Embedded Software Security

ISSISP 2015

(6th Int. Summer School on Information Security and Protection)

Erik Poll

Digital Security group

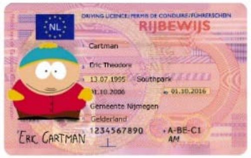
Radboud University Nijmegen

The Netherlands

Digital Security @ Radboud University Nijmegen

- formal methods for program analysis and protocol analysis esp. for smartcards & RFID

- smartcard applications
- cryptanalysis
- side channel analysis



- identity management & privacy, incl. legal aspects
- more applied security, eg voting machines, smart grid,...

Overview

1. Smartcards & RFID
2. Embedded Security
3. Side channel attacks & defensive coding } involves software and hardware
4. Dynamic security analysis: Fuzzing & automated reverse engineering } purely software related
5. Physical attacks } purely hardware related

Smartcards & RFID

Example smartcard & RFID uses

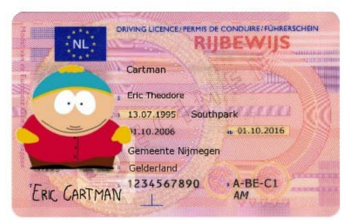
- bank cards
- SIMs in mobile phone

- public transport

- identity documents

  modern passports and national ID cards

  contain (contactless) chip



- access cards

  to control access to buildings, computer networks, laptops,...

- pay TV

Why smartcards?

Using cryptography can solve some security problems,

but *using cryptography also introduces new security problems:*

1. where do we store cryptographic keys?
2. who or what do we trust to use cryptographic keys?
3. key management – ie generating, distributing, revoking keys

Smartcards provide a solution for 1 & 2.

It helps with 3 by providing a way to distribute keys to users.

Alternative solution, used in back-ends: Hardware Security Modules (HSM)

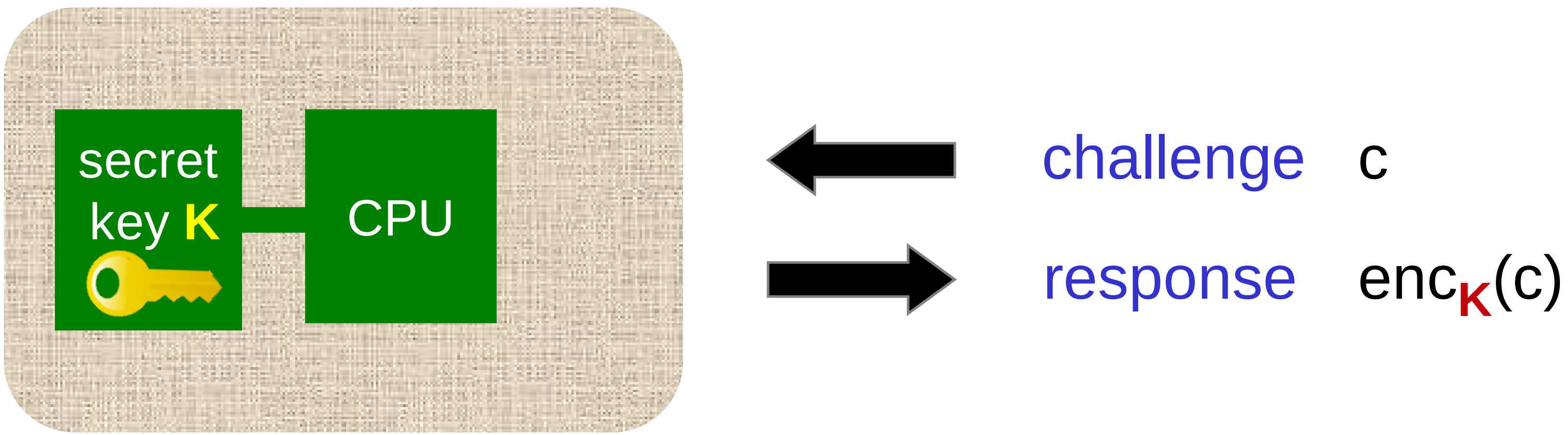
Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations.

They are also large, expensive to maintain, difficult to manage, and they pollute the environment.

It is astonishing that these devices continue to be manufactured and deployed. But they are sufficently pervasive that we must design our protocols around their limitations

– Kaufman, Perlman, and Speciner

Typical use of smartcard for authentication



- If card performs encryption, then key K *never* leaves the card
- *The card issuer does not have to trust the network, the terminal, or the card holder*

- The use of keys for encryption to ensure confidentiality is usually less important than for signing or MAC-ing to ensure integrity/authentication

What is a smartcard?

- Tamper-resistant computer, on a single chip, embedded in piece of plastic, with limited resources
  - aka *chip card* or *integrated circuit card (ICC)*
  - tamper-resistant, to a degree – *not* tamper-*proof*
  - tamper-evident, to a degree

- Capable of "securely" storing & processing data
  - This processing capability is what makes a smartcard *smart;* stupid cards can store but not process
  - NB processing capabilities vary a lot!

- Cards can have contact interface, or contactless (RFID) interface

3 types of functionality

1. ***stupid card*** just reports some data

   eg card shouts out a (unique) serial number on start-up

2. ***stupid smartcard*** aka **memory card**

   provides configurable file system with access control

   by means of *PIN code/passwords* or *crypto keys*

   or even simpler: *irreversible writes*

3. ***smart smartcard*** aka **microprocessor card**

   provides programmable CPU that can implement any functionality

   eg complicated security protocols

What type of attacks can 2 & 3 withstand that 1 can't?

Replay attacks!

Microprocessor smartcard hardware

- CPU (usually 8 or 16, but now also 32 bit)
- possibly also crypto coprocessor & random number generator (RNG)
- memory: volatile RAM and persistent ROM & EEPROM
  - EEPROM serves as the smartcard's hard disk
- NB no power, no clock!

A modern card may have 512 bytes RAM, 16K ROM, 64K EEPROM and operate at 13.5 MHz

Important reason for low capabilities: cost!

Also, keeping smartcard simple allows higher confidence:
you don't want Windows or Linux as operating system on a smartcard

Smartcard Operating System (OS)

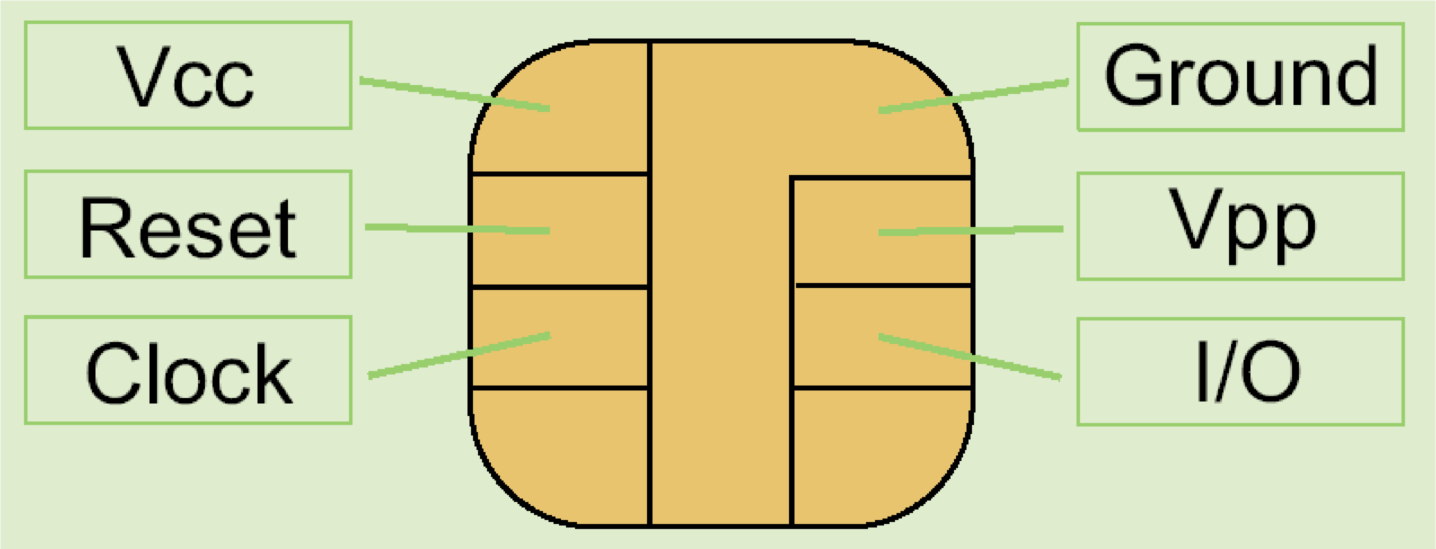
- Microprocessor smartcards come with very simple operating system. Simple because there is no multi-threading, no complex device drivers
- Old-fashioned smartcards contain one program, that cannot be changed
- Modern smartcard platforms
  - are multi-application, ie allow multiple, independent programs (aka applets) to be installed on one card
  - allow post-issuance download: applications to be added (or removed) after the card has been issued to the card holder

Of course, this is tightly controlled - by *digital signatures*

Examples of such modern platforms: JavaCard and MULTOS

Application management typically uses the GlobalPlatform standard

Contact cards (ISO 7816)



External power supply and external clock

- Originally 5 V, now also 3V or 1.8V
- Vpp - higher voltage for writing EEPROM
  No longer used as it introduces a serious security weakness!

The terminal problem!

- THE fundamental problem with smartcards

  no trusted I/O between user and card

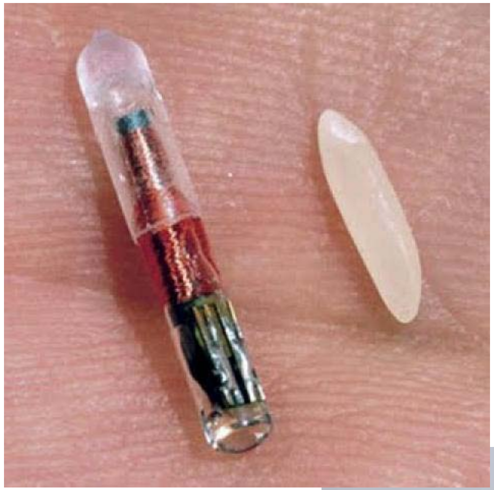
  - no display
  - no keyboard

Solutions:

- Card with built-in display & keyboard
- Alternative: give people a reader

RFID tags

There are many types of RFID, with different operating ranges.

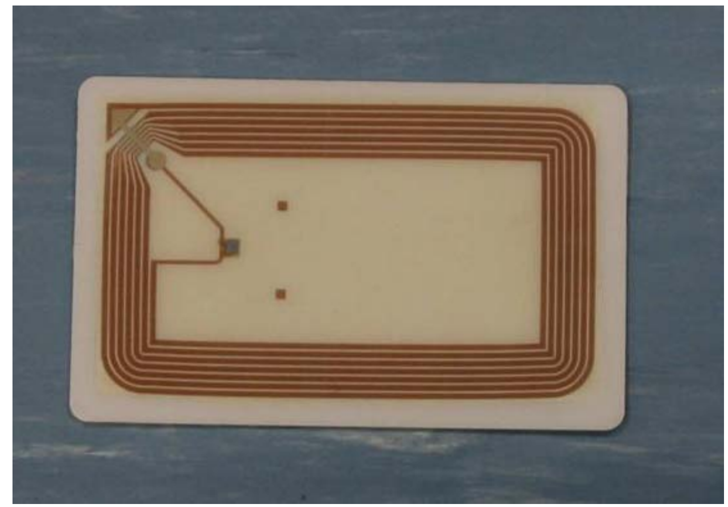


Contactless cards aka RFID (ISO 14443)

Commonly used (passport, public transport, contactless bank cards):

ISO14443 aka proximity cards

- operating range of < 10 cm
- antenna used both for
  - powering card
  - communication



- ISO14443 is compatible with NFC in mobile phones

Using and/or attacking *at larger distances*

- Max. distance at which a proximity card can be activated is 50 cm
- Max. distance at which it can be eavesdropped is > 10 meters

[René Habraken et al., *An RFID skimming gate using Higher Harmonics*, RFIDSec 2015]

Embedded Security

What makes embedded security special?

Important characteristics

- Limited resources (storage, processing, I/O bandwidth, ..)
- More exotic hardware & software platforms, and protocols

Some good news for security:

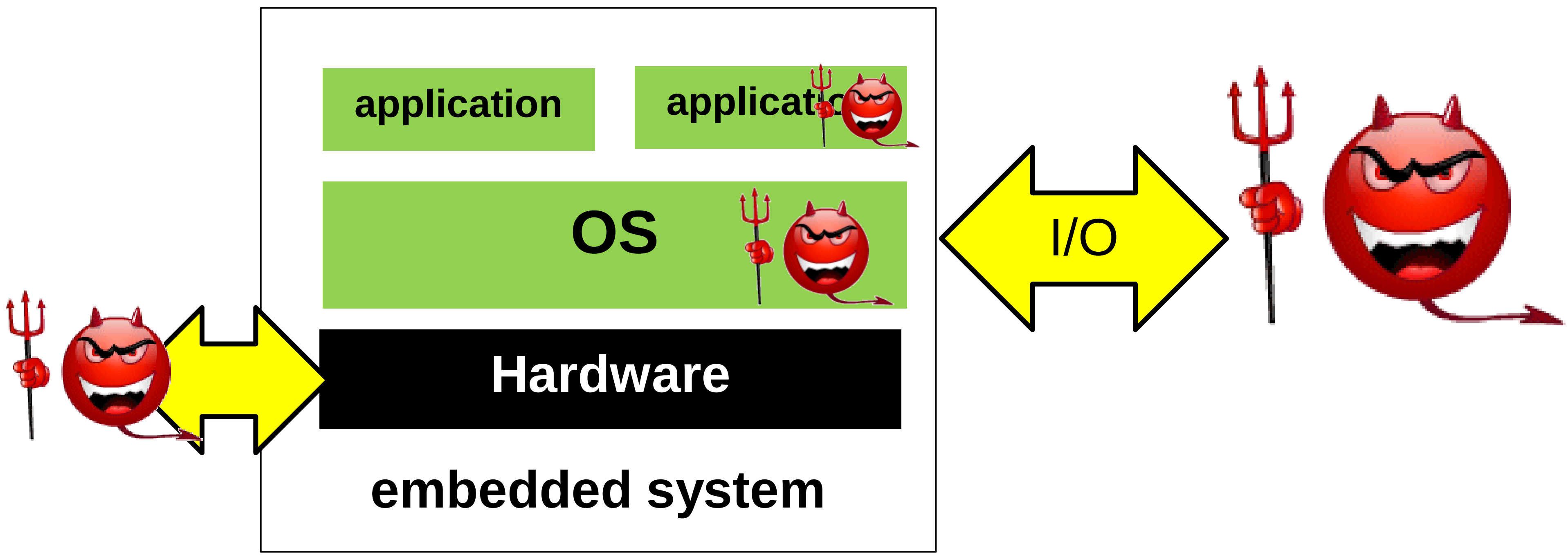
- less software – hence fewer exploitable bugs
- attacker may lack knowledge about the platform & protocols
- no OS access, eg to binaries

But also some bad news:

- attacker has **physical access**
- fewer resources for countermeasures

Attacker model for embedded security

Even if all software is correct & free from security bugs
then the attacker can still *observe* or *attack* the hardware

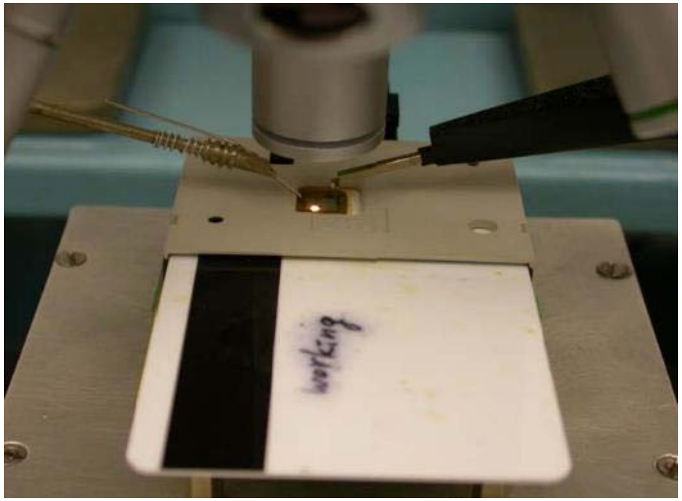


Specific threat for embedded software

Attacker can get physical access
and carry out physical attacks
on the endpoints.

Physical attacks can attack

1. the static hardware
(eg extracting ROM content) , or
2. attack the card while it is executing



Security flaws can happen at many levels

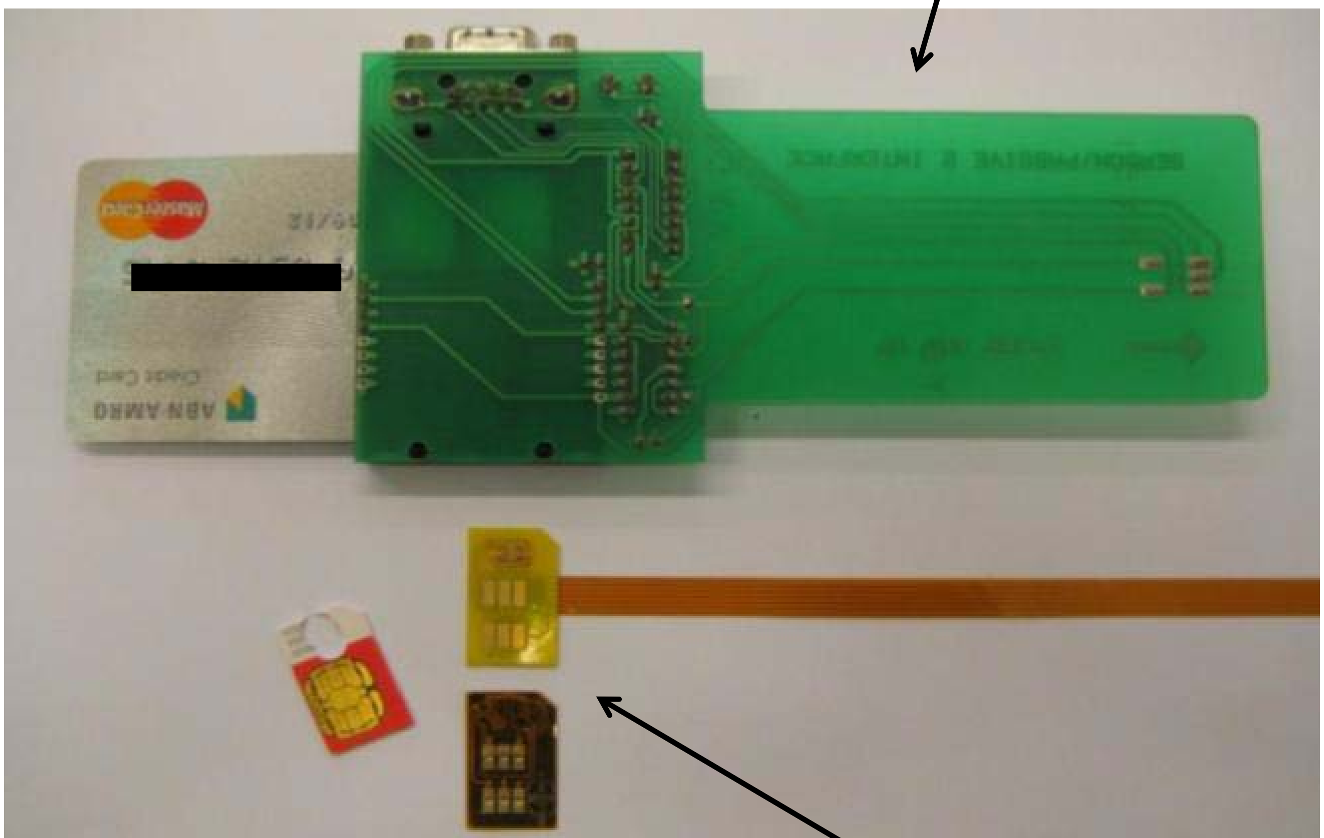
1. Choice of cryptographic primitives (AES, RSA, …)
   Classic mistake: proprietary crypto
2. Key management
   Classic mistake: default keys or same key in many devices
3. Implementation of the cryptographic primitives
   Bad implementation will leak keys, as we will discuss at length
4. Design of the security protocol
   Designing security protocols is notoriously tricky
5. Implementation of the security protocol
   ie. software bugs, as usual
6. Purely physical attacks, eg to extract memory contents (especially keys)

1 & 2 common
but easy to avoid!

3, 5, 6 depend on
HW characteristics

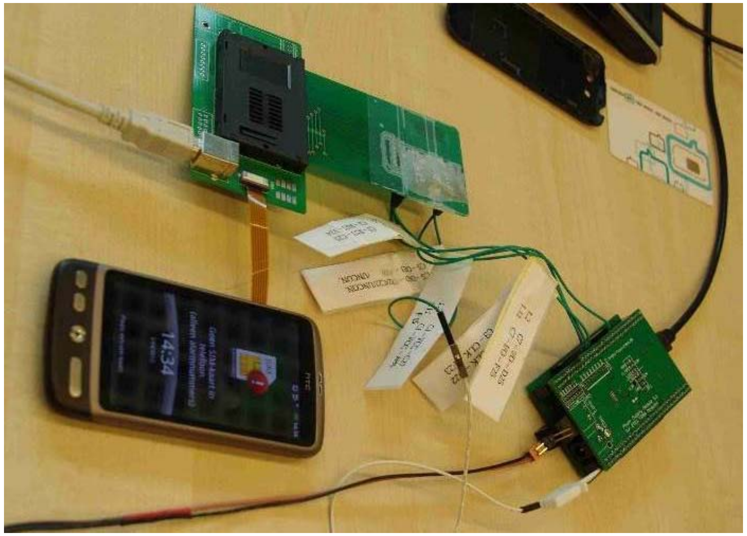
Our toys for protocol analysis

old-fashioned version
(used for hacking pay TV)

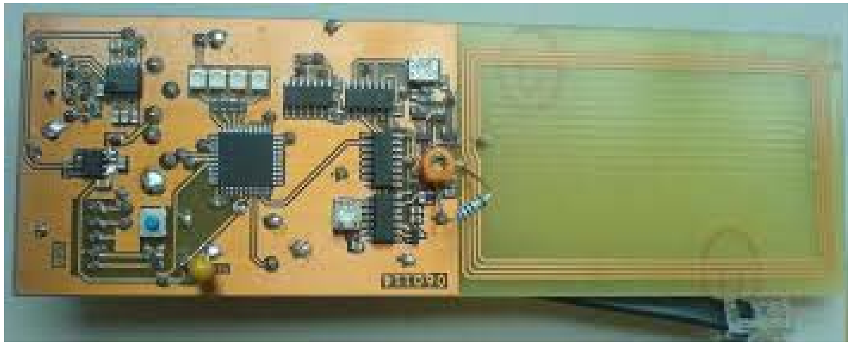


newer, thin versions
(used for studying SIM locking)

Our toys for protocol analysis



Our toys for protocol analysis



Classic flaw 1: flawed cryptography

Homemade, *proprietary* cryptographic algorithms are routinely broken.

