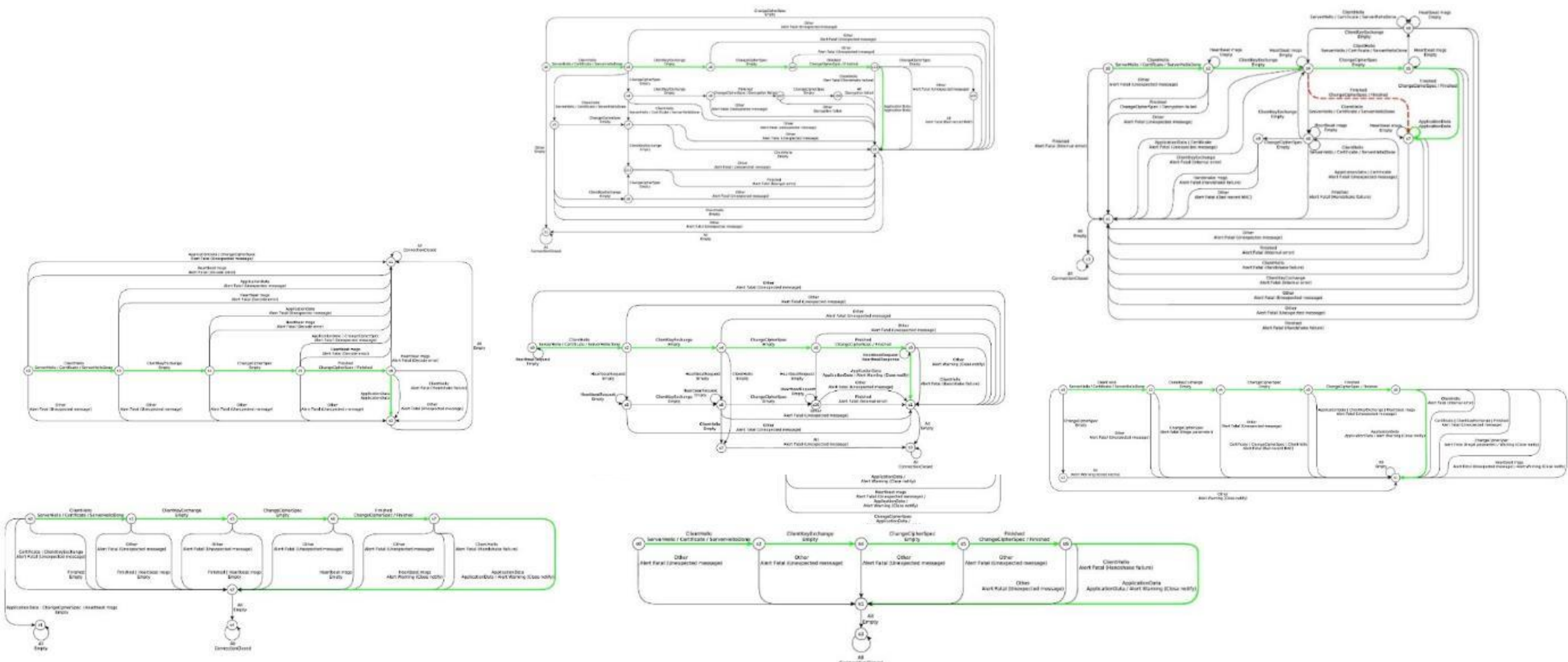
TLS... according to OpenSSL



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]

Results with learning state machines for TLS

- **Three new security flaws found**, in
  - OpenSSL
  - GnuTLS
  - Java Secure Socket Extension (JSSE)
- One (not security-critical) flaw found in newly proposed reference implementation nqbs-TLS
- For most TLS implementations, models can be learned within 1 hour

Conclusions: protocol state machines

Rigorous & clear specs using protocol state machines can improve security:

- by avoiding ambiguities
- useful for programmer
- useful for model-based testing

In the absence of state machines in specs, extracting state machines from code using state machine inference is great for

- security testing & analysis of implementations
- obtaining reference state machines for legacy systems
  - without having to read nasty RFCs or other specs