For example

- Crypto-1 used in MIFARE Classic, eg. by MetroRio
- COMP128 and A5/1 used in GSM
- SecureMemory, CryptoMemory, CryptoRF
- iClass, iClass Elite
- HiTag2
- Megamos
- ...

https://www.youtube.com/watch?v=NW3RGbQTLhE
https://www.youtube.com/watch?v=S8z9mgIkqBA

Stick to AES, RSA, ECC, SHA2, 3DES, …!

Classic flaw 2: flawed key management

Many systems are poorly configured wrt. using cryptographic keys

For example,

- Lukas Grunwald found 75% of systems using MIFARE RFID tags
  - use the default key (which is A0A1A2A3A4A5)
  - or, use keys used in examples in documentation

- *All* HID iClass RFID tags *worldwide* use the same master key

  Moreover, this master key is in all RFID readers sold by HID.

Extracting one key from one device should <u>not</u> break the entire system!

Classic flaw 3: flawed security protocols

Security protocols are notoriously tricky

`Three line programs people still manage to get wrong' *[Roger Needham]*

For example

- Some systems using RFID cards only rely on the unique serial number (UID) of the card for authentication.
  This UID can be trivally replayed, and possibly copied to other cards
- One variant of MasterCard's EMV-CAP standard for internet banking using a nonce (random *N*umber used *ONCE*) which is always 0x0000000
- HID's more expensive HiClass Elite is *less secure* than the standard HiClass.
- …

Keep security protocols – and hence their implementations – simple!

Side-channel attacks

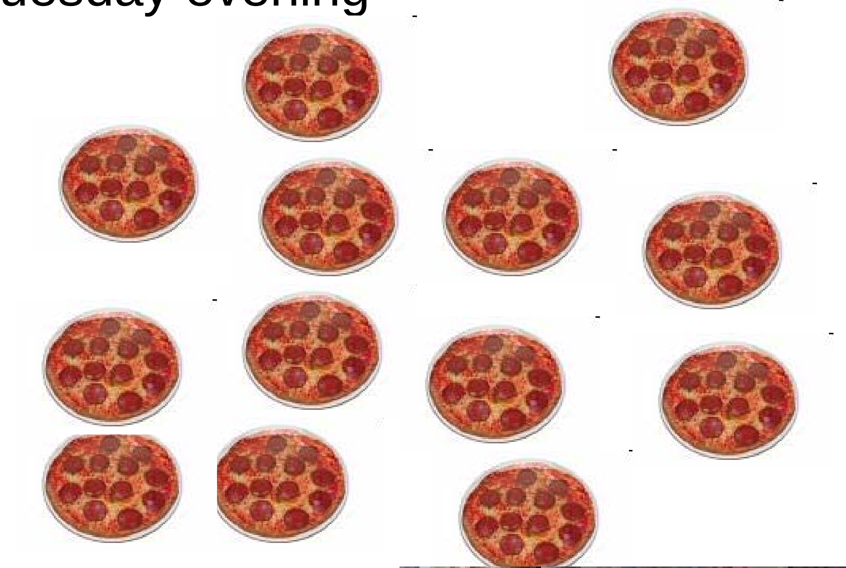
Pizza deliveries as side-channel

Example side channel: pizza deliveries to the Pentagon

Pizza deliveries as side-channel

monday evening

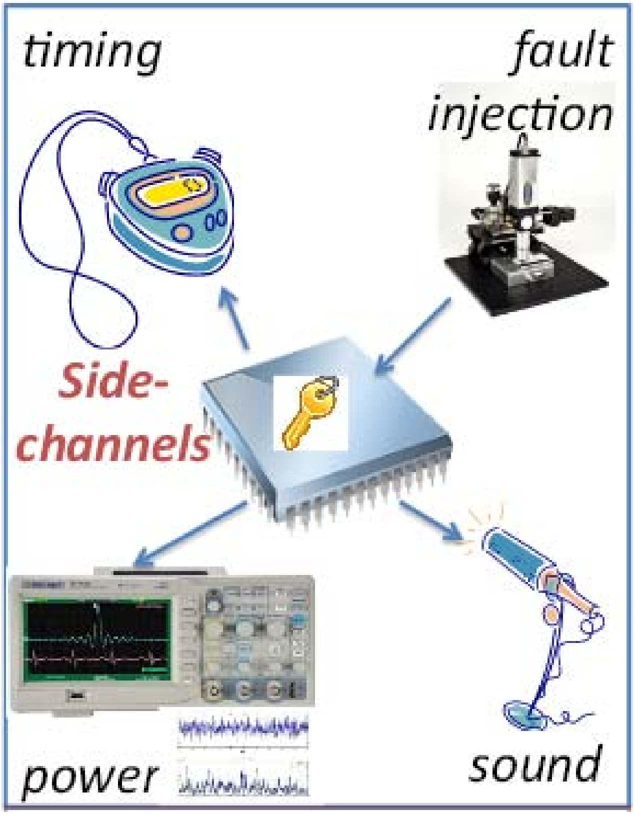
tuesday evening



What morning is the invasion taking place?

Side channel analysis

- Side-channel = any other channel than the normal I/O channel that may be observed (or interfered with)
- Possible side-channels:
  - power consumption
  - timing
  - electro-magnetic (EM) radiation
  - sound
  - ....



Some history of side channel attacks

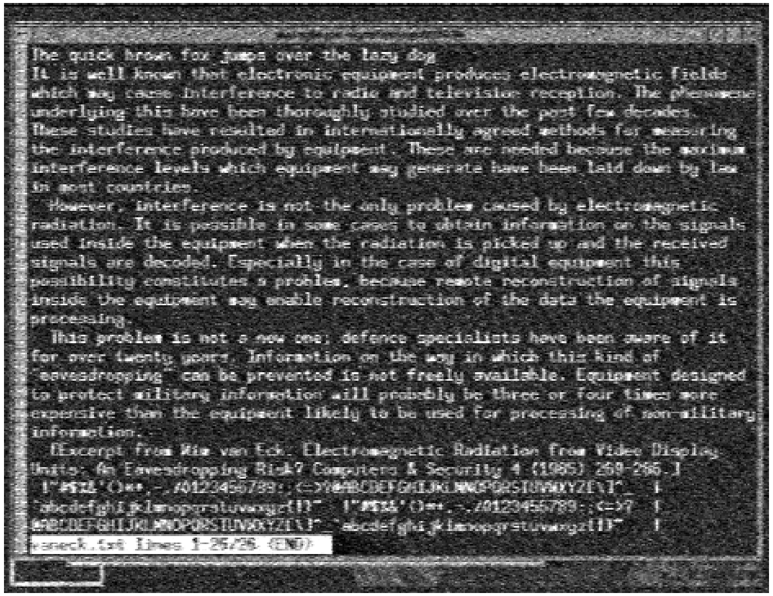
- TEMPEST since 1960s computers are known to emit EM signal
  - First evidence in 1943: an engineer using a Bell Telephone 131-B2 noticed that a digital oscilloscope spiked for every encrypted letter
  - Declassified in 2008
- In 1965 MI5 put microphone near rotor-cipher machine in Egyptian embassy: click-sounds of machine used to break encryption
- First academic publications by Paul Kocher et al. in 1990s:
  1996 (timing) and 1999 (power)

  [P. Kocher. *Timing Attacks on implementations of Diffie-Hellman, RSA, DSS and Other Systems*, CRYPTO 1996]

  [P. Kocher, J. Jaffe, B. Jun, *Differential Power Analysis*, CRYPTO 1999]

TEMPEST examples

- Laptop screen at 10 meter through 3 (thin) walls



- Dutch electronic voting machines were banned because EM radiation could break vote secrecy

Side channel attacks

Side channel attacks are the Achilles' heel of cryptography!

- The mathematics of cryptography is *very elegant*,
  making implementation resistant to side-channel attacks is *very messy*

- Origin of the CHES (Cryptographic Hardware & Embedded Systems) conference

Examples coming up

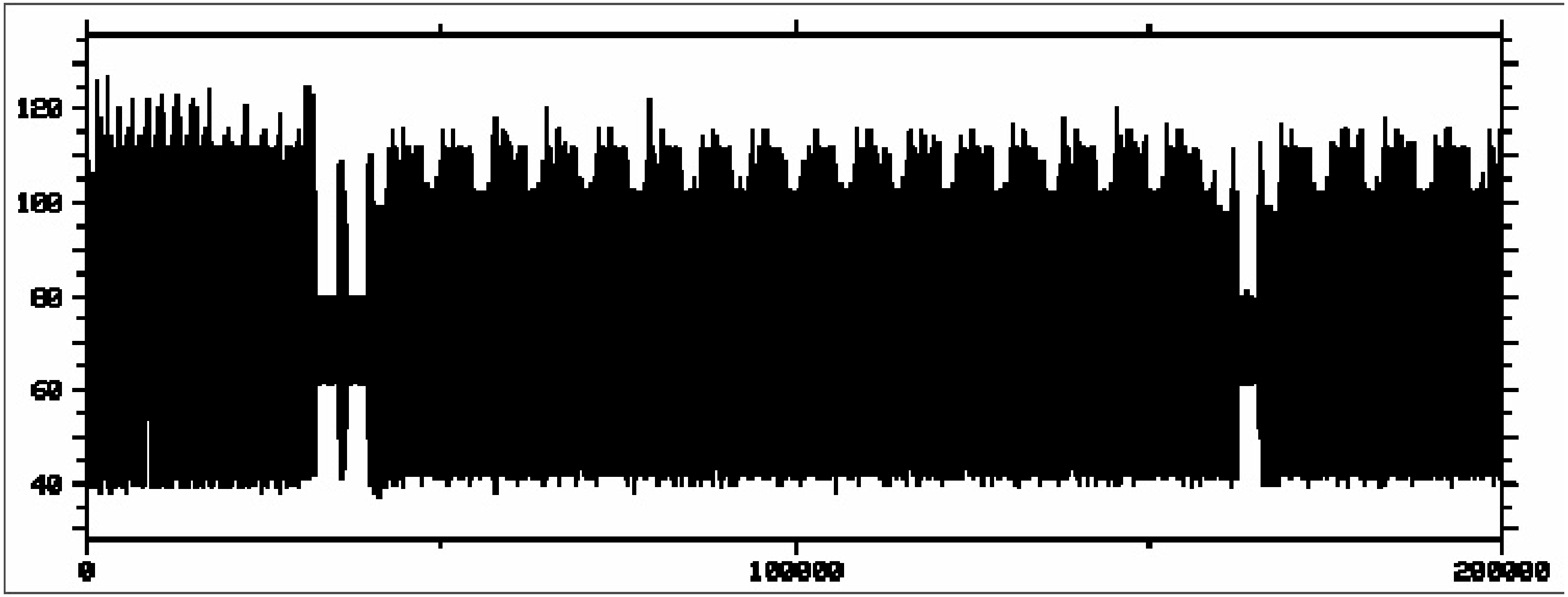
- Simple Power Analysis (SPA) } passive
- Differential Power Analysis (DPA) } passive
- Fault Injection } active

Power analysis

- Power analysis uses the electricity consumption of a device as side-channel
- Power analysis typically leaves the card intact, so is *not tamper-evident;* it is a so-called non-invasive attack
- The power consumption can depend on
  - the instruction being executed
  - Hamming weight of data being manipulated
    ie. number of 1 bits in the data
  - Hamming distance between consecutive values
    eg. for CMOS, switching 1→0 will have different power profile than 1→1, and the power consumption will depend on the number of bits flipped.

Equipment to analyse power as side-channel

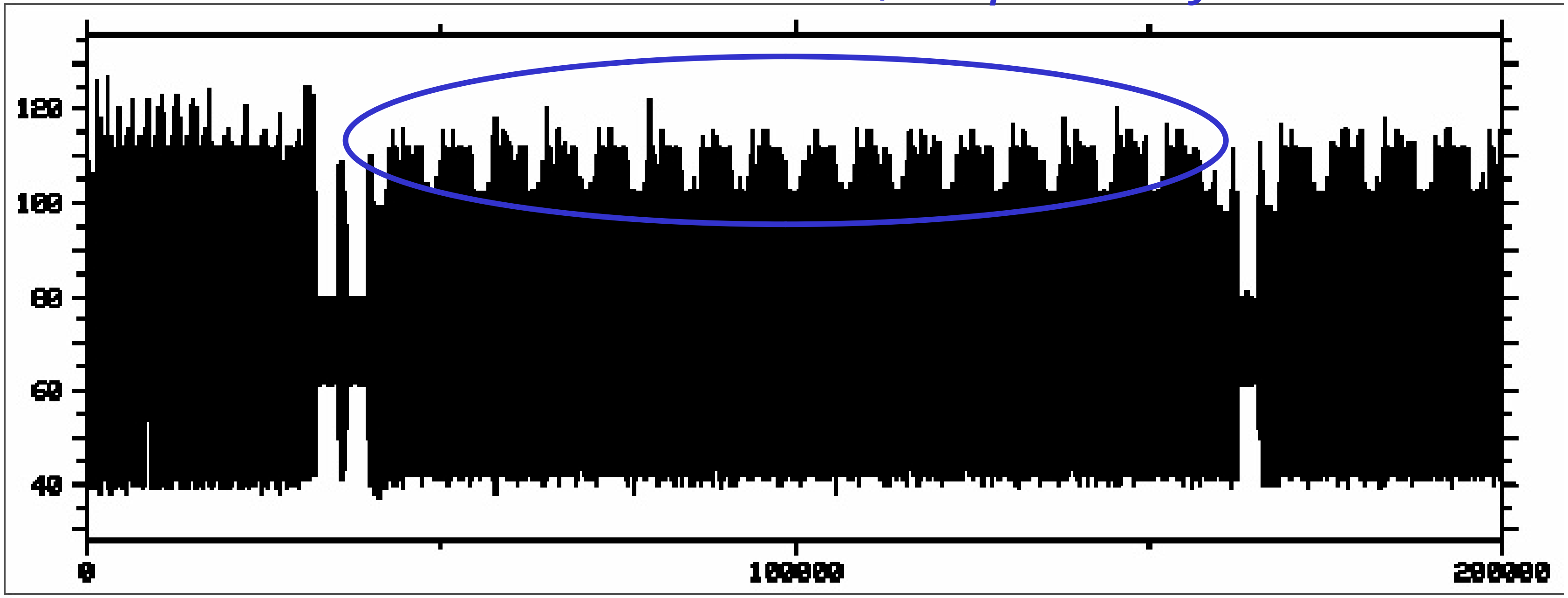
Power consumption of a smartcard



What is this card doing?

This is a DES encryption!

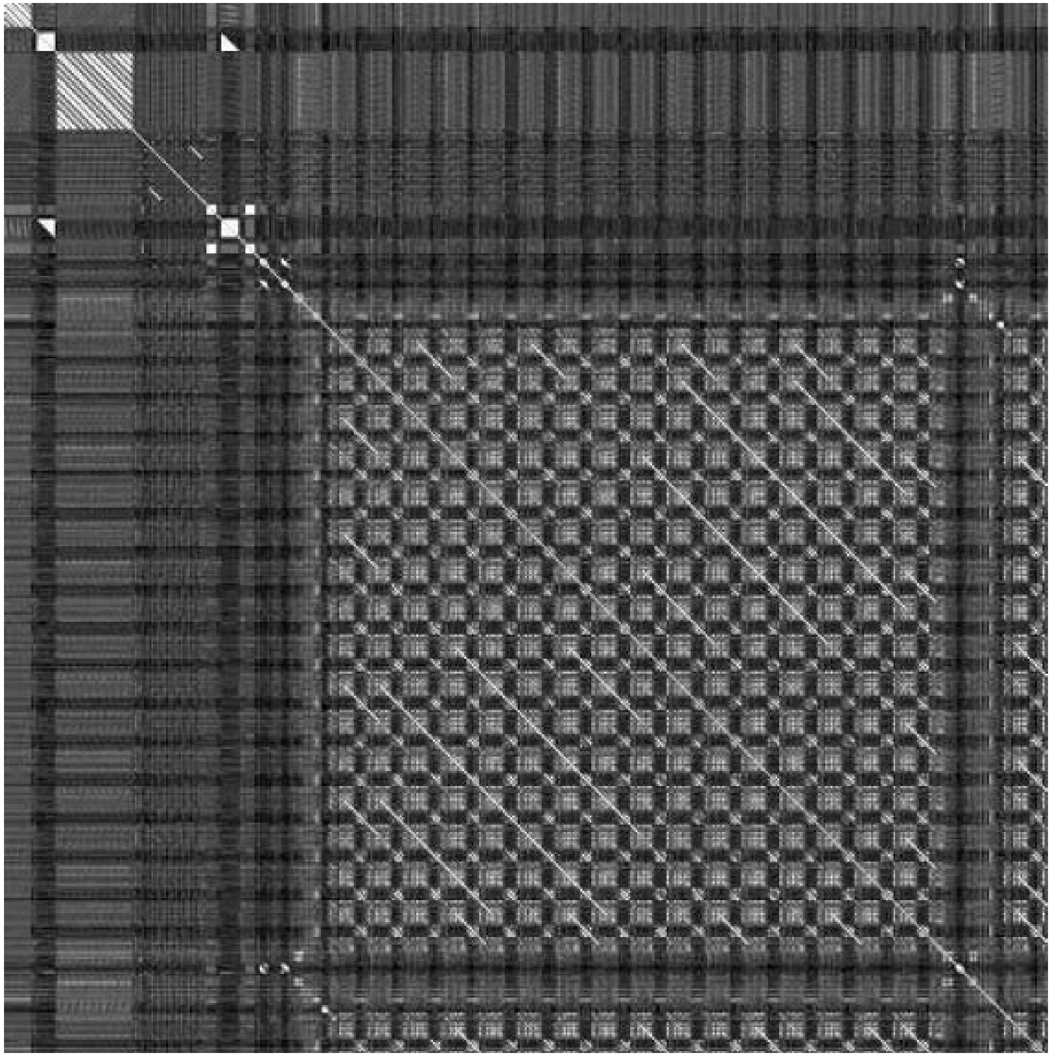
16 rounds, so probably DES



What is the key?

In a really poor implementation, you might be able to read it off…

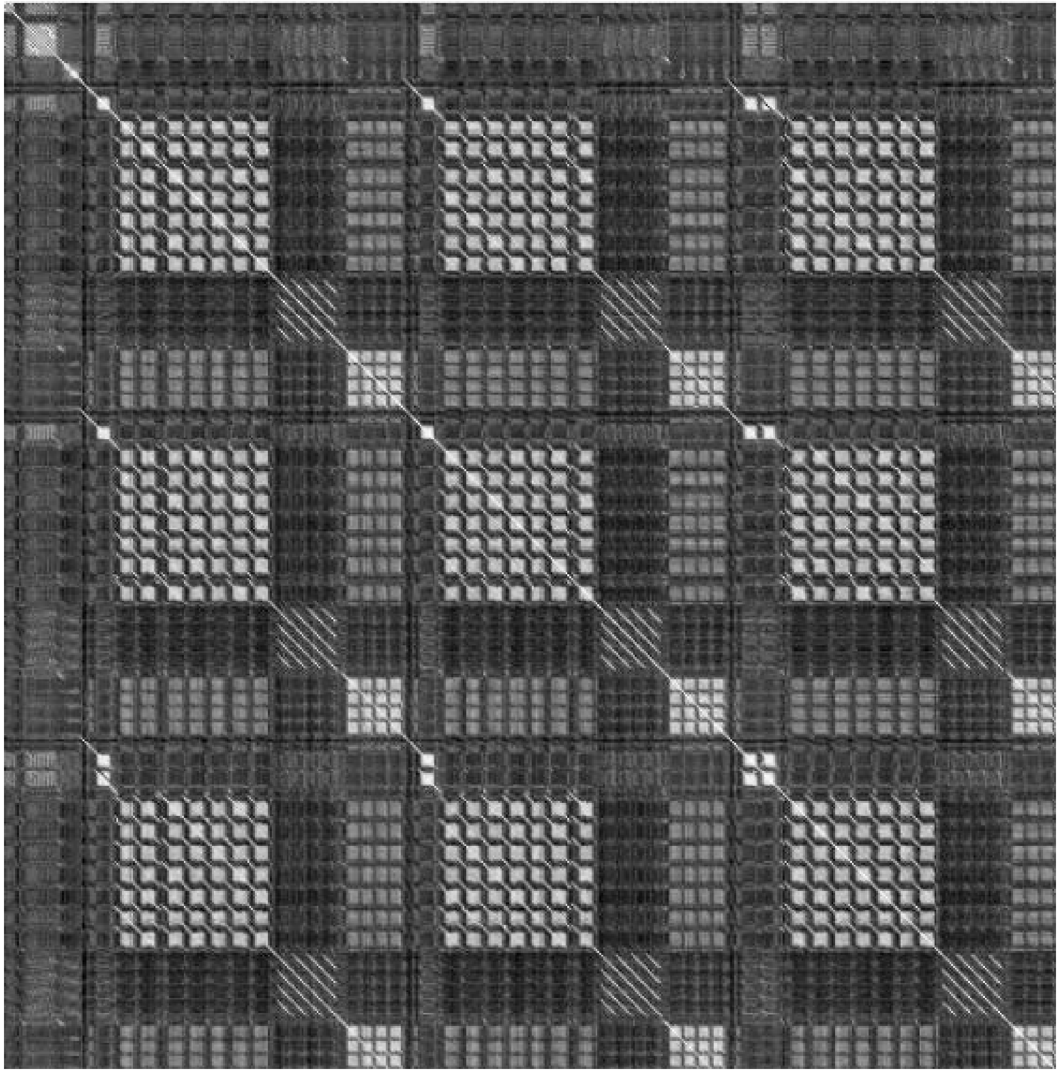
2D correlation matrix



- power trace s split in small sections s_i
- matrix m_{ij} expressing correlation between s_i and s_j
- light – higher correlation
- dark – lower correlation