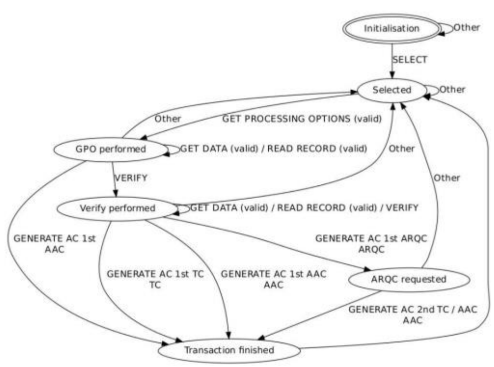
specs

implementing

code

model-based
testing

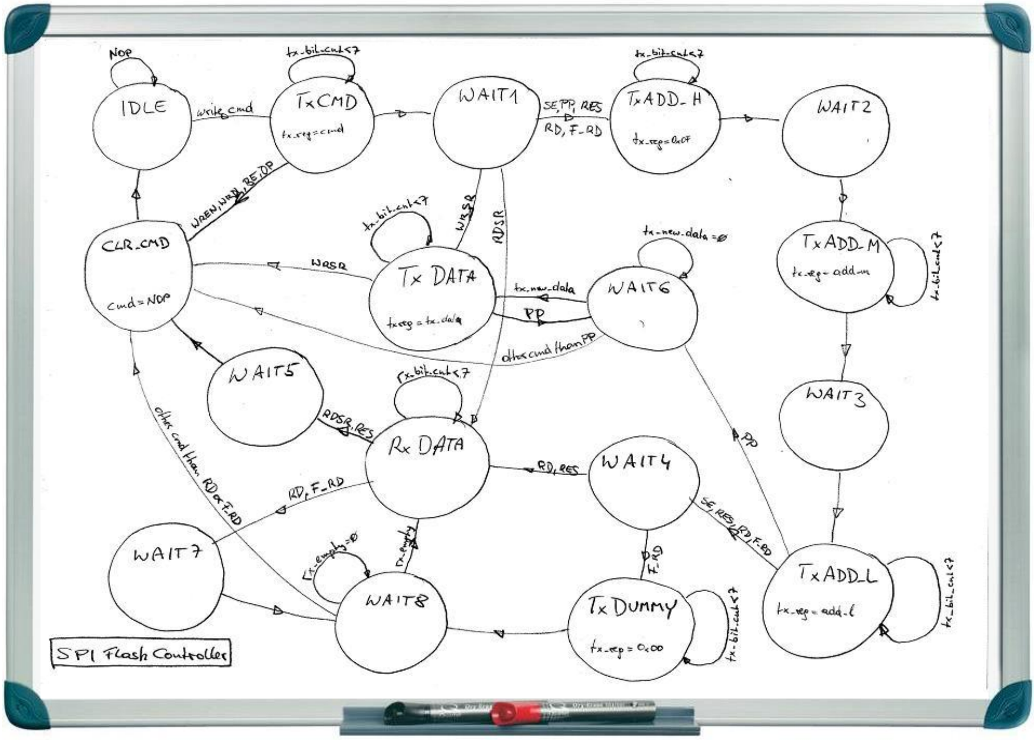
state machine
inference



model

The people who write specs, make implementations, or do security analyses probably all draw state machines on their whiteboards…

But will it they all draw an identical ones?




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

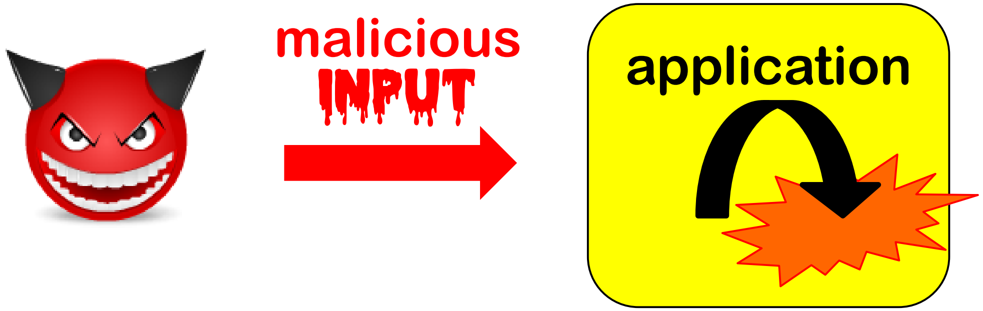
   – Classic examples: SQL injection, path traversal, XSS, Word macros