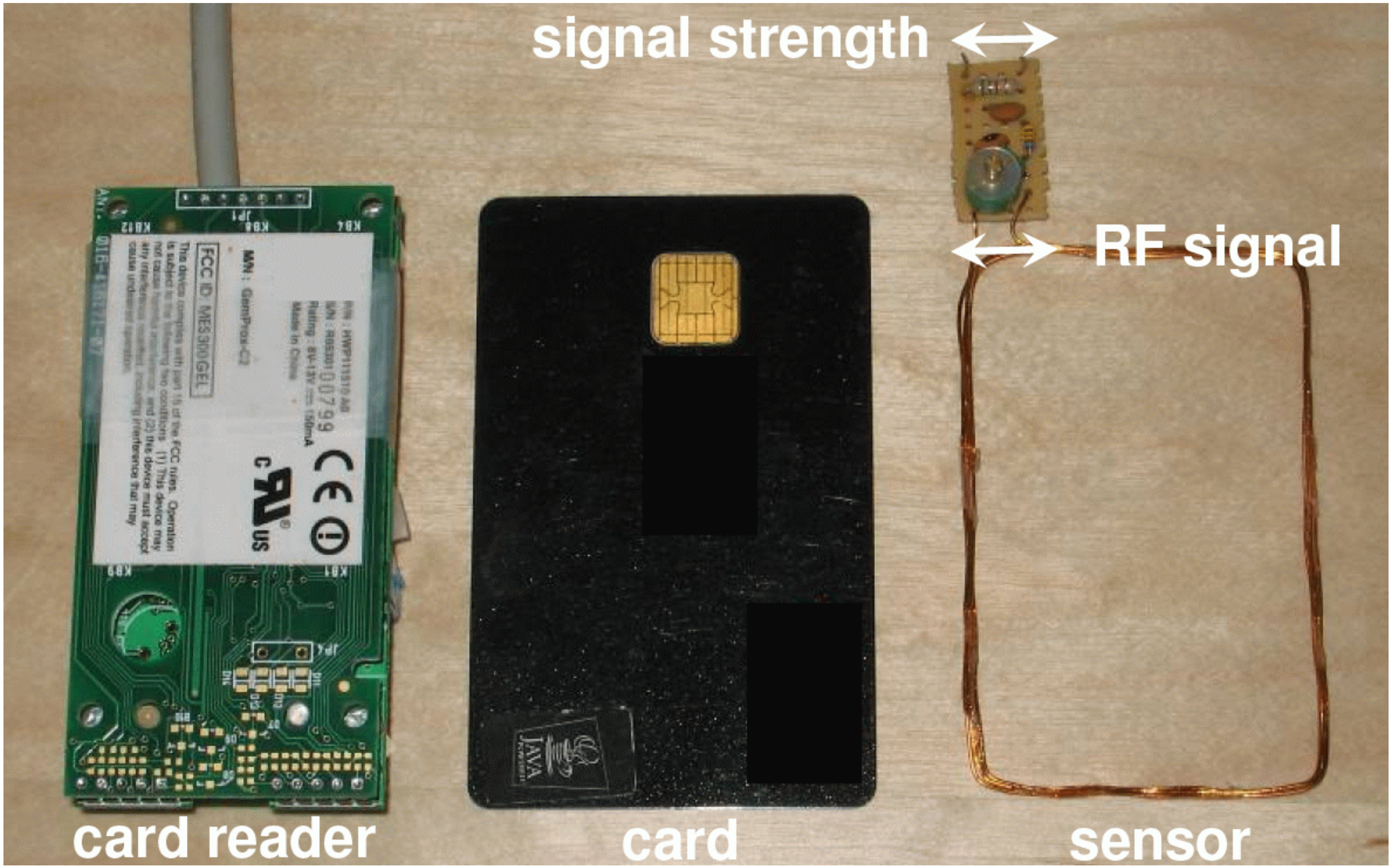
Example thanks to brightsight®

2D correlation matrix



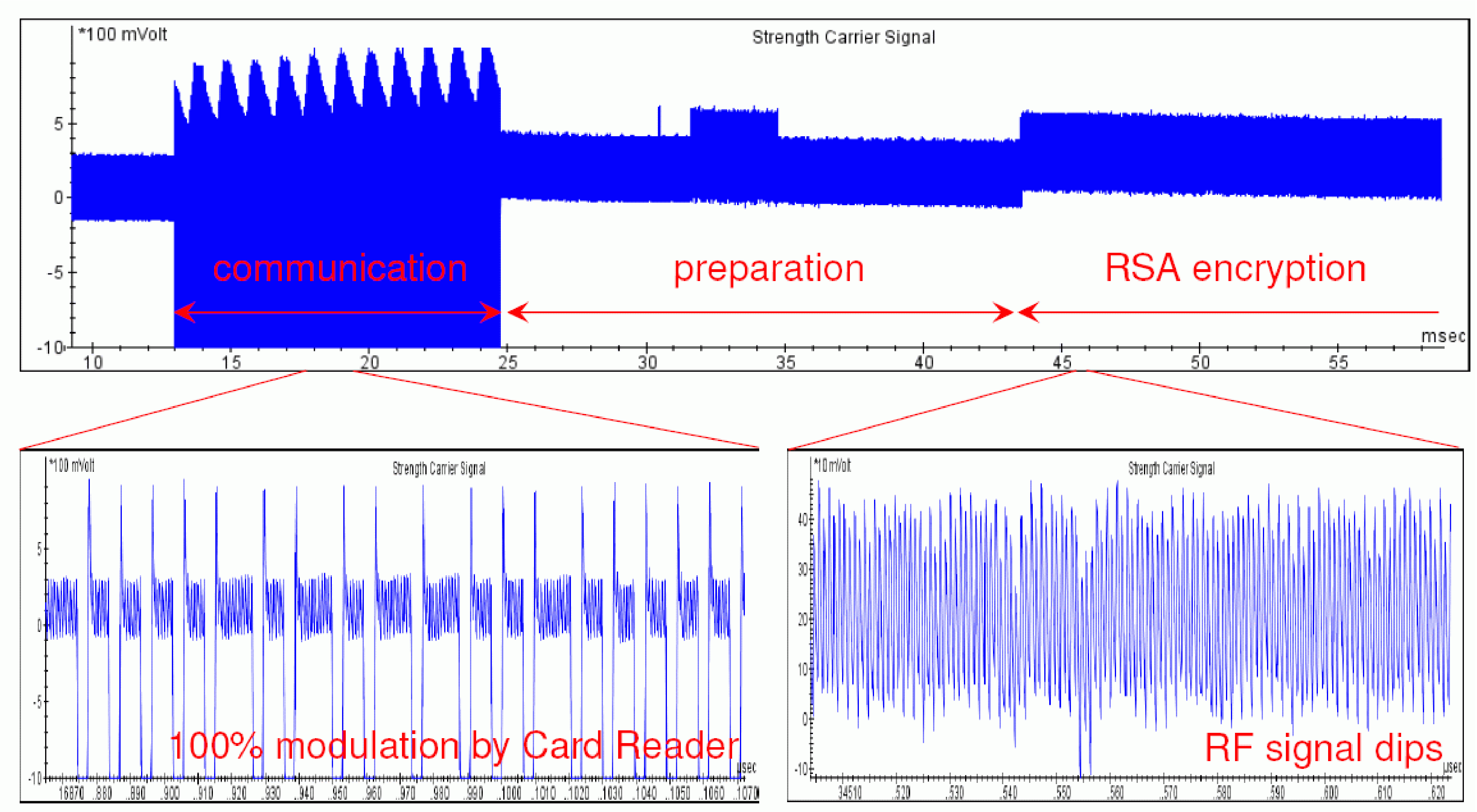
more detailed view

Example SPA analysis of contactless card

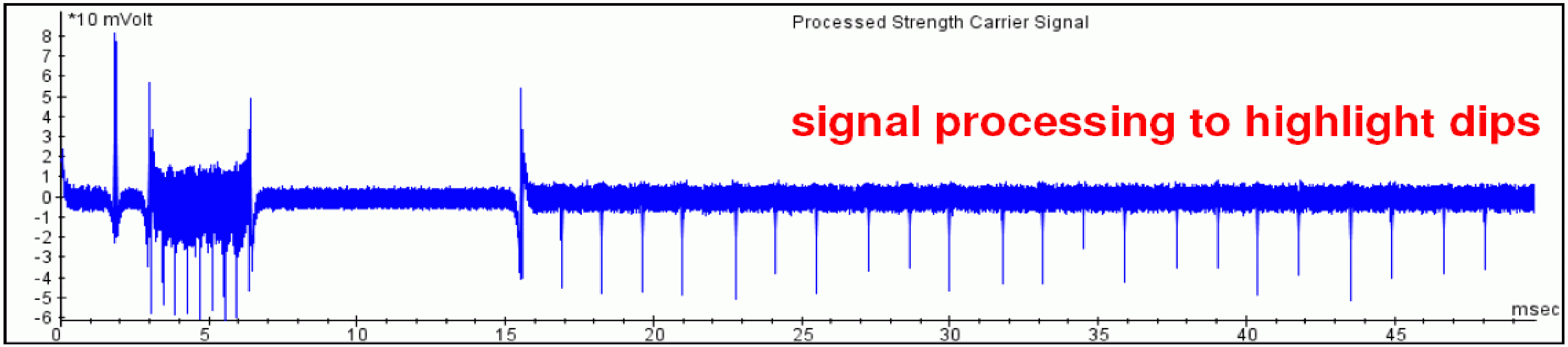


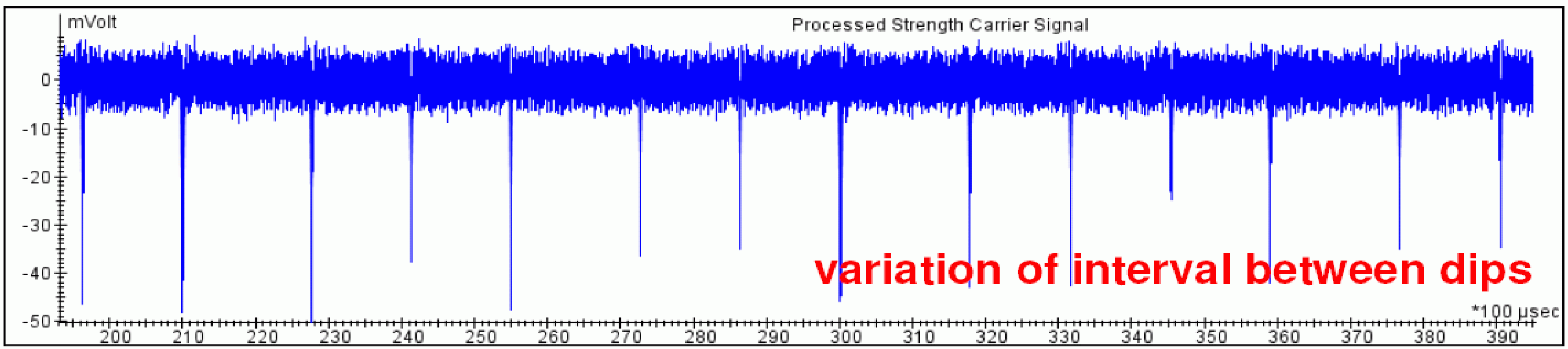
Example thanks to riscure

Power trace


*100 mVolt
Strength Carrier Signal
communication
preparation
RSA encryption
msec
100% modulation by Card Reader
RF signal dips

Power trace detail of RSA encryption





RSA implementation

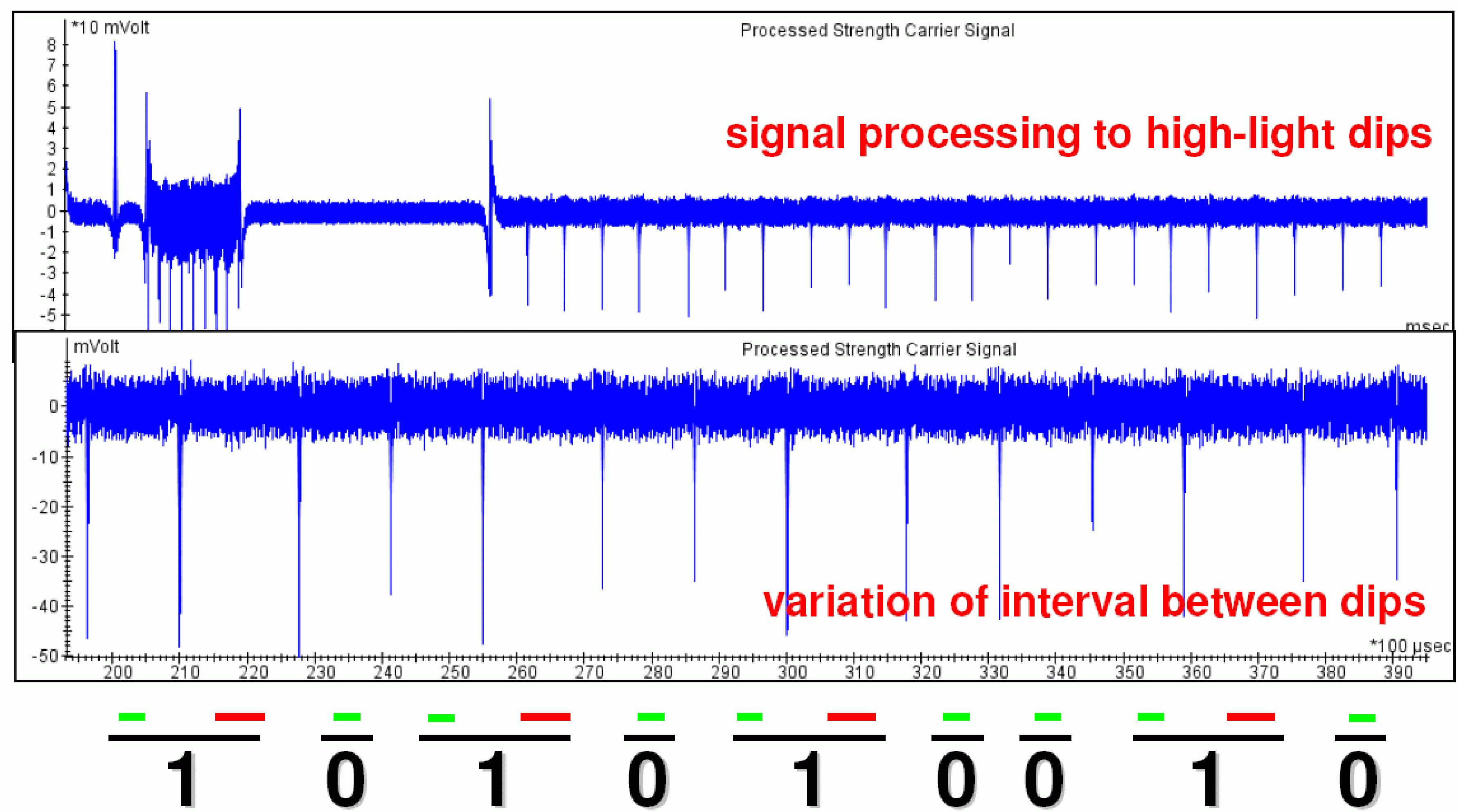
- RSA involves exponentiation $s = (x^y) \bmod n$
- Typical implementation with binary square & multiply:

eg $x^{27} = x^{1} * x^{2} * x^{8} * x^{16}$ of the form

```
s=1;
while(y) { if (y&1) { s = (s*x) mod n; }
                      y>>=1;
                      x = (x*x) mod n; }
return s;
```

- Only multiplies if bit in key is 1
- This enables timing or (D)PA attack

SPA: reading the key from this trace!



Countermeasures to side channel analysis

- Make execution time data-independent

  This is possible, possibly at the expense of efficiency, eg replacing

  ```
  if (b) then { x = e } else { x = e' }
  ```

  by

  ```
  a[0]=e; a[1]=e'; x = ( b ? a[0] : a[1] );
  ```

- Use redundant data representation, to reduce/eliminate differences in Hamming weights
- Most extreme case: use dual rail logic, representing 0 as 01 and 1 as 10
- Add redundant computations to confuse attacker;
  eg activating crypto-coprocessor when it's idle
- Add noise (eg clock jitter)

Power Analysis

- Simple Power Analysis – SPA

  analyse an *individual* power trace

  - to find out the algorithm used
  - to find the key length
  - worst case: to find the key, as in the previous example

- Differential Power Analysis – DPA

  statistically analyse *many* power traces to find out the key

  The most serious threat to smartcards in the past 15 years!

Differential Power Analysis (DPA)

- Suppose we have a large set of power traces S of same program (say an encryption) acting on different, randomly chosen data.

  There will be variations in power traces, due to noise & differences in data

- Partition S into two subsets, S_0 and S_1
- Consider the trace Δ = (average of S_0) - (average of S_1)

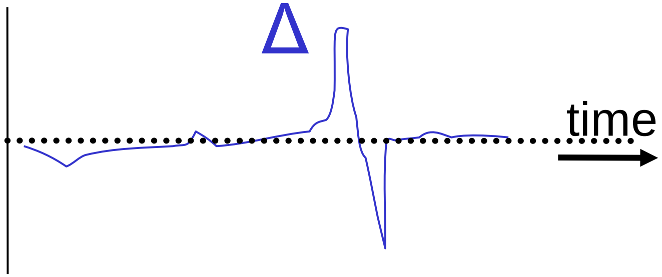
Now

- *What would you expect Δ to be?*

  A flat line, as noise and random data differences should cancel out

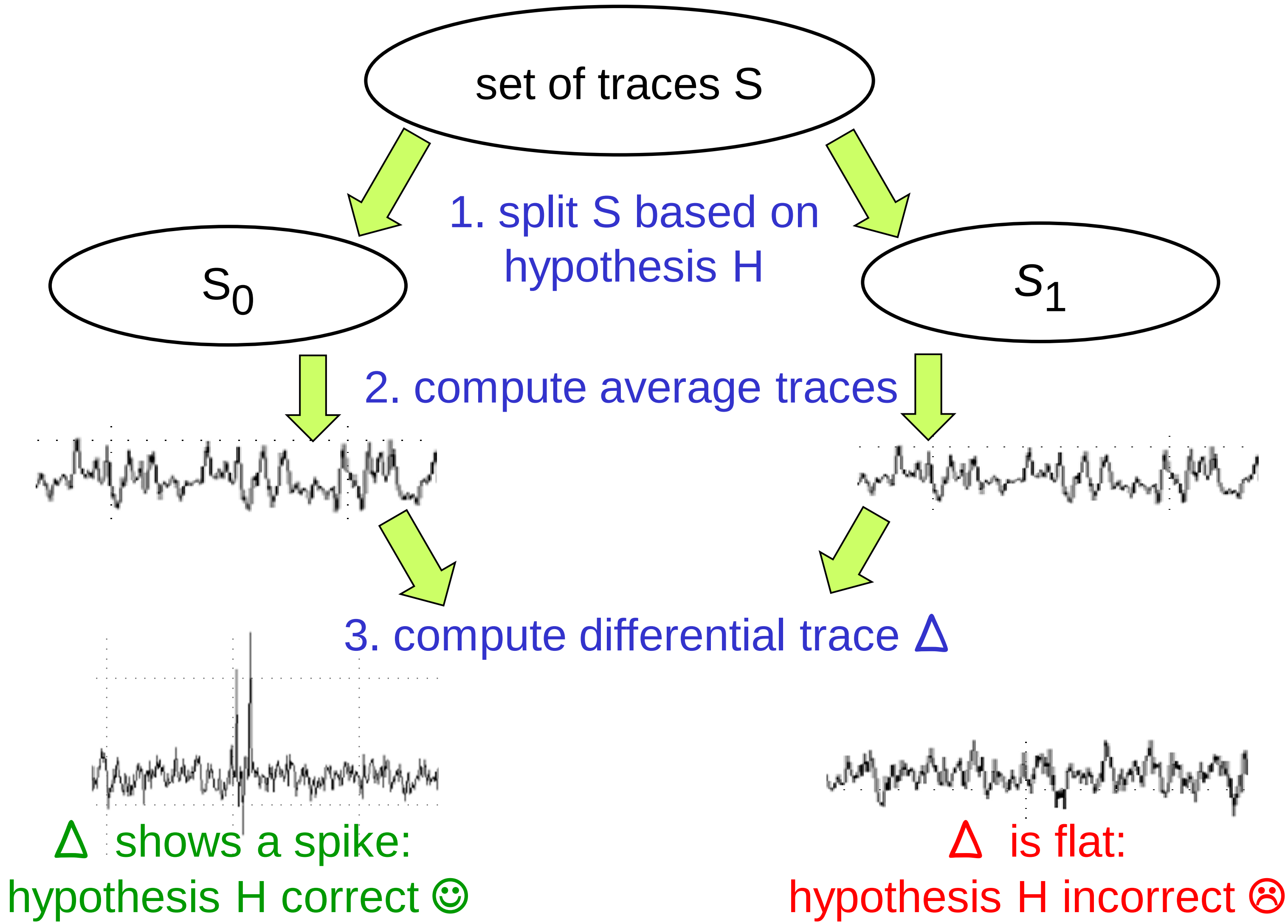
- *What if there is a big blip in Δ?*

  Eg what if Δ looks like



All traces in S_0 must be doing the same at that point in time!

DPA (schematically)



Differential Power Analysis (DPA)

1. Record set S of traces of encryptions of random data with unknown key

2. Split S into S_0 and S_1, based on some hypothesis H about the key

   Idea: there is some correlation between traces in S_0 at some point in time if H is correct

   Eg. traces in S_0 compute the same intermediate value at same point in time

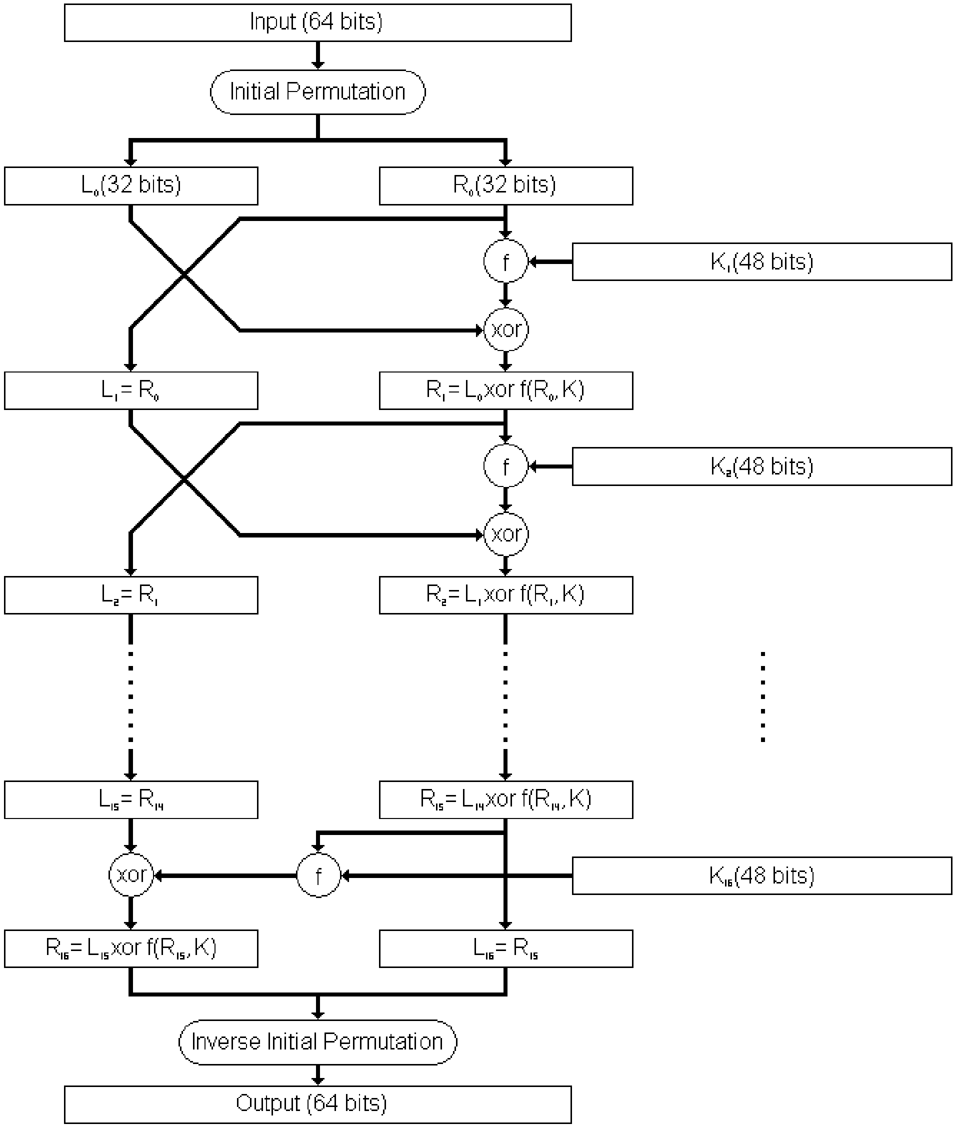
3. Compute differential trace Δ = average of S_0 - average of S_1

4. Now

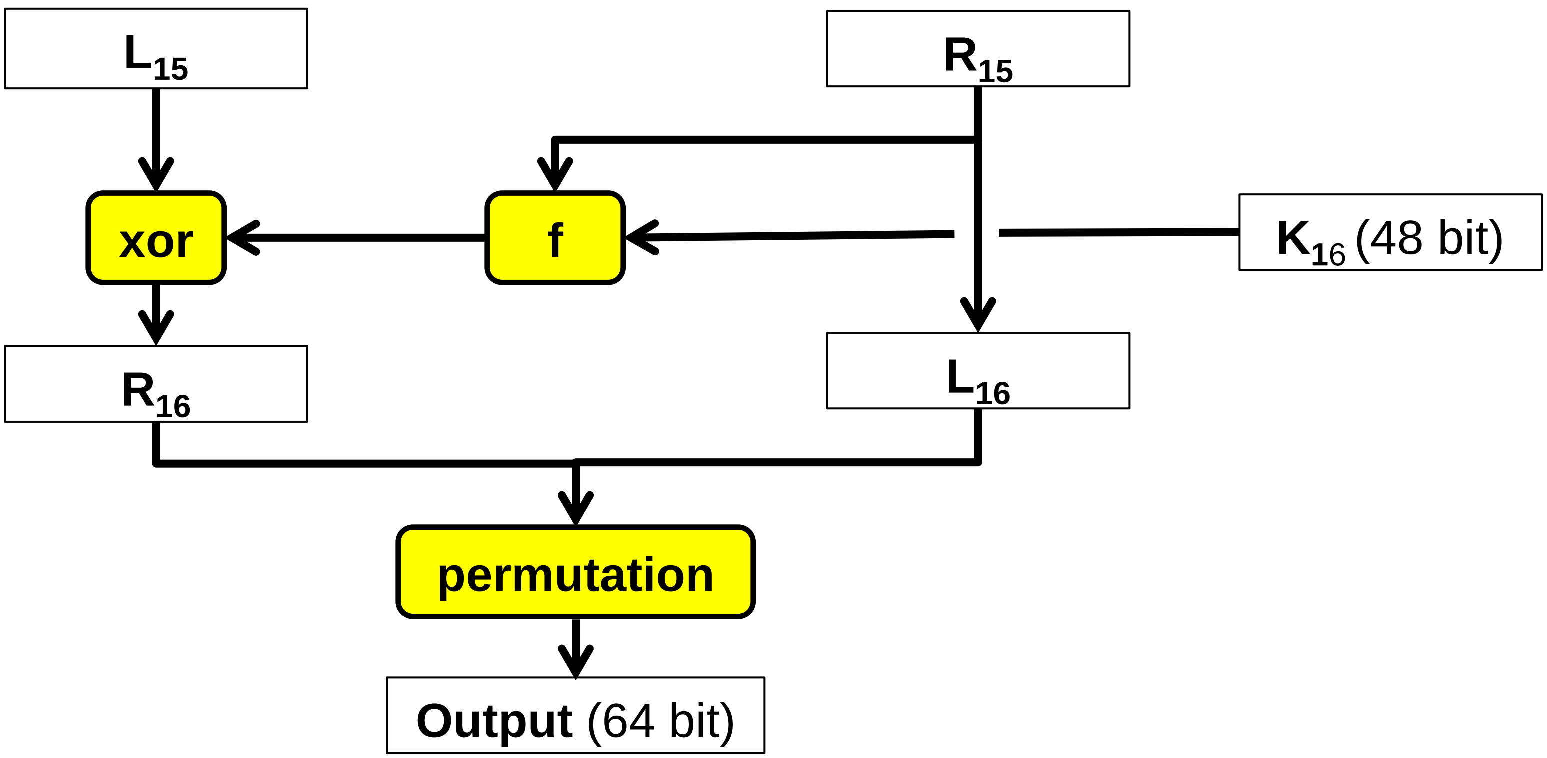
   - if there are blips in Δ then hypothesis H is correct
   - if Δ is flat then hypothesis H was incorrect

We can re-use the same set S for many guesses for H !

DES

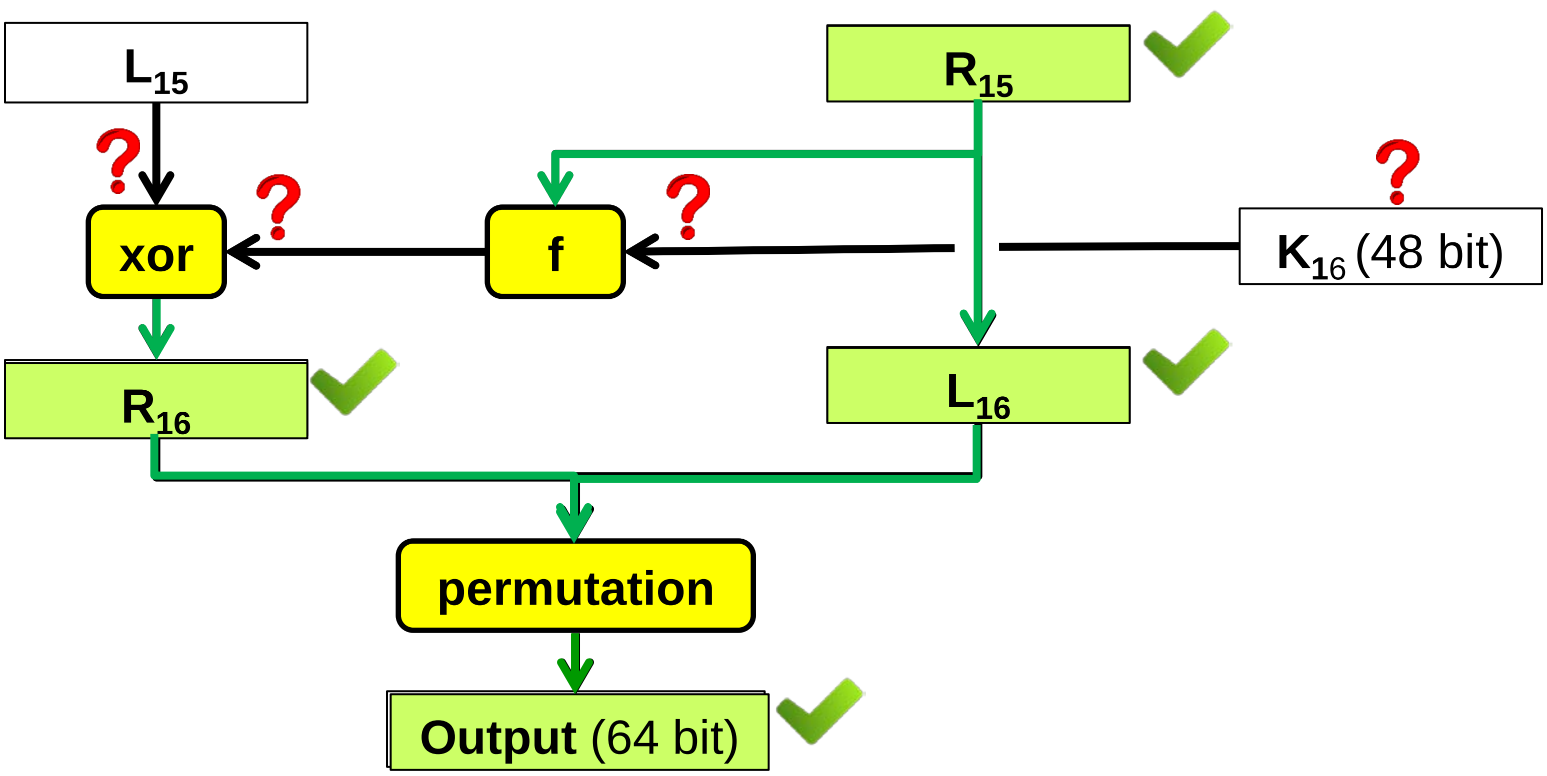

Input (64 bits)
Initial Permutation
L0(32 bits)
R0(32 bits)
f
K1(48 bits)
xor
L1= R0
R1= L0xor f(R0, K)
f
K2(48 bits)
xor
L2= R1
R2= L1xor f(R1, K)
L15= R14
R15= L14xor f(R14, K)
xor
f
K16(48 bits)
R16= L15xor f(R15, K)
L16= R15
Inverse Initial Permutation
Output (64 bits)

DPA attack on DES: the last round of DES



Here f is the function containing the S-boxes.
We can observe the Output, of course. How far can we compute back?

DPA attack on DES: the last round of DES



We can observe the Output, of course. How far can we compute back?

If we guess some value of K_{16}, we know what L_{15} was.

For i=1..32: if we guess 6 bits of K_{16}, we can know what the i^{th} bit in L_{15}

DPA on DES

- We define selection function $D(C,b,K_{sub})$ = value of the b-th bit in L_{15}
  for ciphertext C
  $1 \leq b \leq 32$
  guess K_{sub} for the 6 bits subkey of K_{16} that influence bit b
- So we split the traces in those where bit b in L_{15} is 1 and those where it is 1, for each possible guess for K_{sub}
- Only 2^6 guesses needed K_{sub}; if the splits shows a spike that confirms that guess for K_{sub} was correct.
- Repeating this for 32 bits yields all 32/4*6=56 bits of K_{16}
- Four bits b will depend on the same K_{sub}, which will confirm the correctness of the guess
- Remaining 8 bits can be brute forced or we can do a DPA analysis of the previous round

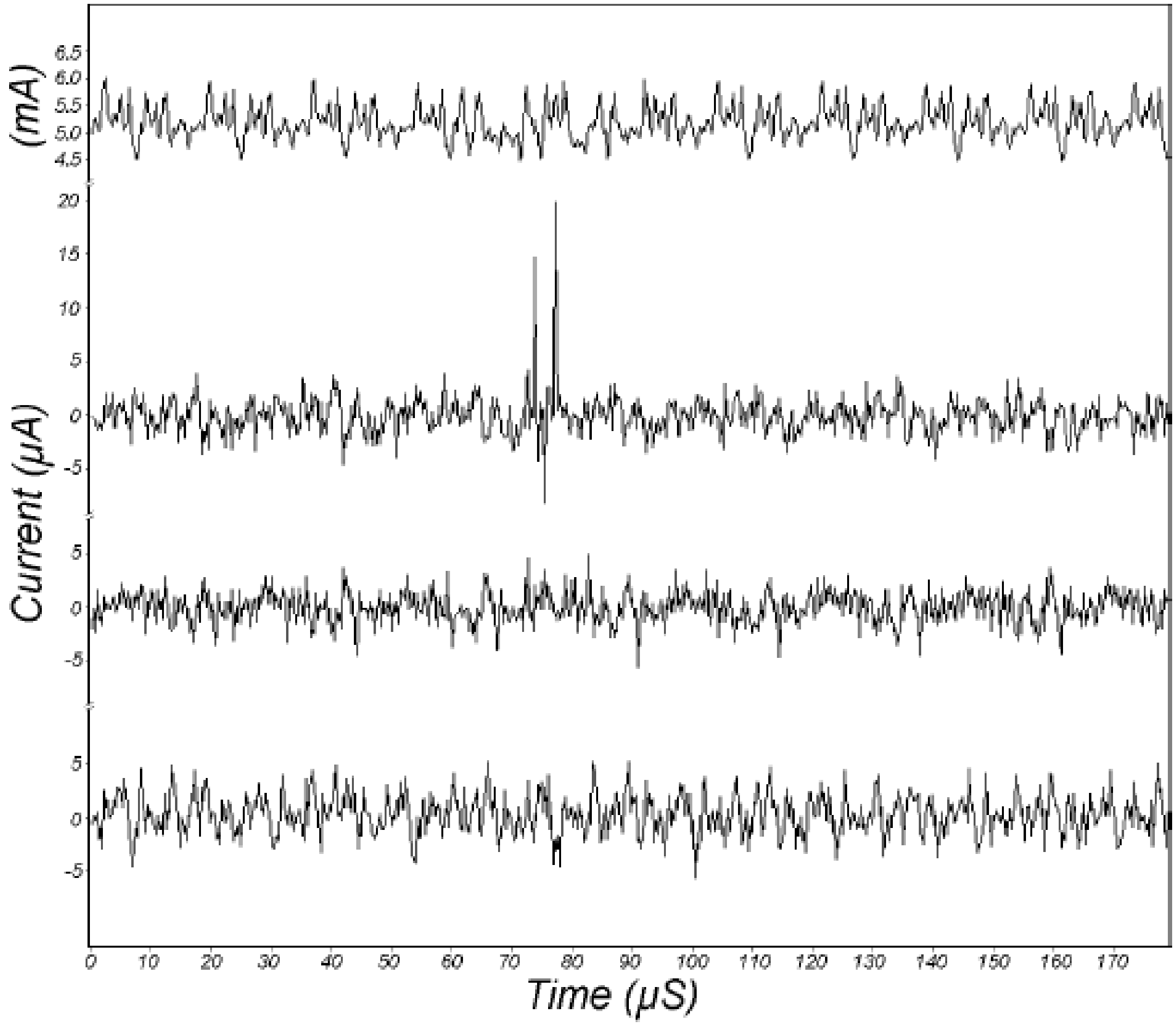
DPA example result

average power consumption

Δ with correct key guess

Δ with incorrect key guess

Δ with another incorrrect key guess



[source: Kocher, Jaffe and Jun, Differential Power Analysis]

DPA on DES

The key idea

We don't have to try all guesses for all possible keys, because the side-channel analysis reveals if guess for a 6-bit sub-key is correct

Instead of having to guess a 56 bit key takes 2^{56} guesses,

but 8 times having to guess a 6-bit key takes only $8 * 2^6 = 2^9$ guesses

DPA

- Obvious countermeasure to all power analysis attacks: add noise to signal

  But DPA is very good at defeating this countermeasure:

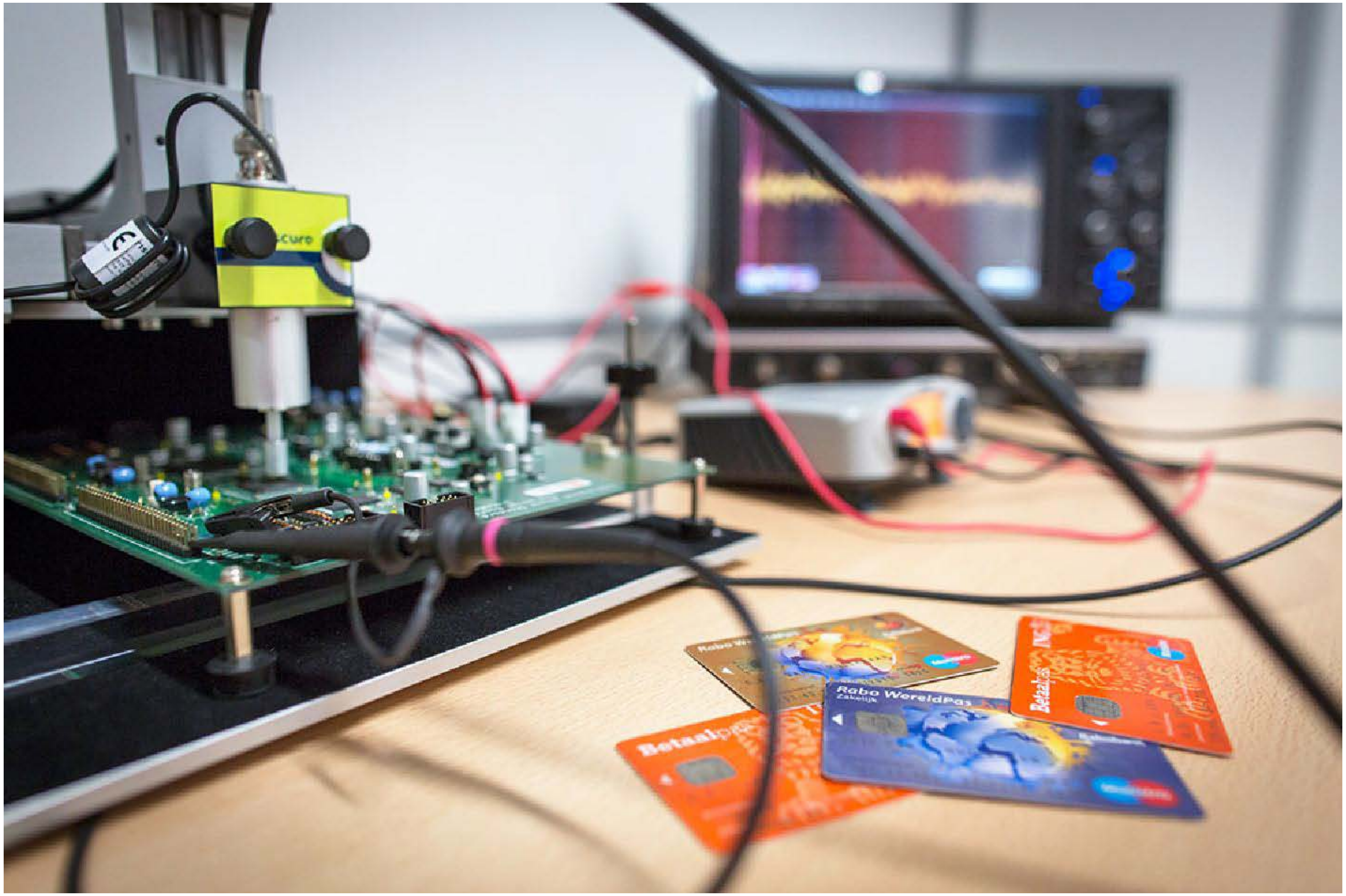
  with enough traces, any random noise can be filtered out

- Technical complications for DPA
  - alligning the traces
  - finding the interesting part of the trace

  Note that this is easier if you can program the smartcard yourself

- Possible countermeasure: masking
  - manipulate secret ⊕ mask instead of secret, for randomly chosen mask
  - now (un)masking may now leak information,
    but this is typically not such an computationally intensive task...

Other side channels: eg EM radiation



Our lab set-up for EM attacks

Conclusions: side-channel analysis

- Resisting side-channel analysis is an ongoing arms race between attackers and defenders
  - with increasingly sophisticated attacks & cleverer countermeasures
- Interesting research question:
  side-channel attacks on obfuscated code, eg. white-box crypto?
- Side channel attacks are a classic example of thinking outside the box

Hacking football!
Penalty by Johan Cruijff
http://www.youtube.com/watch?v=MJHN1mN5SCg

Fault injections

(*active* side-channel attacks)

Fault injections

- So far we discussed *passive* side-channel attacks:
  the attacker monitors some side-channel of the physical hardware

  Passive side-channel attacks threaten confidentiality,
  and typically only target crypto functionality, to retrieve crypto keys

- Side-channel attacks can also be *active*:
  the attacker manipulates physical hardware

  Active attacks threaten integrity – of all software and/or data –
  and can target both crypto- and non-crypto functionality

Example fault attacks with fault injections

- card tears

  removing the card from the reader halfway during a transaction

- glitching

  temporarily dipping the power supply

  Eg to prevent an EEPROM write,

  or to prevent the hardware from representing bits with the value 1

- light attacks

  shoot at the chip with a laser

  To flip some bits...

Laser attacks

Laser mounted on microscope with x-y table to move the card and equipment to trigger timing.

Unlike power analysis, this is tamper-evident

Fault injections: practical complications

Many parameters for the attacker to play with

- *When* to do a card tear?
- *When* to glitch, for *how long?*
- *When* & *where (x and y dimension)* to shoot a laser?
  And for *how long*, *how strong*, and *which* colour laser*?*
- *Multiple faults?*

Multiple glitches are possible, multiple laser attacks harder

This can make fault attacks a hit-and-miss process for the attacker (and security evaluator).

Fault injections: targets

- Attacks can be on data or on code
  - including data and functionality of the CPU, eg the program counter (PC)
- Code manipulation may
  - turn instruction into nop
  - skip instructions
  - skip (conditional) jumps
- Data manipulation may result in
  - special values: 0x00 or 0xFF
  - just random values

Fault injections: targets

Fault attacks can target

- crypto

  Some crypto-algorithms are sensitive to bit flips;
  the classic example is RSA

- any other functionality

  any security-sensitive part of the code or data can be targeted

The smartcard platform (hardware, libraries, and VM) can takes care of some of this, but every programmer has to ensure this for each program separately

Light manipulation

Targets:

- memory
  - targetting RAM: change content or decoding logic to change read out
  - targetting EEPROM: difficult & not tried
- glue logic & CPU
  - unpredictable results
- countermeasures
  - to disable them

Countermeasures?

Physical countermeasures

- Prevention – make it hard to attack a card
- Detection: include a detector that can notice an attack
  - eg a detector for light or a detector for dips in power supply

  This starts another arms race: attackers use another fault attack on such detectors. Popular example: glitch a card and simultaneously use a laser to disable the glitch detector!

Logical countermeasures

- program defensively to resist faults

Example sensitive code: spot the security flaw!

```
class OwnerPIN{
   boolean validated = false;
   short tryCounter = 3;   //number of tries left
   byte[] pin;

boolean check (byte[] guess) {
   validated = false;
   if (tryCounter != 0) {
     if arrayCompare(pin, 0, guess, 0, 4) {
         validated = true;
         tryCounter = 3;
     } else {
         tryCounter--;
         ISOException.throwIt(WRONG_PIN);
    }
   else
     ISOException.throwIt(PIN_BLOCKED);
   }
```

> cutting power at this point will leave **`tryCounter`** unchanged

Example sensitive code: more potential problems?

```
class OwnerPIN{
   boolean validated = false;
   short tryCounter = 3;    //number of tries left
   byte[] pin;

boolean check (byte[] guess) {
   validated = false;
   if (tryCounter != 0) {
     if arrayCompare(pin, 0, guess, 0, 4) {
         validated = true;
         tryCounter = 3;
     } else {
         tryCounter--;
         ISOException.throwIt(WRONG_PIN);
    }
   else
     ISOException.throwIt(PIN_BLOCKED);
   }
```

`validated` flag should be allocated in RAM,not EEPROM

Can timing behaviour of **`arraycompare`** leak info?

can **`ArrayIndexOutOfBounds-Exception`** wrong length **`guess`** leak info?