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

Processing vs Forwarding Flaws

Processing Flaws

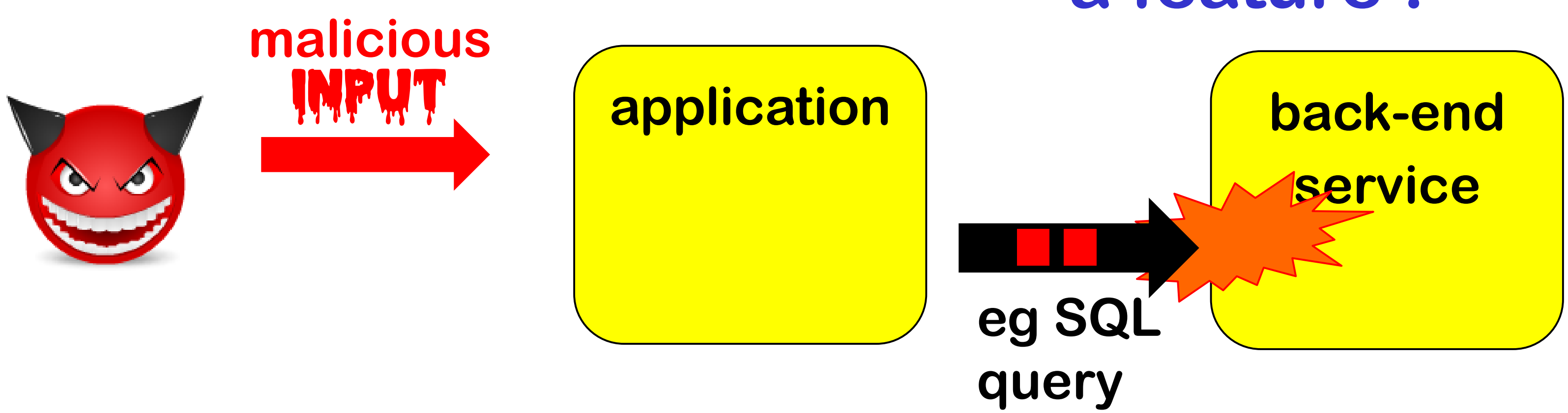
a bug !