Defensive coding for OwnerPIN

- Checking & resetting PIN try counter in a safe order
  - to defeat card tear attacks
- **`validated`** should be allocated in RAM and not EEPROM
  - to ensure automatic reset to false
- Does timing of **`arraycompare`** leak how many digits of the PIN code we got right?

  Read the JavaDocs for **`arraycompare`** !
- Can potential **`ArrayIndexOutOfBoundsException`** reveal if we got the first digits of the PIN code right? (Eg by supplying a **`guess`** of length 1.)

(The JavaCard platform provides standard libraries for PIN codes:
you should always use that and not implement your own!)

Getting more paranoid

- checking for illegal values of **`tryCounter`**
  - eg negative values or greater than 3
- redundancy in data type representation
  - eg record **`tryCounter*13`**

    or use an error-detecting/correcting code
- keeping two copies of **`tryCounter`**
- even better?: keep one of these copies in RAM
  - initialised on applet selection

attacker must attack both RAM & EEPROM synchronously

- doing security sensitive checks twice

Secure order of branches?

```
if (pinOK)  { // perform some security-critical task
                ...
            }
       else { // error handling
                ...
            }
```

Better

```
if (!pinOK) { // error handling
                 ...
            }
       else { // perform some security-critical task
                 ...
            }
```

Better to branch (conditionally jump) to the "good" (ie "dangerous") case

if faults can get the card to skip instructions

Even more paranoid

```
if (!pinOK) { // error handling
                   ...
            }
       else { if (pinOK) {
                 ...
              }
              else {
                // We are under attack!
                // Start erasing keys
                 ....
            }
```

An attacker observing the power trace can tell when countermeasure is triggered, and then cut the power

And more paranoid still

```
if (!pinOK) { // error handling
                    ...
            }
       else { if (pinOK) {
                  ...
              }
              else {
                // We are under attack!
                // Set a flag and start erasing keys
                // some time in the future
                  ....
            }
```

Defensive coding tricks

- avoiding use of special values such as 00 and FF
  - don't use C or JavaCard booleans!
- use restricted domains and check against them
  - ideally, domains that exclude 00 and FF, and elements with equal Hamming weights
- introduce redundancy
  - when storing data and/or performing computations
  - forcing attacker to synchronise attacks or combine different attacks (eg on EEPROM *and* RAM)
- jump to good (ie dangerous) cases
- make sure code executes in constant time

Defensive coding tricks

- additional integrity checks on execution trace
  - doing the same computation twice & checking results
  - for asymmetric crypto: use the cheap operation to check validity of the expensive one
- check control flow integrity
  - add ad-hoc trip-wires & flags in the code to confirm integrity of the run

    eg set bits of a boolean at various points in the code, and check in the end if all are set

People have proposed beginSensitive() and endSensitive() API calls for the JavaCard smartcardplatform to turn on VM countermeasures

Physical attacks

Physical attacks

- Much more costly than logical or side channel attacks.
  - expensive equipment
  - lots of time & expertise
- Also, you destroy a few chips in the process:
  - –they are invasive attacks, and tamper-evident
- Some forms are largely historic, as modern chips are too small and complex to analyse/
- Examples: probing, fibbing, reading memory contents

Smartcard attacks: cost

Logical attacks - ie. look for flaws in protocols or software

- Only 50$ of equipment, but possibly lots of brain power!
- Analysis may take weeks, but final attack can be in real time

Side channel attacks (SPA, DPA)

- 5K$ of equipment
- Again, lots of time to prepare, but final attack can be quick

Physical attacks

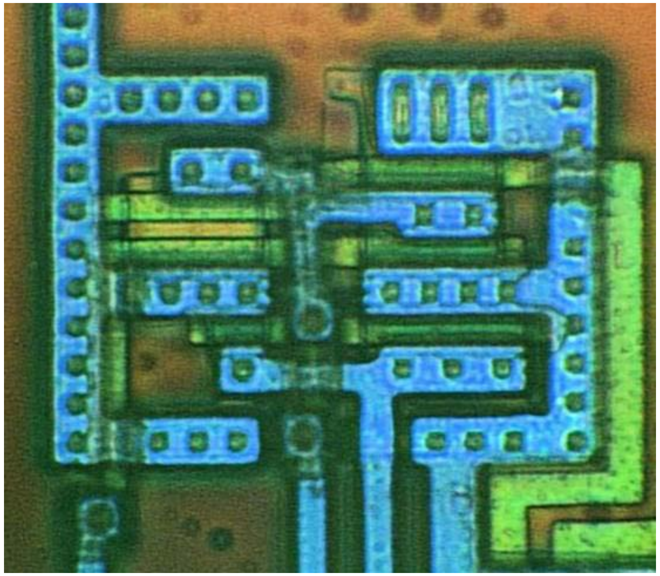
- 100K$
- Several weeks to attack a single card, and attack is tamper-evident

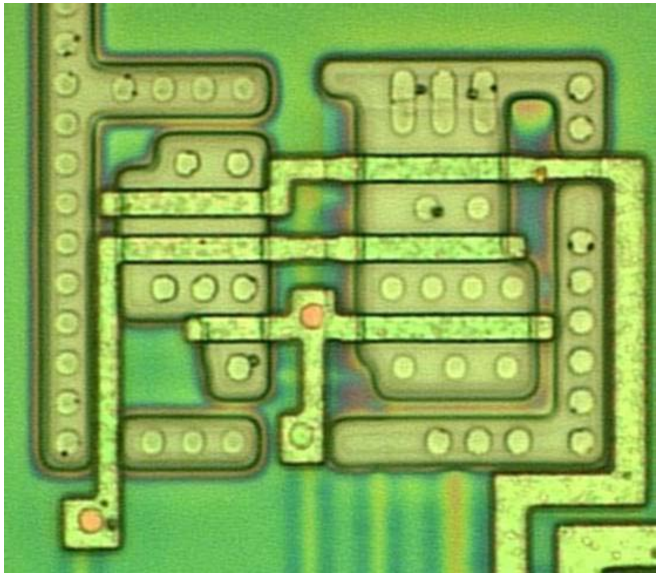
First step: removing chip from smartcard

using heat & nitric acid

[Source: Oliver Kömmerling, Marcus Kuhn]

Optical reverse engineering





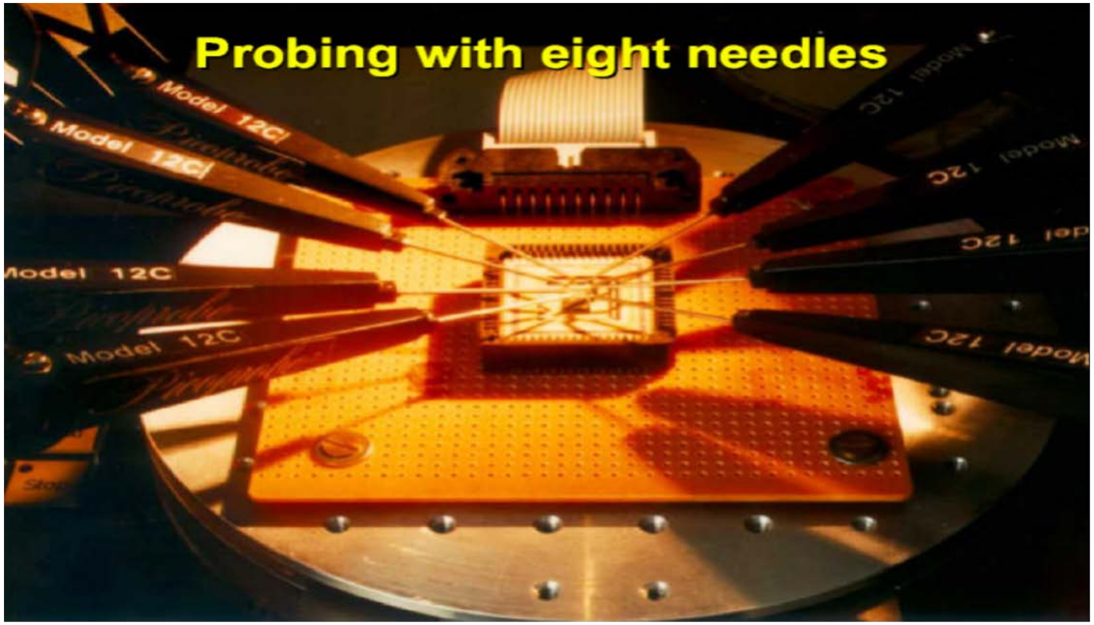
microscope images with different layers in different colours, before and after etching

[Source: Oliver Kömmerling, Marcus Kuhn]

Physical attack: probing

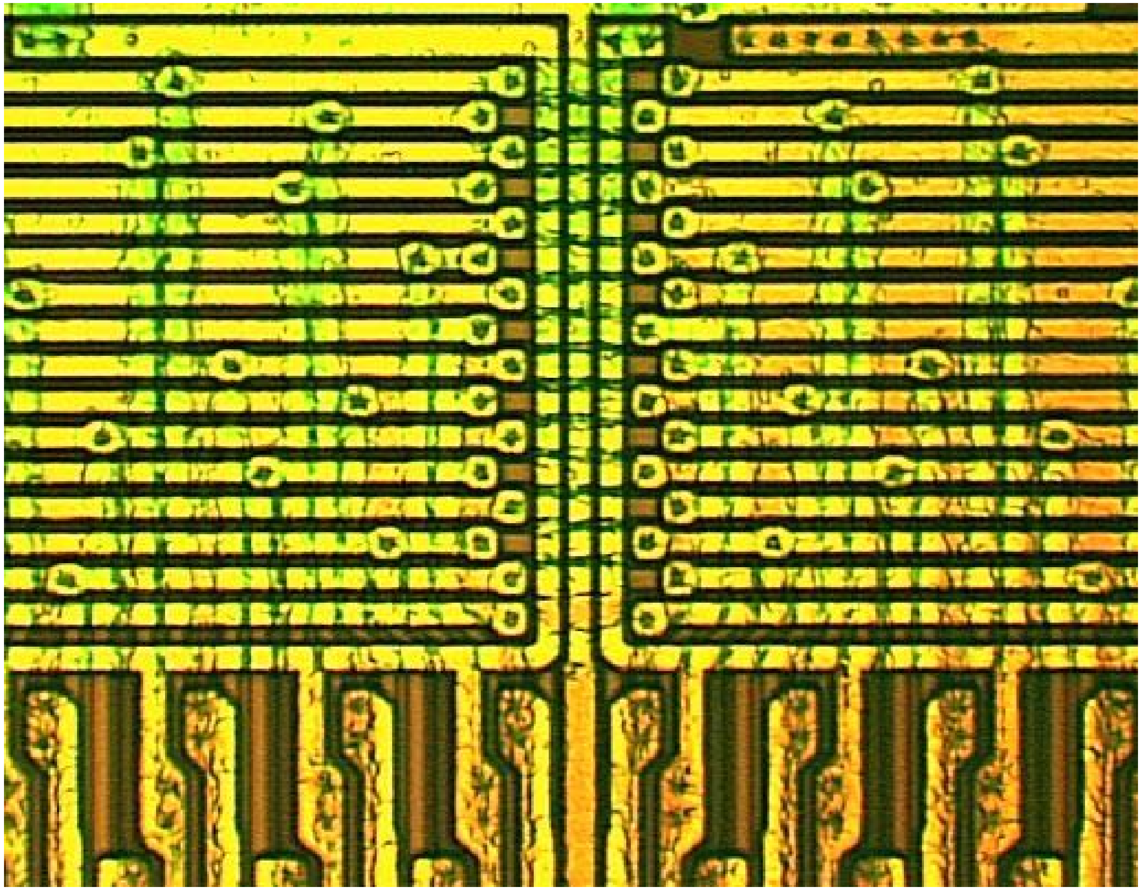
Observe or change the data on the bus while the chip is in operation eg to observe keys

probing with
8 needles



Probing can be done with physical needles (>0.35 micron)
or electron beam

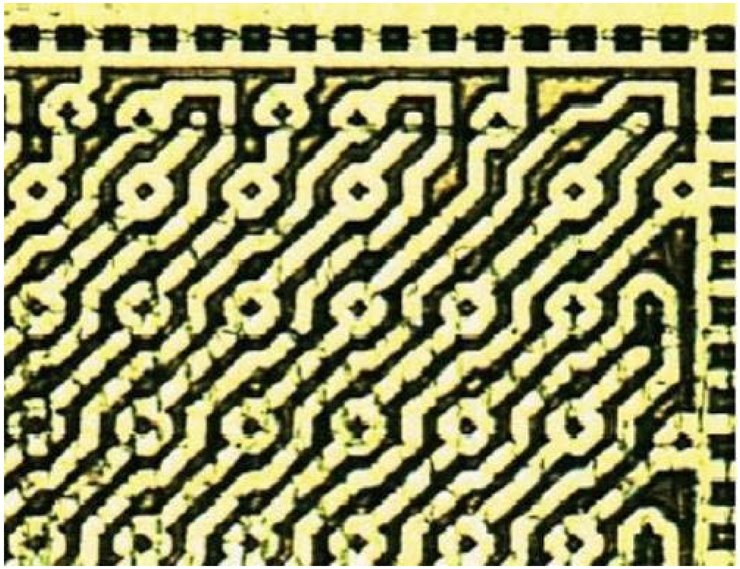
Visual reconstruction of bus permutation

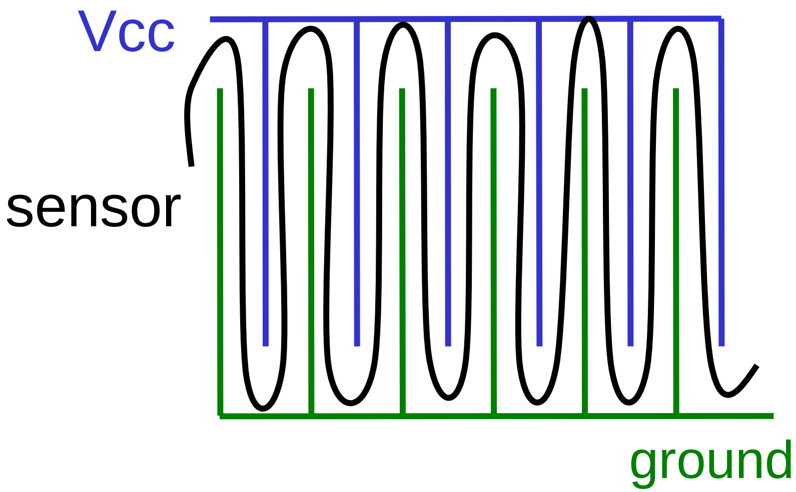


[Source: Oliver Kömmerling, Marcus Kuhn]

Protective sensor mesh

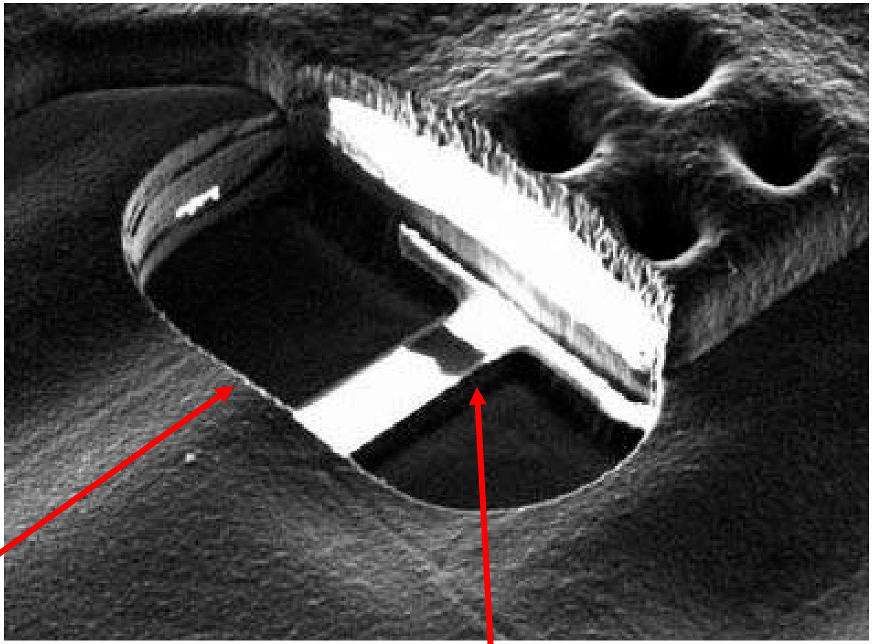
- sensor line checked for interruptions or short-circuits, which trigger alarm: halt execution or erase memory
- but.. external power supply is needed to eg. erase persistent memory
- attacker will fingerprint active countermeasures (eg by power consumption) to interrupt power supply in time





Physical attack: probing

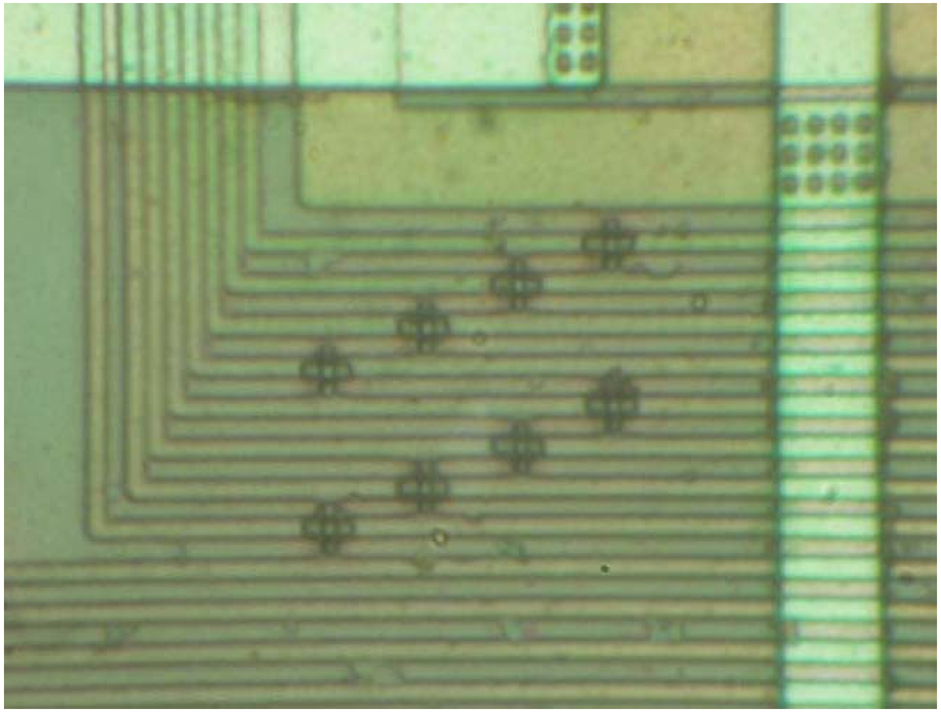
- FIB = Focussed Ion Beam
- can observe or modify chip by
- drilling holes
- cutting connections
- soldering new connections and creating new gates



hole drilled in the chip surface

blown fuse

Using FIB in probing



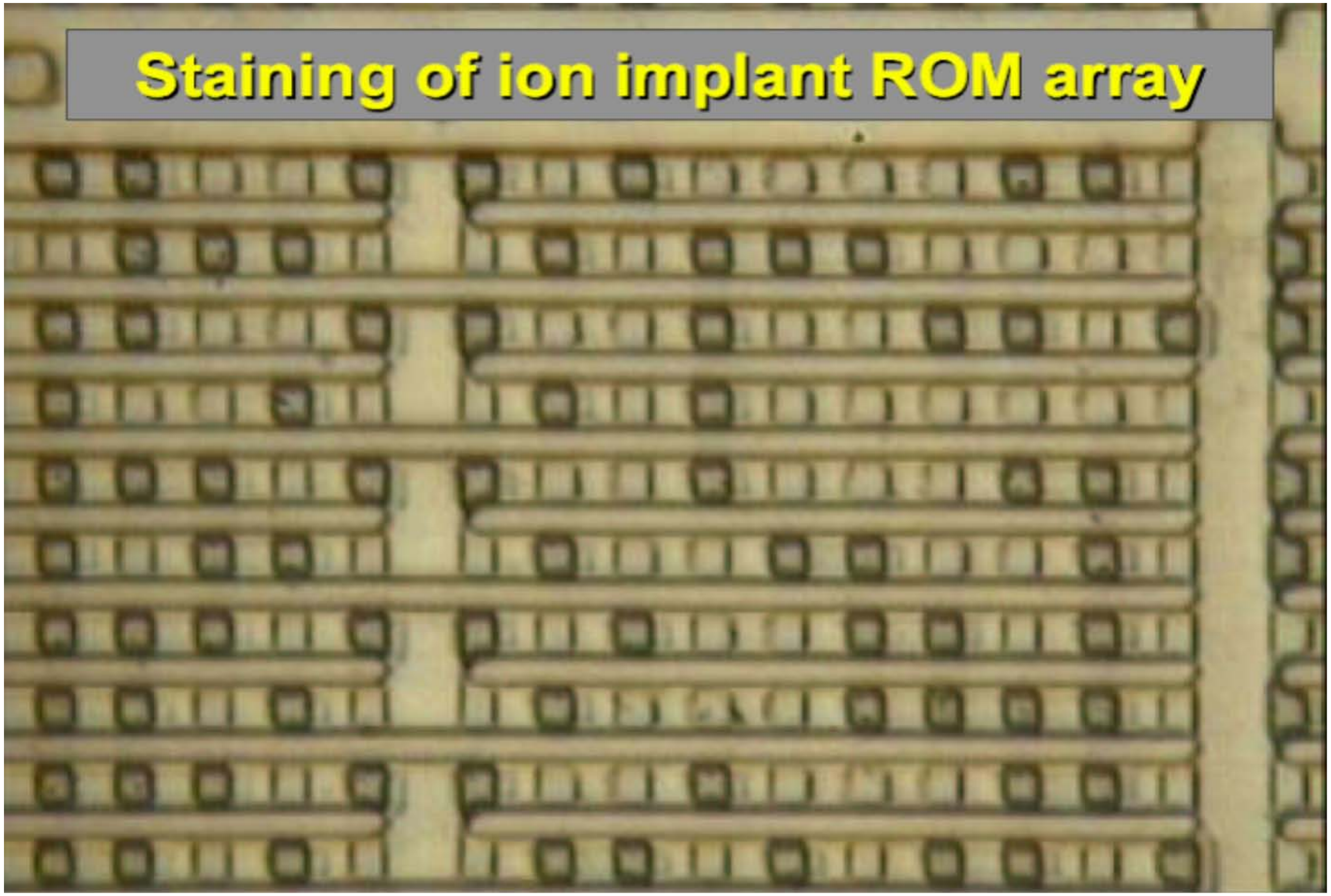
[Source: Sergei Skorobogatov]

Fibbing can be used to

- add probe pads
- for lines too thin or fragile for needles
- surface buried lines
  - –poking holes through upper layers

Physical attack: extracting ROM content

Staining can optically reveal the bits in ROM: dark squares are 1 light squares are 0



[Source: Brightsight]

Physical attack: extracting RAM content

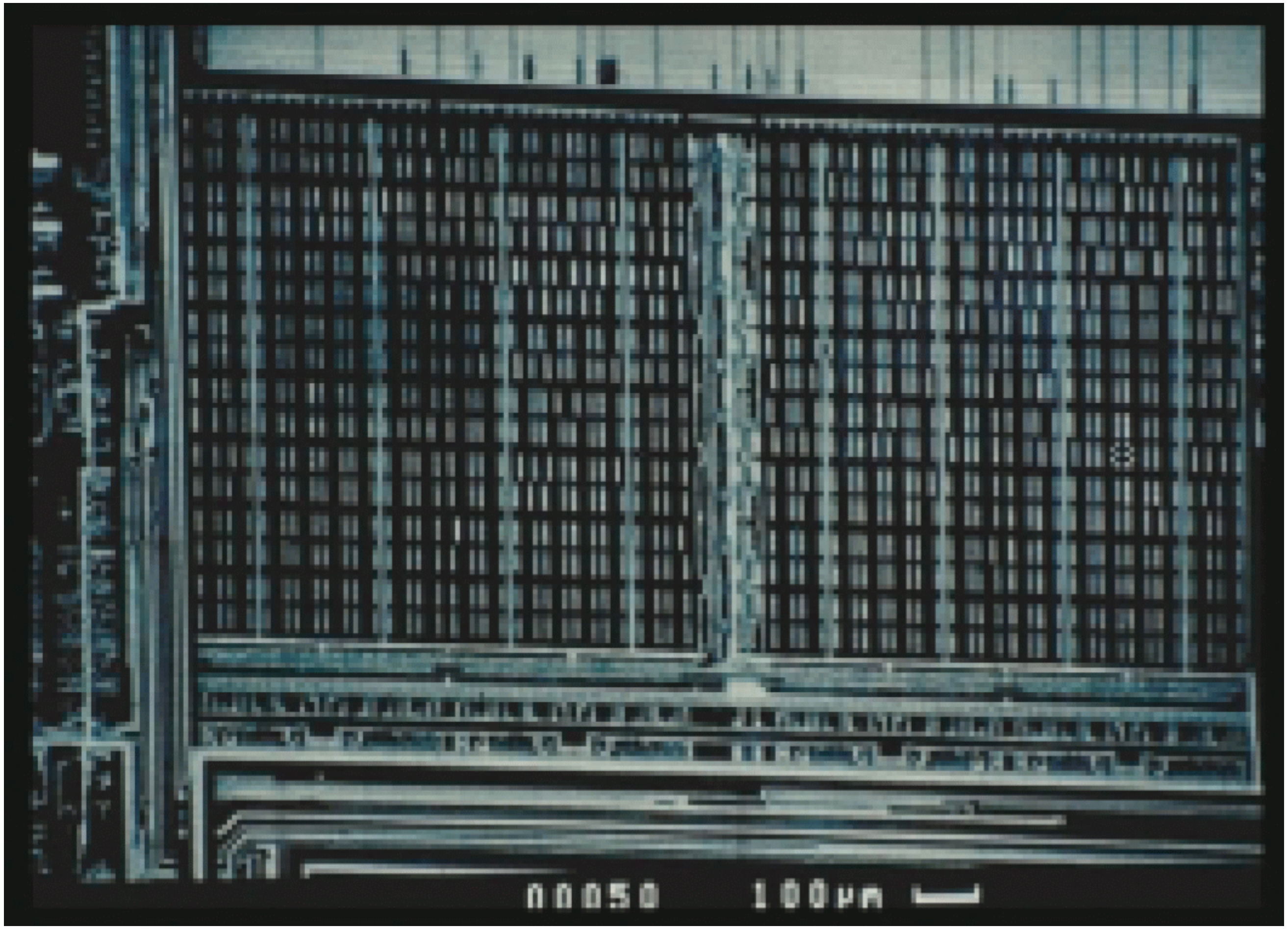
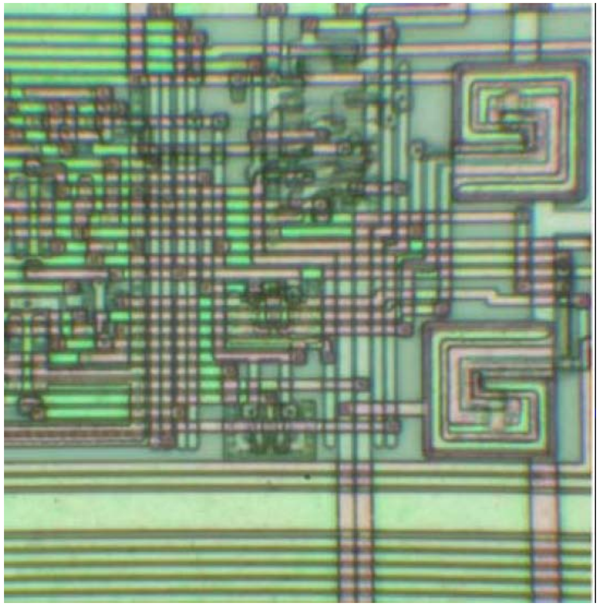
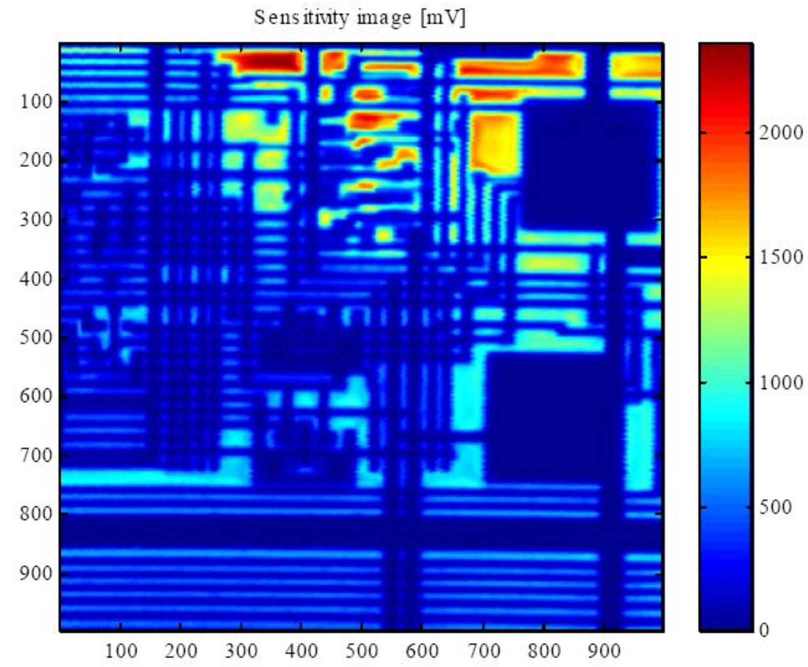


Image of RAM with voltage sensitive scanning electron microscope

Newer imaging techniques





[Source: Sergei Skorobogatov]

Optical Beam Induced Current

- signalling hot spots on chip surface
  - eg to locate crypto co-processor

Physical attacks: countermeasures

- protective mesh to prevent access to the chip surface
- obfuscate chip layout, eg by scrambling or hiding bus lines
- scramble or encrypt memo
- sensors for low and high temperatures, light, clock frequency, voltage, to trigger active countermeasure
  - But… external power supply needed for reaction
  - Sensors can be destroyed when power is off => they must be tested periodically in normal operation

- The good news: as circuits become smaller & more-complex, physical attacks become harder … ultimately too hard?

Dynamic security analysis:

Fuzzing & automated reverse engineering

Accidental DoS attacks over the years

All *accidentally* made to crash with unexpected inputs

Fuzzing: different forms & case studies

1. original form of fuzzing: trying out long inputs to find buffer overflows

2. message *format* fuzzing:
   trying out strange inputs, given some format/language

3. message *sequence* fuzzing
   trying out strange sequences of inputs

   to infer the protocol state machine from an implementation

Message Format Fuzzing

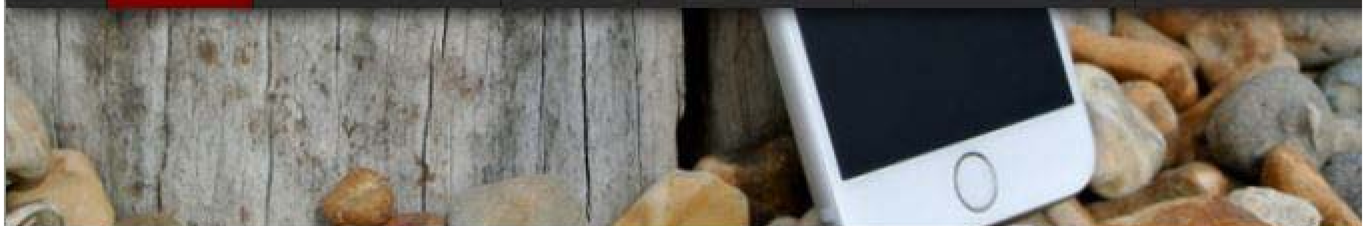
aka Protocol Fuzzing

Fuzzing some protocol/format/protocol/input language

Example: GSM

Input problems

NEWS INTERVIEWS TECH BUSINESS SOCIAL MEDIA DIGITAL MED



iPhone /

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

By Jacques Coetzee: Staff Reporter on 28 May, 2015

The scariest line of characters on the internet right now makes absolutely no sense, but will crash your iPhone nonetheless.

All input is potentially evil!

Scan This Guy's E-Passport and Watch Your System Crash

By Kim Zetter 08.01.07

RFID expert Lukas Grunwald says e-passport readers are vulnerable to sabotage.
Photo: Courtesy of Kim Zetter

A German security researcher who demonstrated last year that he could clone the computer chip in an electronic passport has revealed additional vulnerabilities in the design of the new documents and the inspection systems used to read them.

Lukas Grunwald, an RFID expert who has served as an e-passport consultant to the German parliament, says the security flaws allow someone to seize and clone the fingerprint image stored on the biometric e-passport, and to create a specially coded chip that attacks e-passport readers that attempt to scan it.

Grunwald says he's succeeded in sabotaging two passport readers made by different vendors by cloning a passport chip, then modifying the JPEG2000 image file containing the passport photo. Reading the modified image crashed the readers, which suggests they could be vulnerable to a code-injection exploit that might, for example, reprogram a reader to approve expired or forged passports.

Malformed image on a electronic passport could crash passport readers.

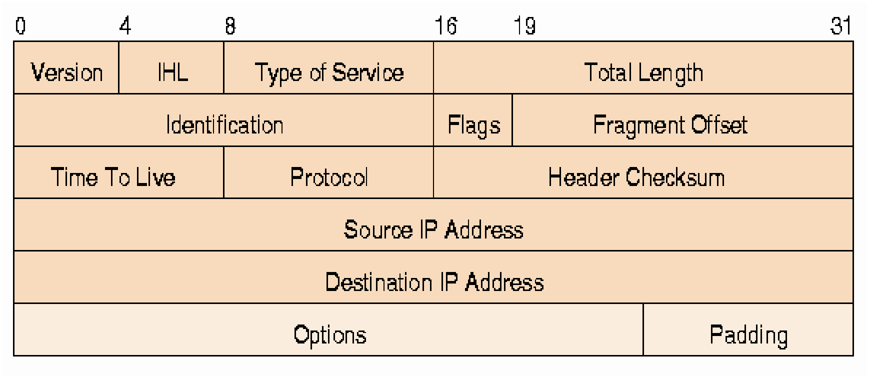
Moral:
Beware of *all* inputs

not just the obvious ones that come over the network…

Message format aka protocol fuzzing

Fuzzing based on a known protocol format

ie format of packets or messages



Typical things to try in protocol fuzzing:

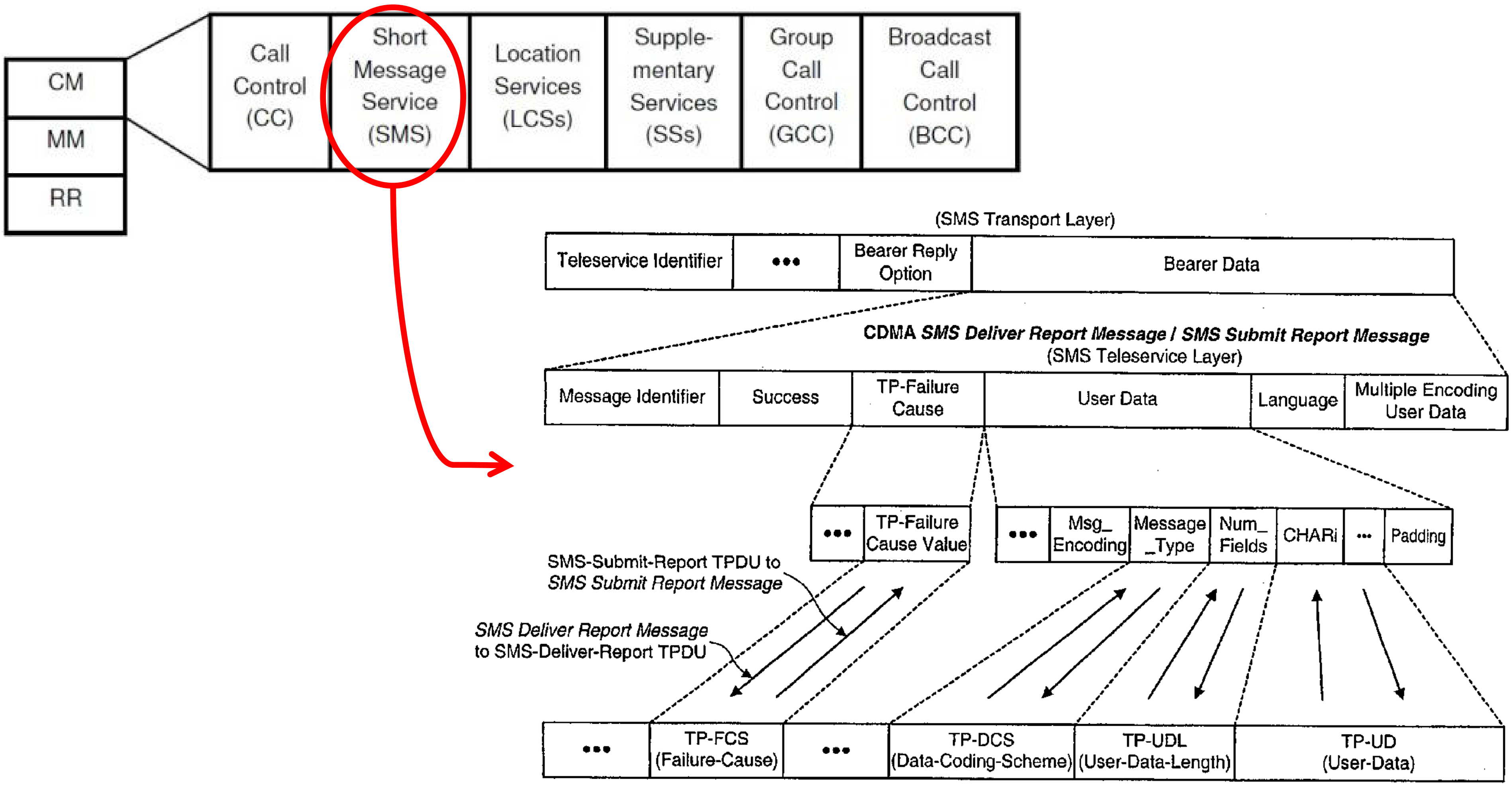
- trying out many/all possible value for specific field
  esp undefined values, or values Reserved for Future Use (RFU)
- giving incorrect lengths, length that are zero, or payloads that are too short/long

Note the relation with LangSec: a good description of the input language is not just useful to generate parsers, but also for fuzzing!

Tools for protocol fuzzing: SNOOZE, Peach, Sulley

Example : GSM protocol fuzzing

GSM is a extremely rich & complicated protocol



SMS message fields

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

Example: GSM protocol fuzzing

Lots of stuff to fuzz!

We can use a USRP

with open source cell tower software (OpenBTS)

to fuzz any phone

[Mulliner et al., SMS of Death]

[F van den Broek, B. Hond, A. Cedillo Torres, Security Testing of GSM Implementations]

Example: GSM protocol fuzzing

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

Example: GSM protocol fuzzing

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

- eg possibility to send faxes (!?)

you have a fax!

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

Example: GSM protocol fuzzing

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

garbage SMS text

garbage SMS text, incl.names of games installed on the phone

Example: GSM protocol fuzzing

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

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

LangSec (Language-theoretic security)

Fuzzing file or protocol formats naturally fits with LangSec.
LangSec recognizes the role of input languages, esp.

- complexity and variety of input languages
- poor (informal) specification of input languages
  (not with eg EBNF grammar)
- ad-hoc, handwritten code for parsing, which mixes parsing & interpreting
  (aka shotgun parsers)

as root causes behind software insecurity

If you're interested in producing secure software, read up on this at langsec.org

Eg

Towards a formal theory of computer insecurity httpp://ww.youtube.com/watch?v=AqZNebWoqnc

CCC presentation The Science of Insecurity http://www.youtube.com/watch?v=3kEfedtQVOY

Automated Reverse Engineering

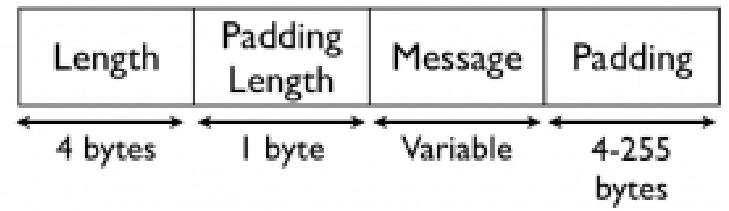
by learning Protocol State Machines

Case studies 

& 

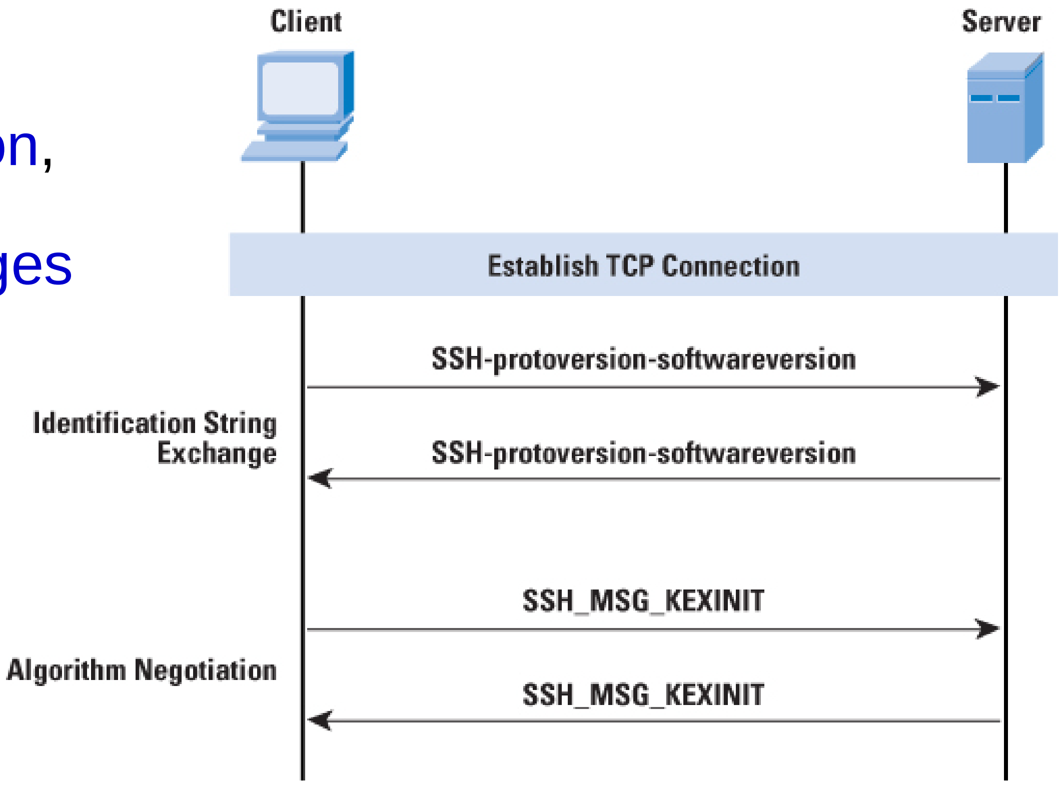
Input languages: messages & sessions

Most protocols not only involves nottion of of input message format



but also language of session,

ie. *sequences* of messages



message sequence chart

1. $C \rightarrow S$: CONNECT 2. $S \rightarrow C$: VERSION_S server version string 3. $C \rightarrow S$: VERSION_C client version string	protocol identification
4. $S \rightarrow C$: SSH_MSG_KEXINIT I_C 5. $C \rightarrow S$: SSH_MSG_KEXINIT I_S	key exchange algorithm negotiation
6. $C \rightarrow S$: SSH_MSG_KEXDH_INIT e where $e = g^x$ for some client nonce x 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	key exchange
10. ...	session, incl. SSH authentication and connection protocols