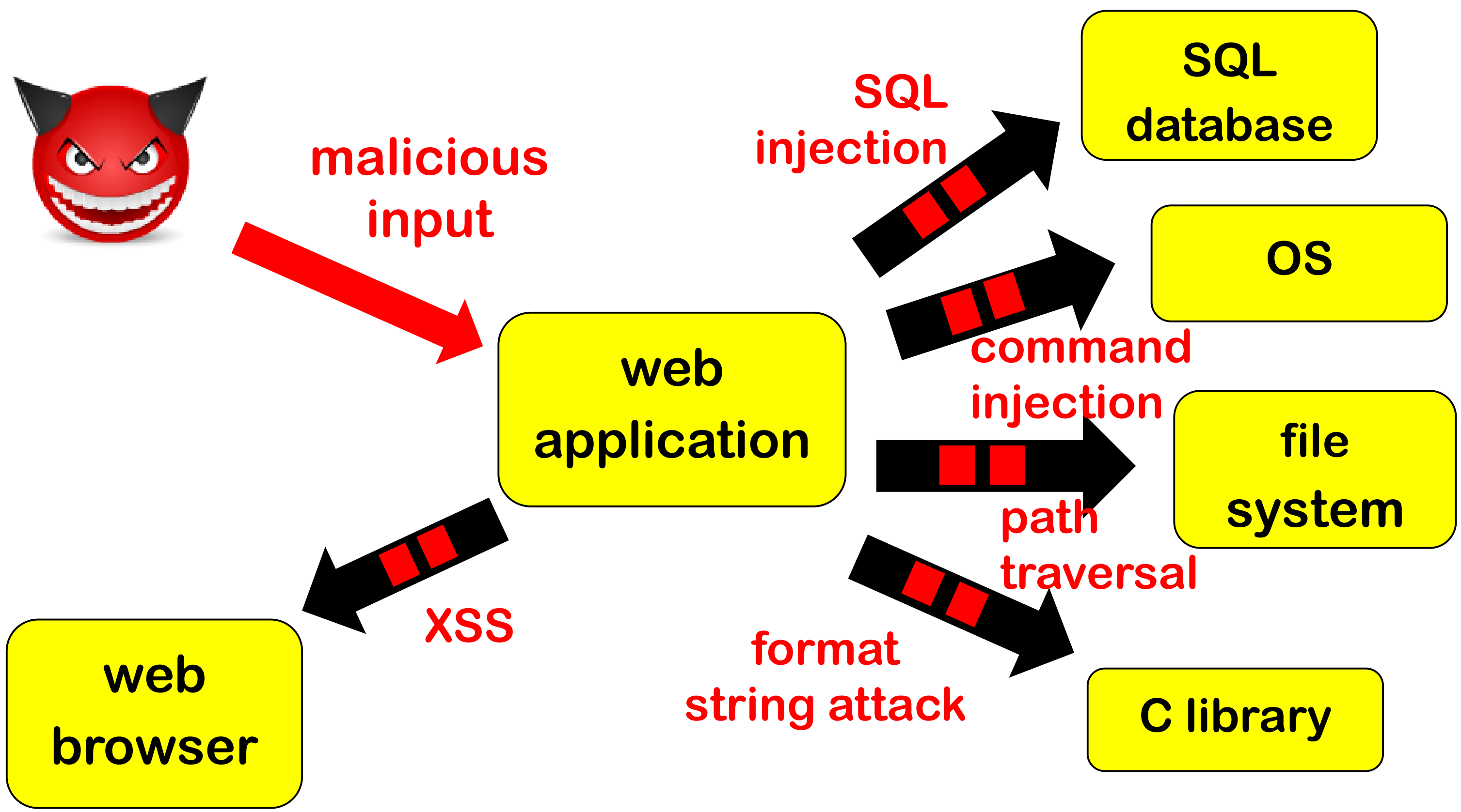
eg buffer overflow
in PDF viewer

Forwarding Flaws

(abuse of)
a feature !

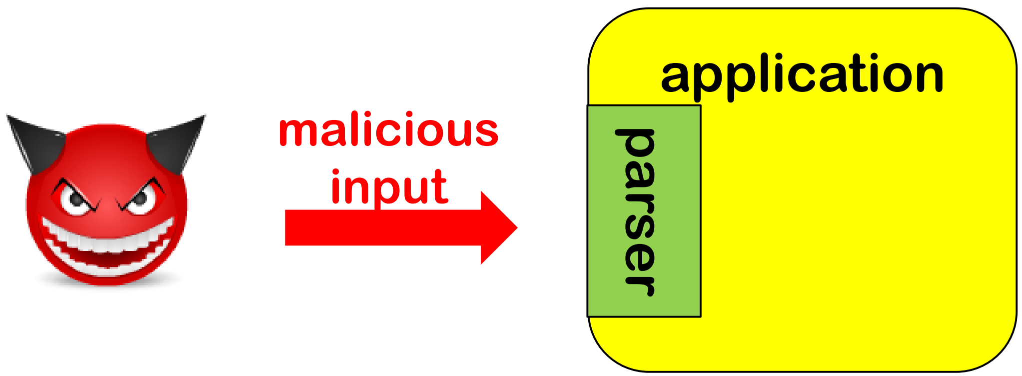


More back-ends, more languages, more problems



How & where to tackle input problems?