This *oversimplifies*

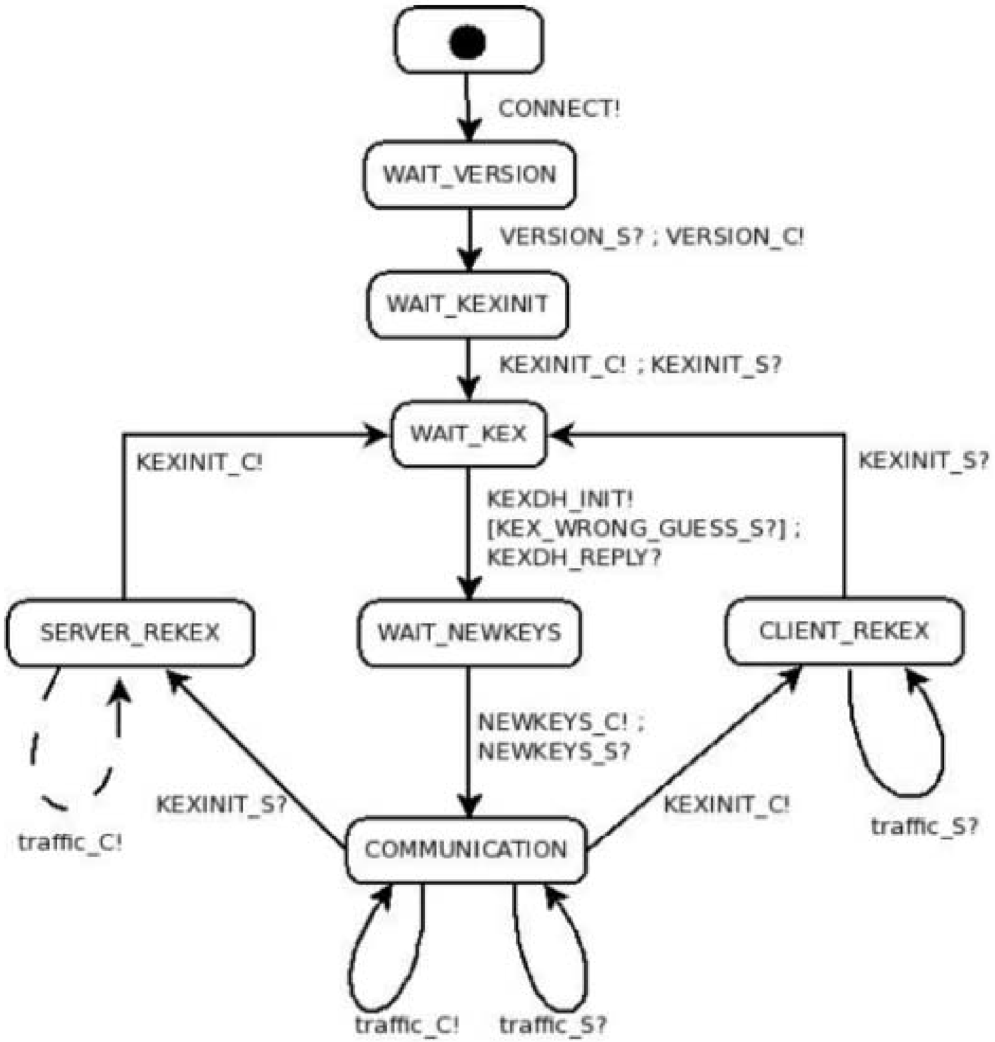
because it only specifies *one correct, happy flow*

protocol state machine

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

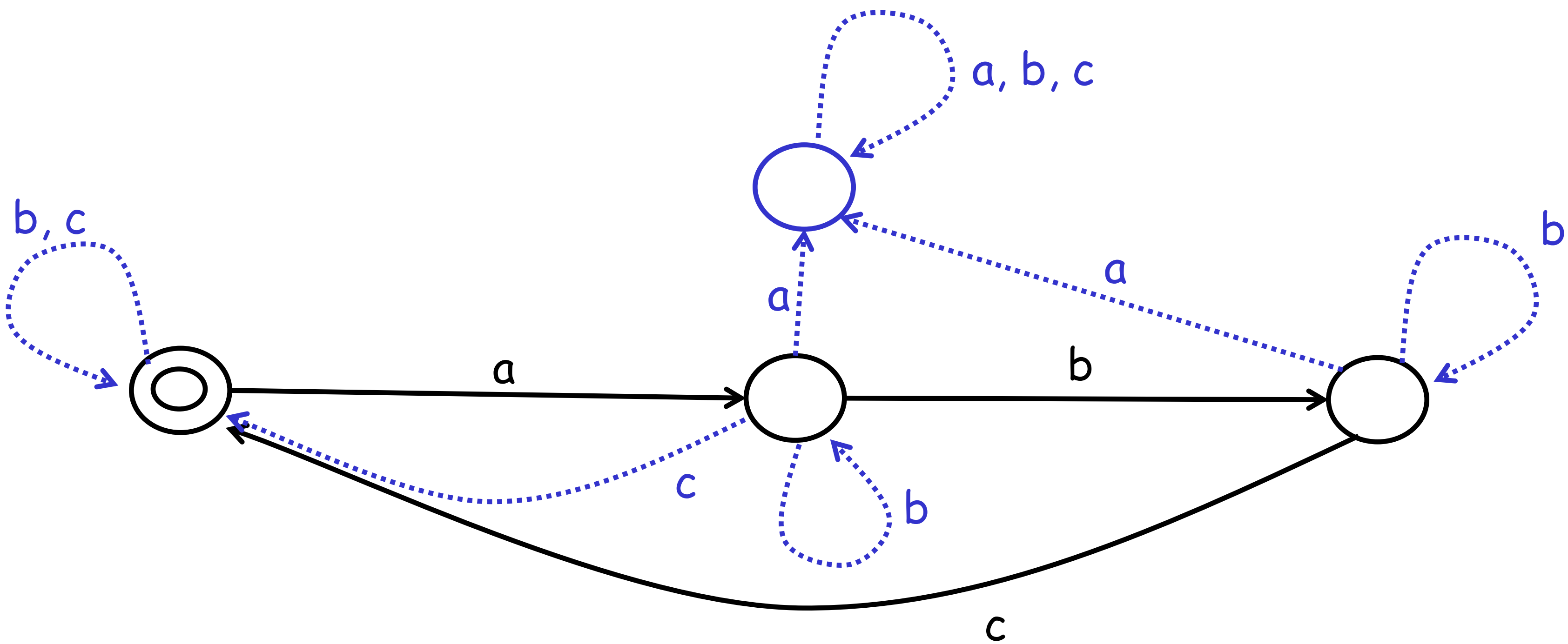
This can be nicely specified using a finite state machine (FSM)

This still oversimplifies: it only describes the happy flows and the implementation will have to be input-enabled



SSH transport layer

input enabled state machine



A state machine is input enabled if in *every* state it is able to receive *every* message

Often, unexpected messages are ignored (eg b above)
or lead to some error state (eg as a above)

Extracting protocol state machine from code

We can infer a finite state machine from implementation

by black box testing using state machine learning

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

This is effectively a form of 'stateful' fuzzing

using a test harness that sends typical protocol messages

This is a great way to obtain protocol state machine

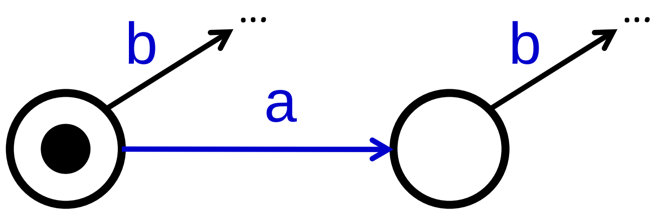
- without reading specs!
- without reading code!

State machine learning with L*

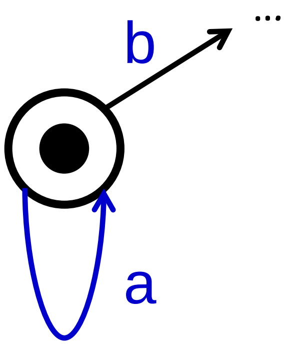
Basic idea: compare response of a deterministic system to different input sequences, eg.

1. b
2. a ; b

If response is different, then



otherwise



The state machine inferred is only an *approximation* of the system,

and *only as good as your set of test messages*

Case study in state machine learning (1):

EMV smartcards for banking

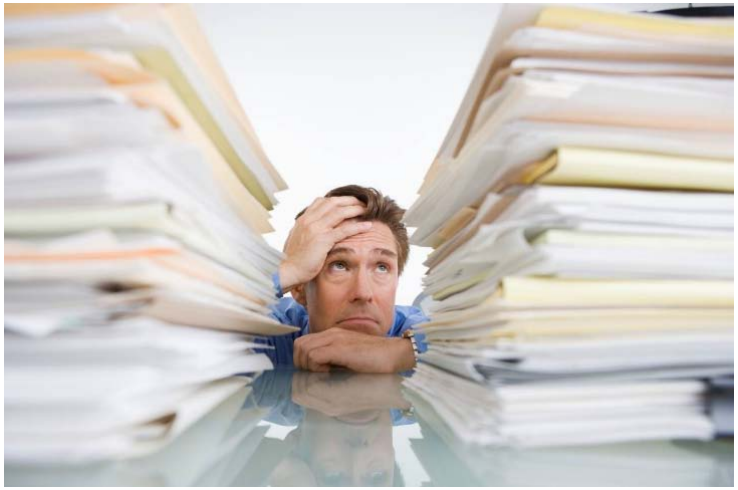
Case study: EMV

- Started 1993 by EuroPay, MasterCard, Visa
- Common standard for communication between
  - smartcard chip in bank or credit card (aka ICC)
  - terminal (POS or ATM)
  - issuer back-end
- Specs controlled by EMVCo which is owned by
- Over 1 billion cards in use

The EMV protocol suite

- EMV is not a protocol, but a "protocol toolkit suite":
  many options and parameterisations (incl. proprietary ones)
  - 3 different card authentication mechanism
  - 5 different cardholder verification mechanisms
  - 2 types of transactions: offline, online

  All these mechanisms again parameterized!



- Specs public but very complex (4 books, totalling >750 pages)

More complexity still: EMV variants

- EMV-CAP, for using EMV cards for internet banking

  Proprietary and secret standard of MasterCard.

- Contactless EMV for payments with contactless bankcard or NFC phone

  Specs public, 10 documents totalling >1600 pages

Visa payWave

Google wallet

MasterCard paypass

Pay

Example: one sentence from these 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.”

[Thanks to Jordi van Breekel for this example]

Test harness for EMV

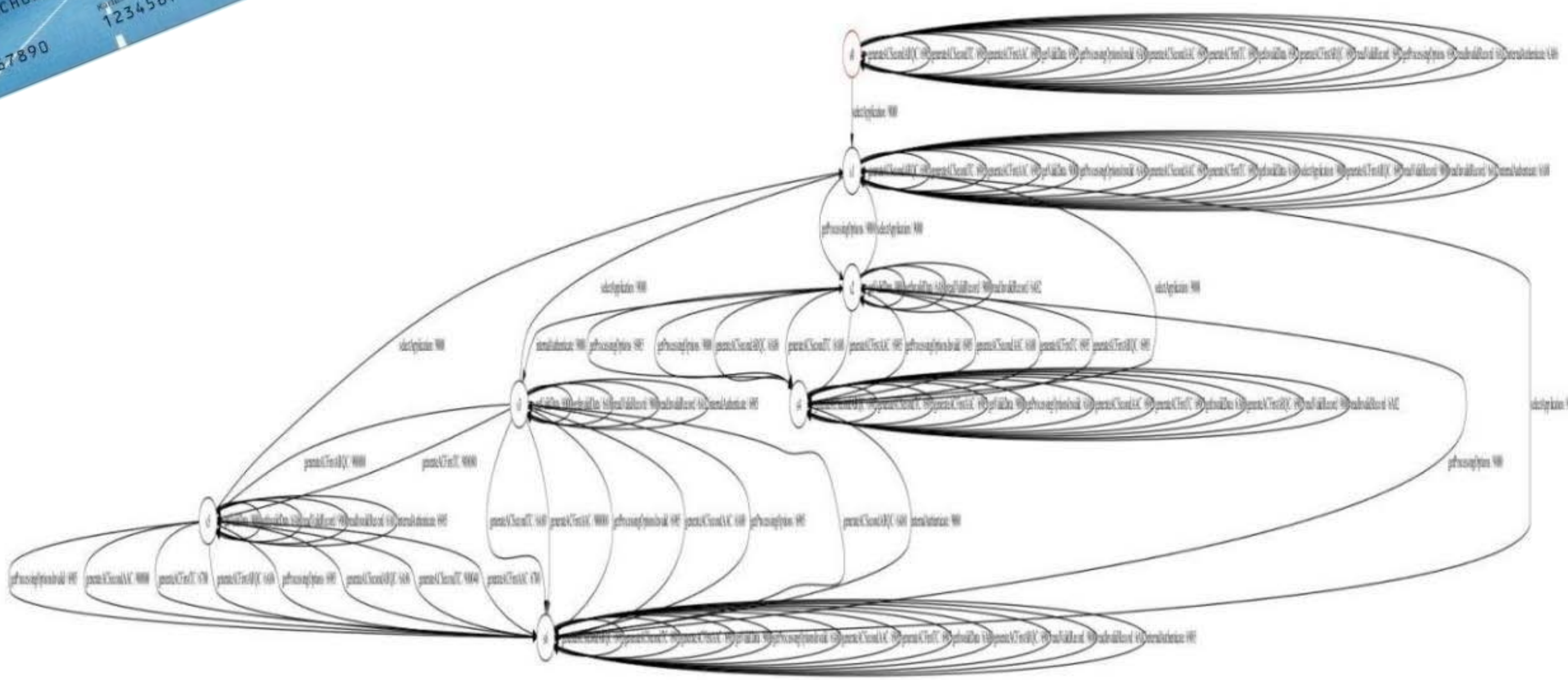
Our test harness implements standard EMV instructions, eg

- SELECT (to select application)
- INTERNAL AUTHENTICATE (for a challenge-response)
- VERIFY (to check the PIN code)
- READ RECORD
- GENERATE AC (to generate application cryptogram)

LearnLib then tries to learn all possible combinations

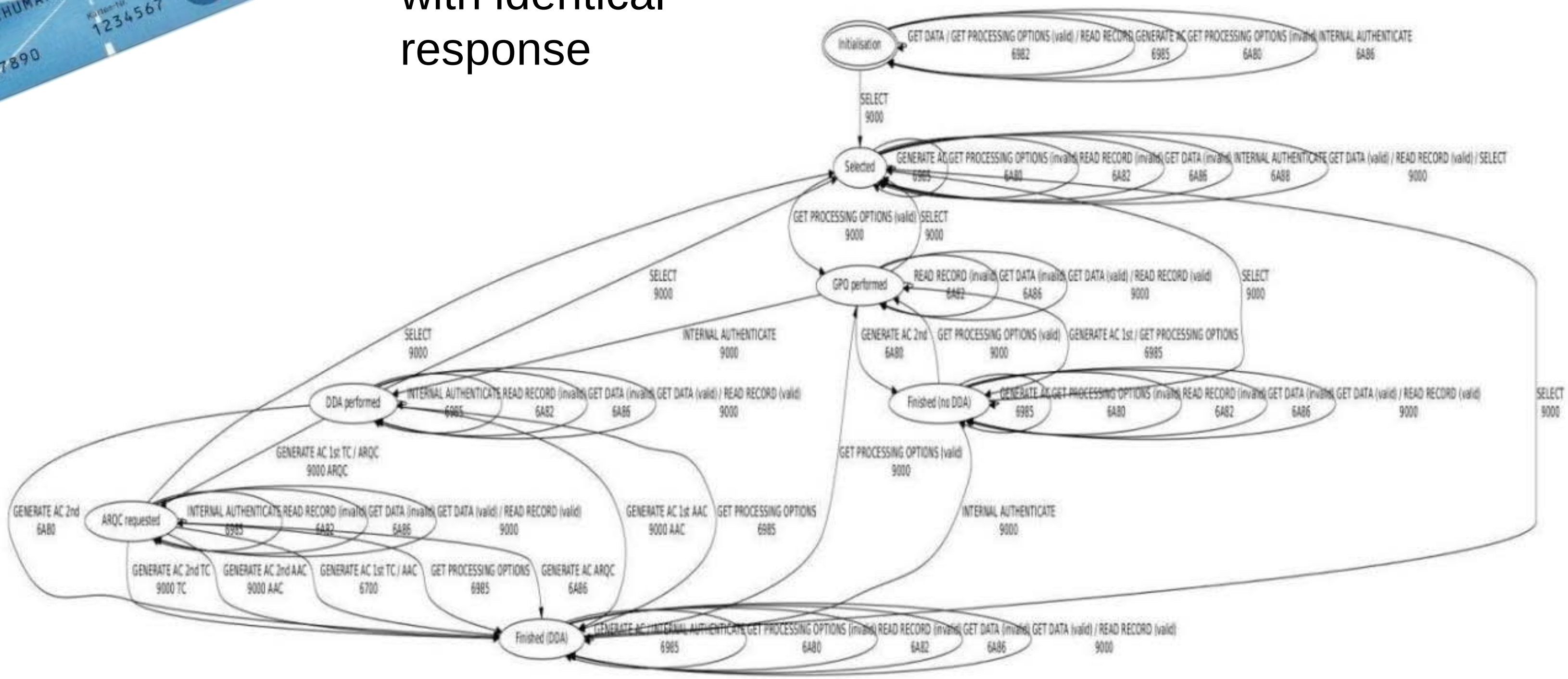
- most commands with fixed parameters, but some with different options

State machine learning of Maestro card

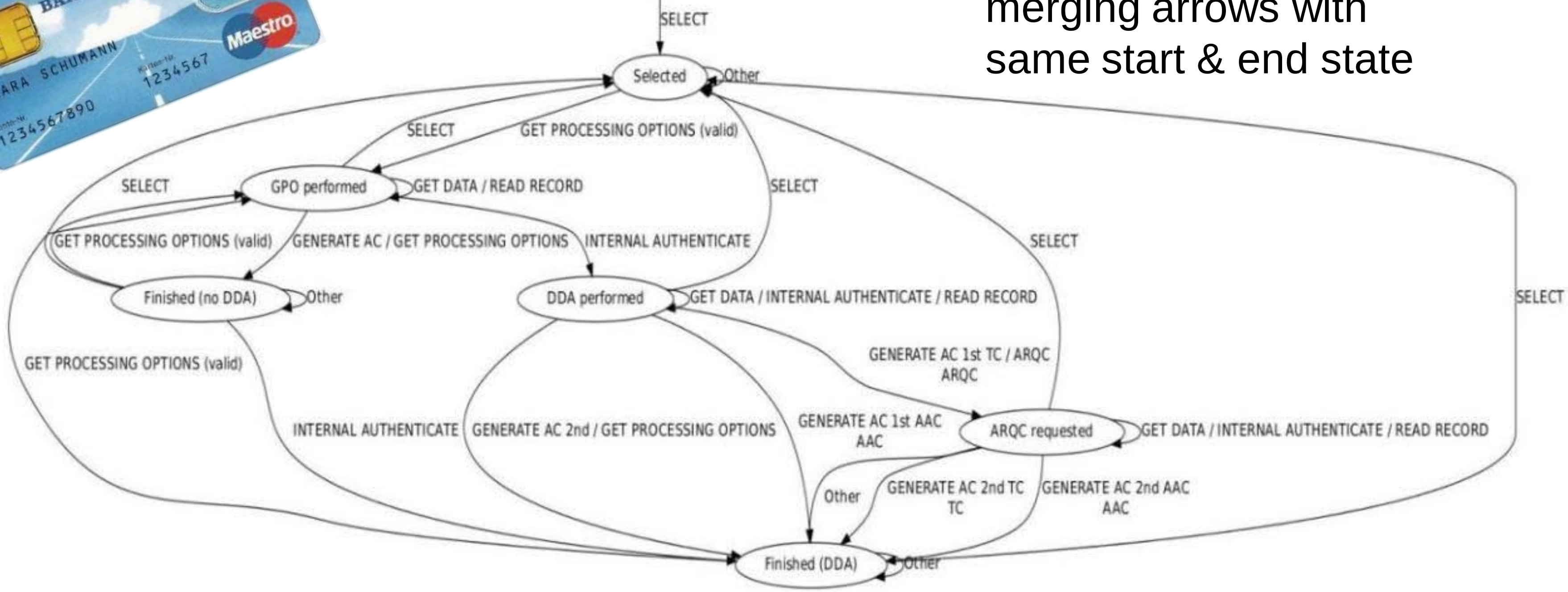


State machine learning of Maestro card

merging arrows
with identical
response



State machine learning of Maestro card



merging arrows with
same start & end state

Formal models of banking cards for free!

- Experiments with Dutch, German and Swedish banking and credit cards
- Learning takes between 9 and 26 minutes
- Editing by hand to merge arrows and give sensible names to states
  - this could be automated
- Limitations
  - We do not try to learn response to incorrect PIN as cards would block...
  - We cannot learn about one protocol step which requires knowledge of card's secret 3DES key
- No security problems found, but interesting insight in implementations

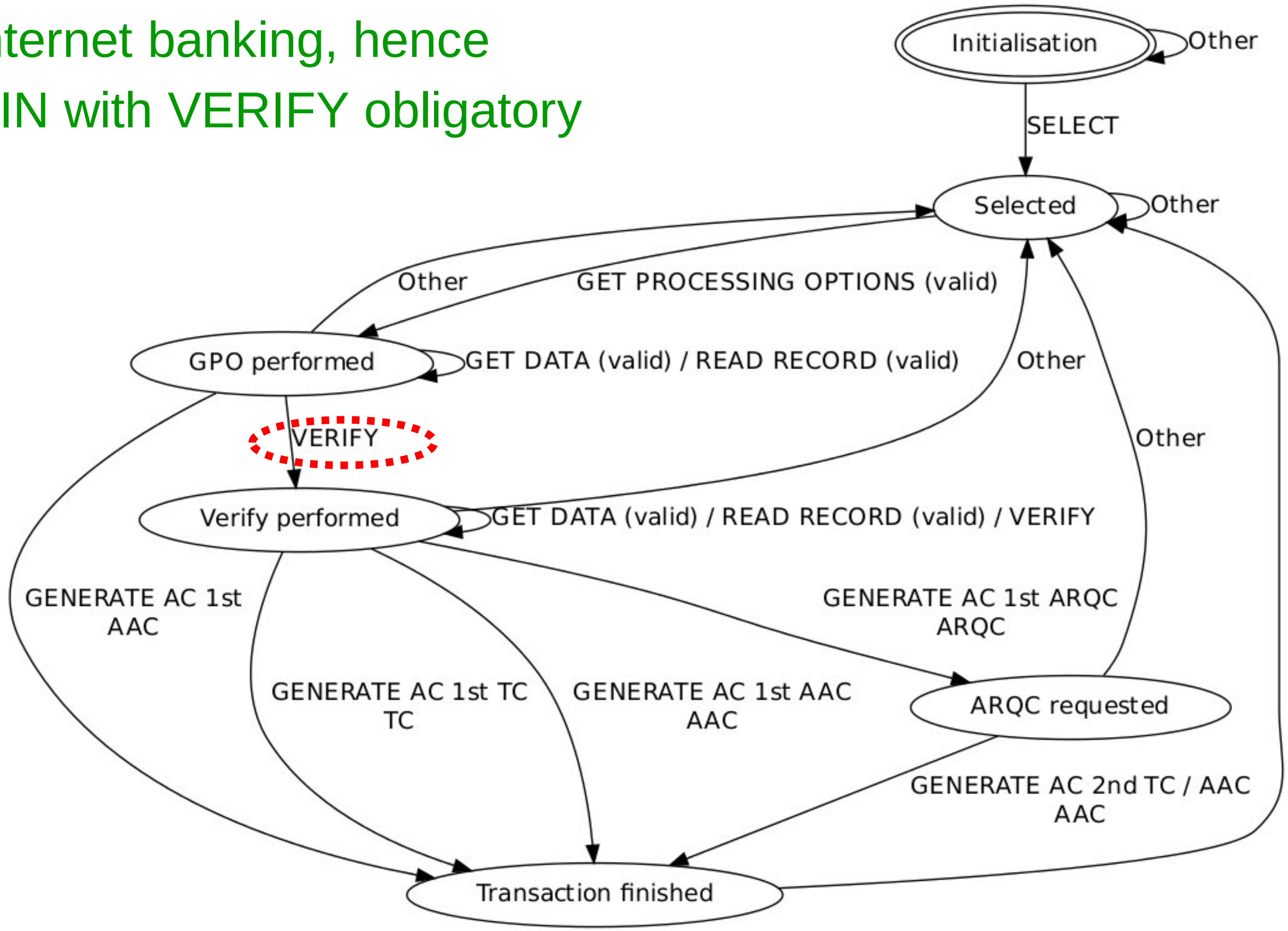
[F. Aarts et al, Formal models of bank cards for free, SECTEST 2013]