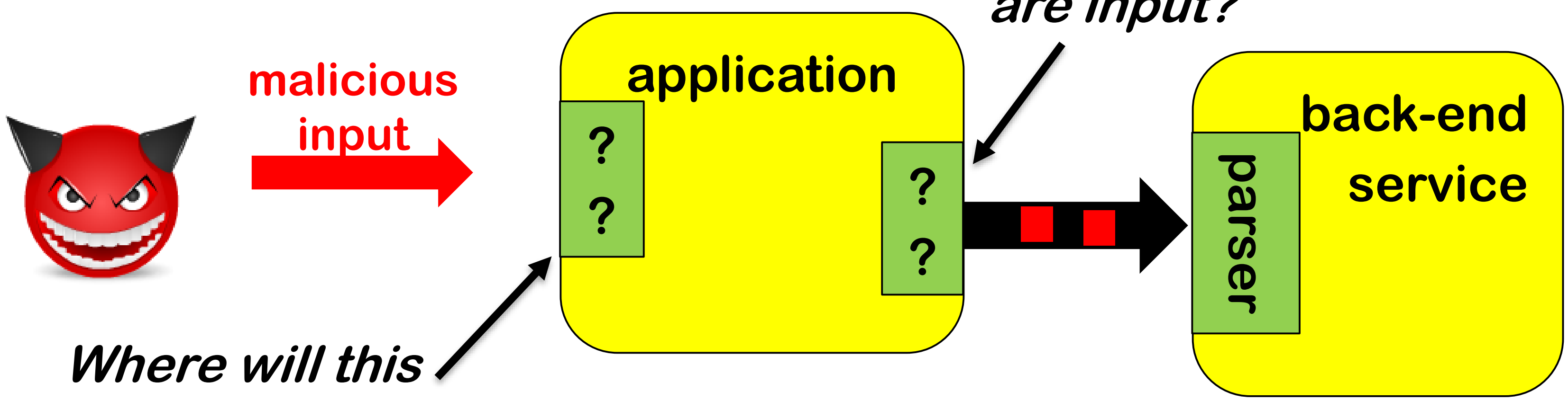
Tackling processing flaws



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

Tackling forwarding flaws?

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:
  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

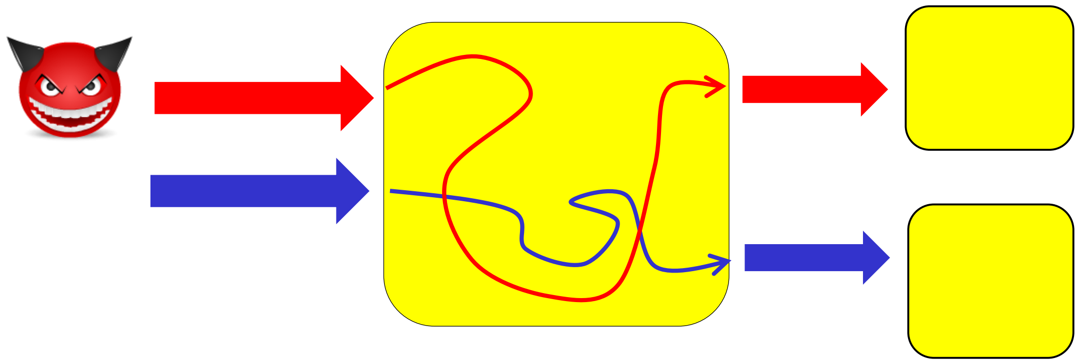
Types to the rescue!

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

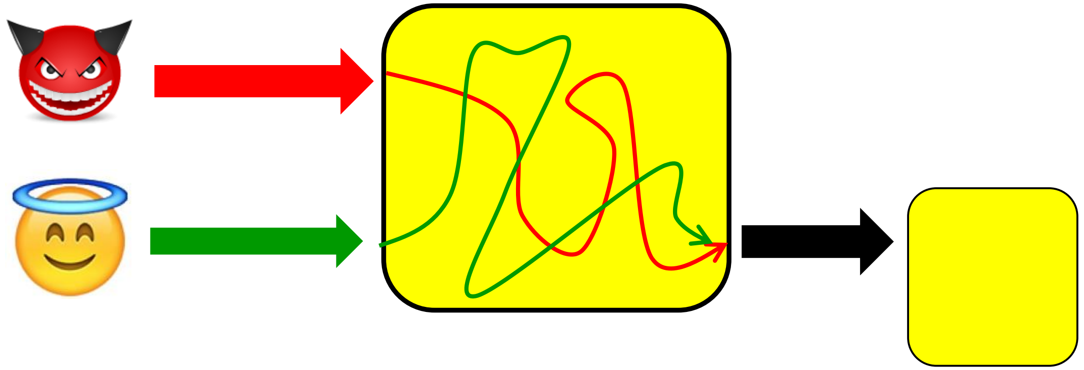


Remedy: Types (2) to distinguish *trust levels*

- 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

- 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

To read

- **Protocol state machines and session languages: specification, implementation, and security flaws LangSec'15**
- **LangSec revisited: input security flaws of the second kind, LangSec'18**