Using such protocol state diagrams

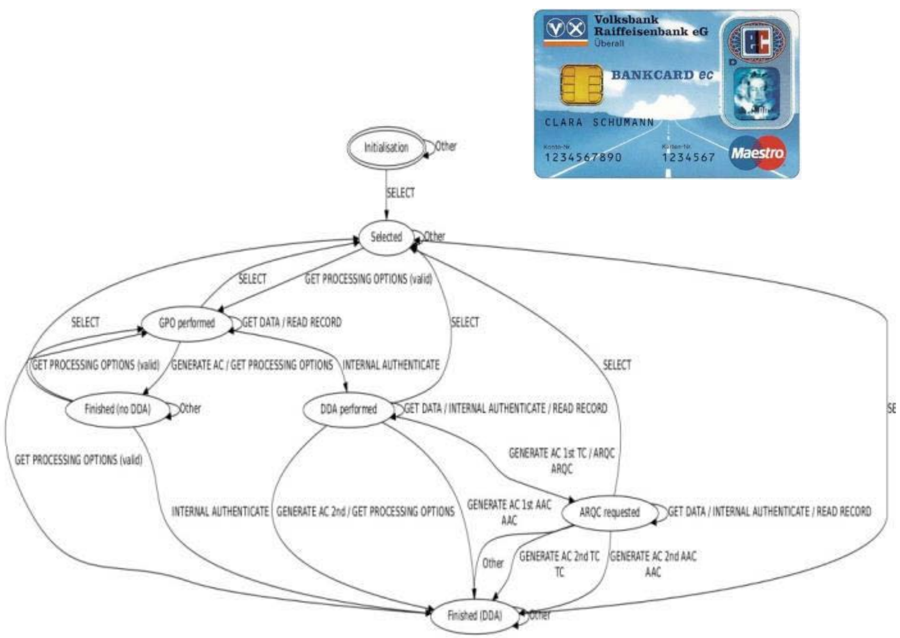
- Analysing the models by hand, or with model checker, for flaws
  - to see if *all paths* are correct & secure
- Fuzzing or model-based testing
  - using the diagram as basis for “deeper” fuzz testing
    - eg fuzzing also parameters of commands
- Program verification
  - *proving* that there is no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

SecureCode application on Rabobank card

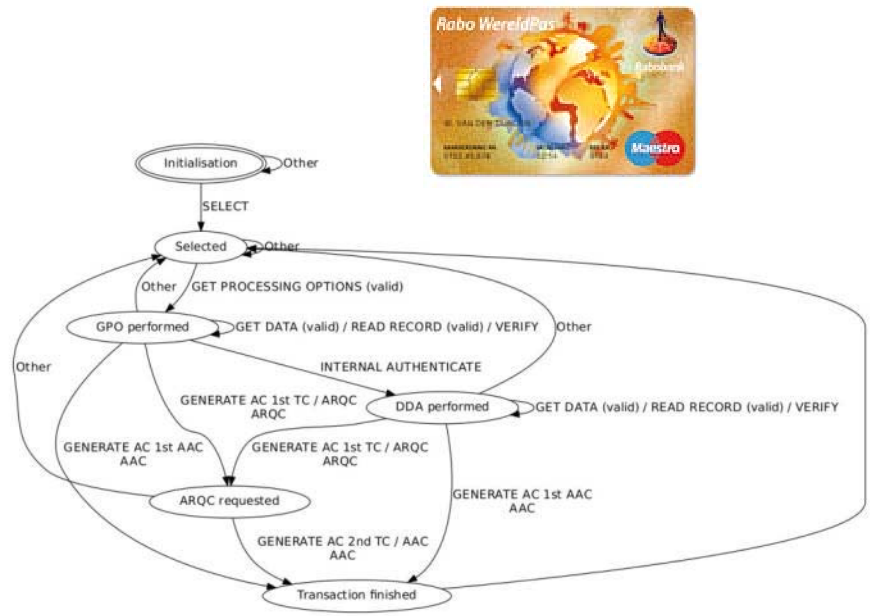
used for internet banking, hence
entering PIN with VERIFY obligatory



Comparing implementations



Volksbank Maestro implementation



Rabobank Maestro implementation

Are both implementations correct & secure? And compatible?

Presumably they both passed a Maestro-approved compliance test suite...

Case study in state machine learning (2):

EMV-CAP internet banking

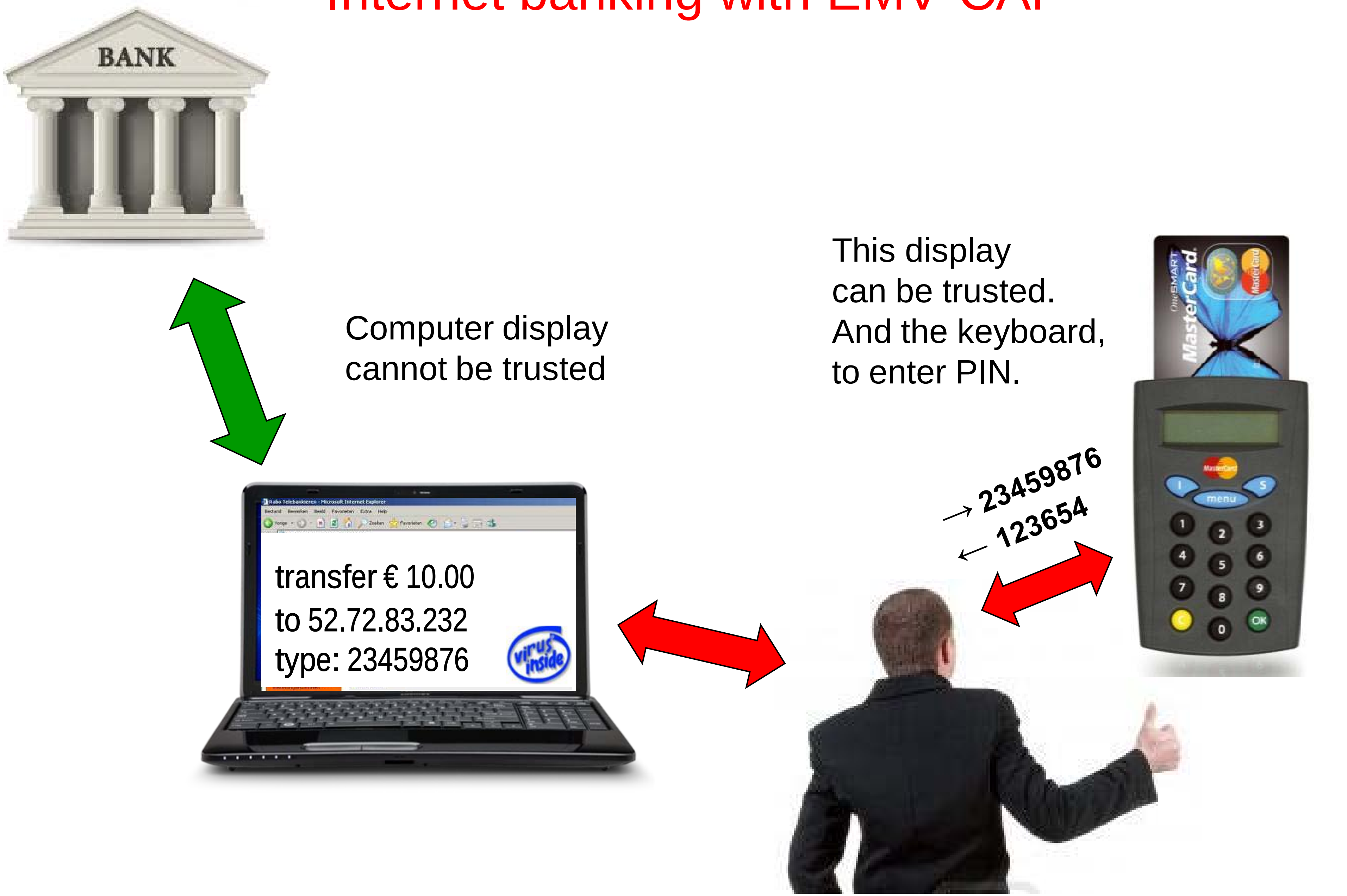
Fundamental problems

How can the bank authenticate the user?
Or (trans)actions of the user?

Computer display
cannot be trusted
despite 

How can the user authenticate the bank?

Internet banking with EMV-CAP



Internet banking with EMV-CAP

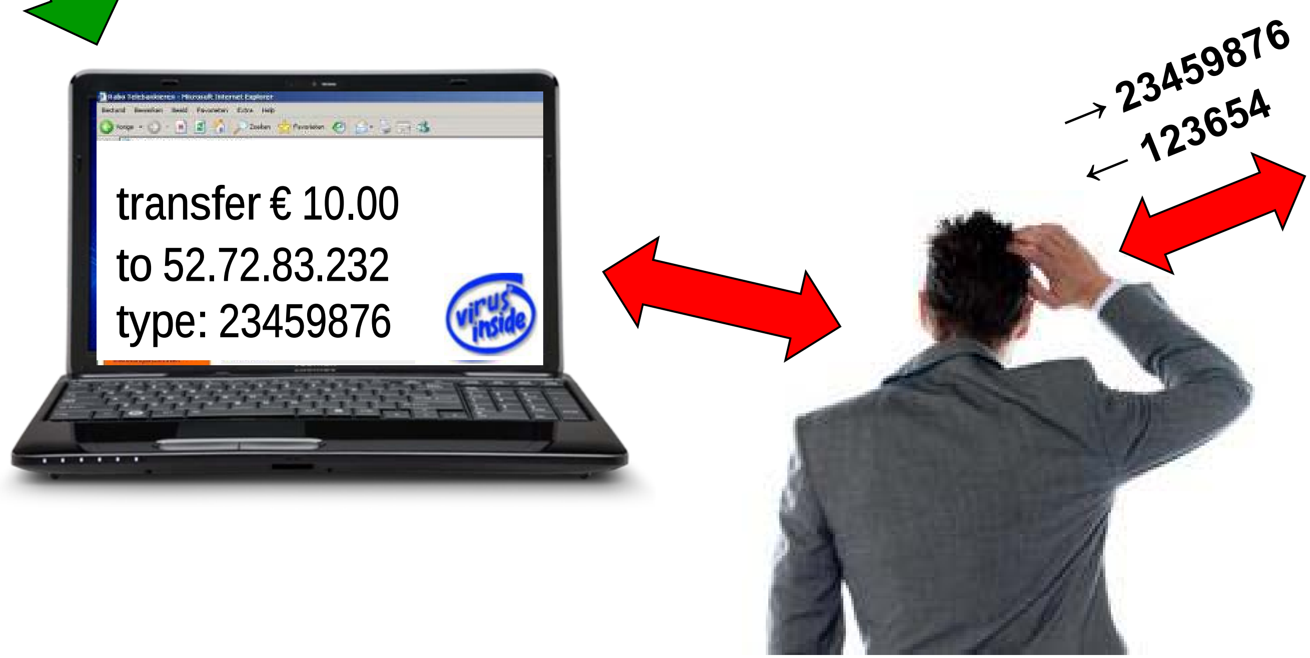
Limitation:

meaning of the numbers 23459876 unclear...

Solution:

Let the user enter transaction details
(eg amount, bank accounts,...)
on the device

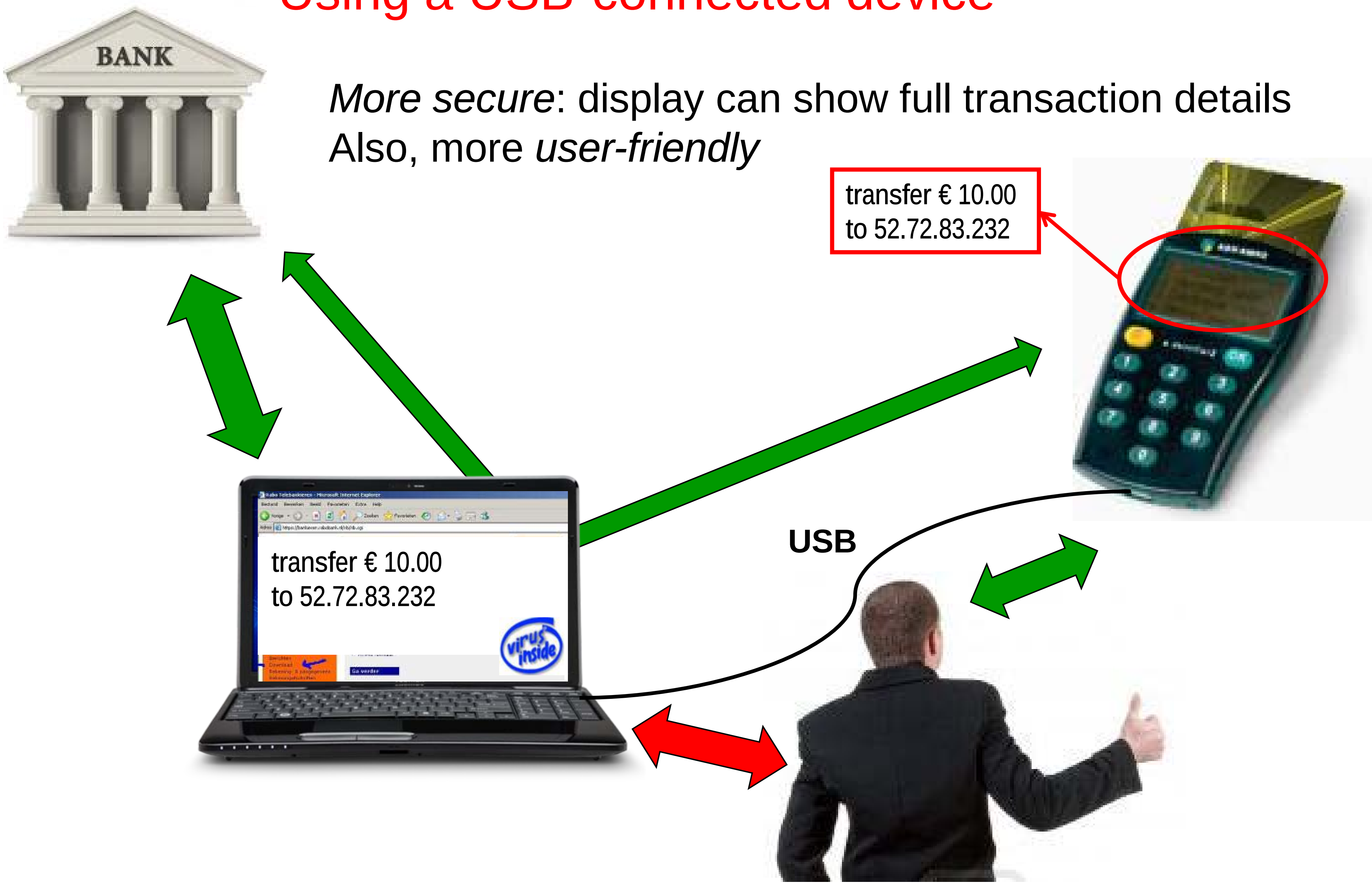
Problem: lots of typing....



Using a USB-connected device

More secure: display can show full transaction details

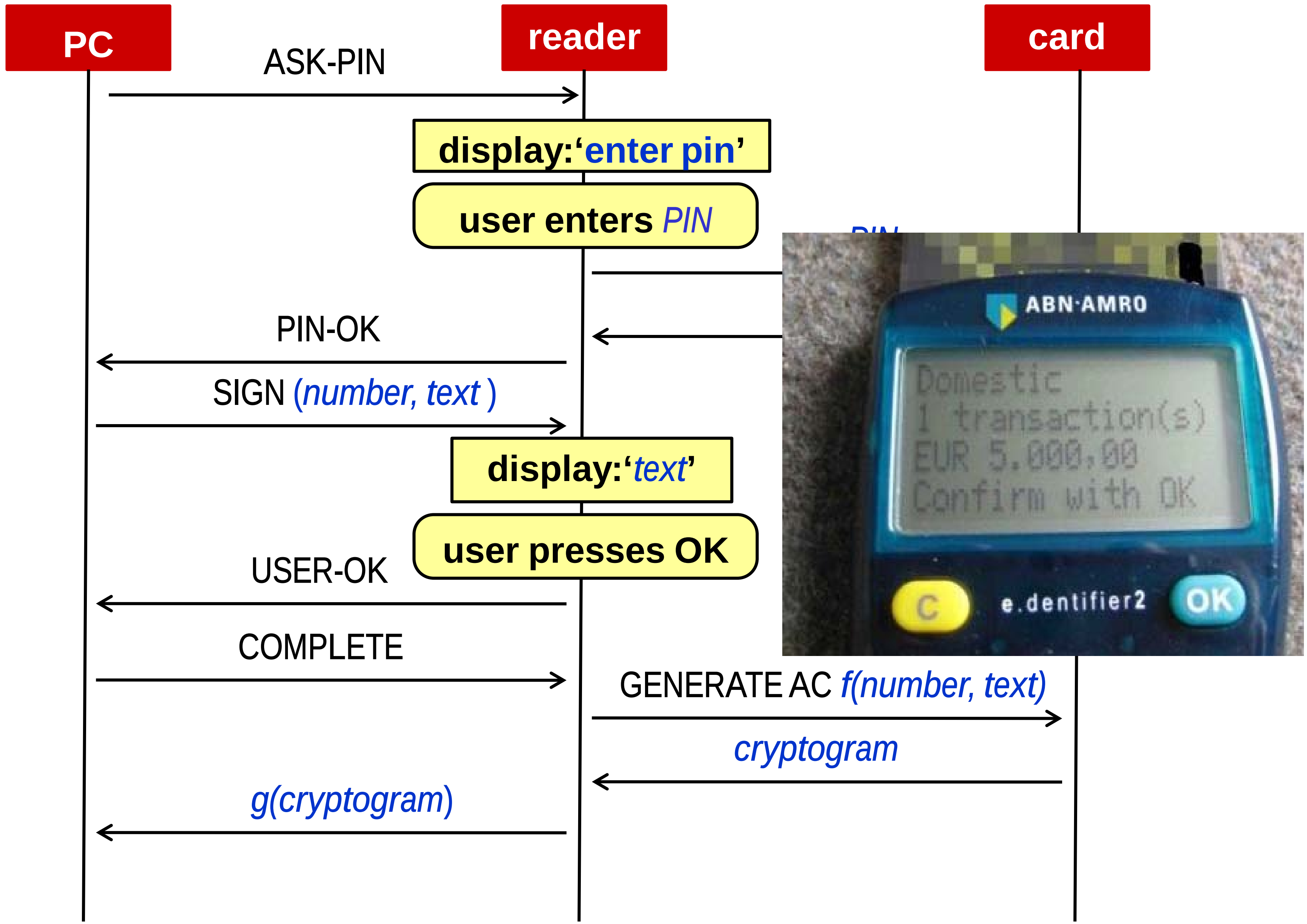
Also, more *user-friendly*



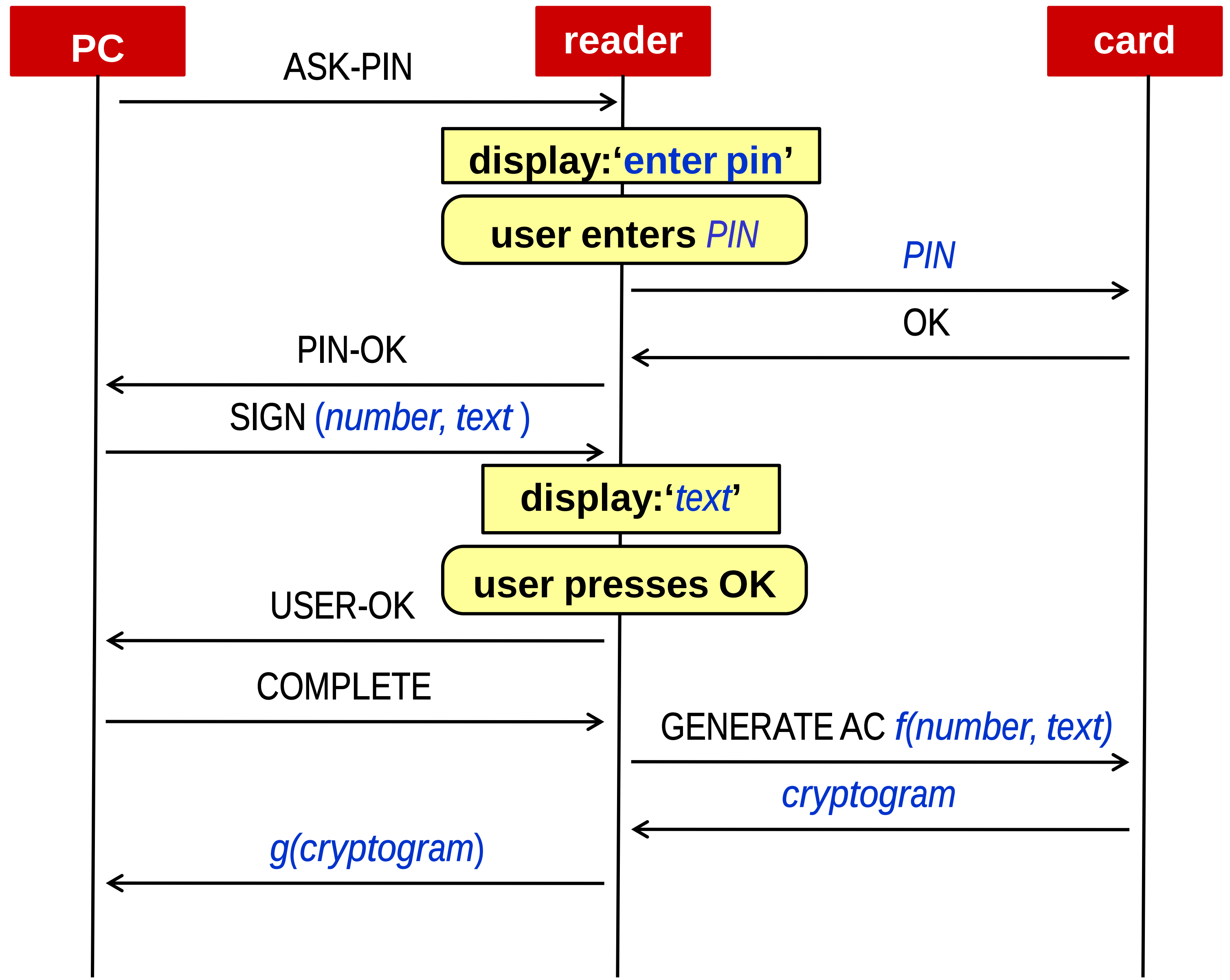
Analysis: first observation

- Some text for display goes in plain-text over USB line
- The PC can show
  - messages predefined in the e.dentifier2
  - any message that it wants to be signed

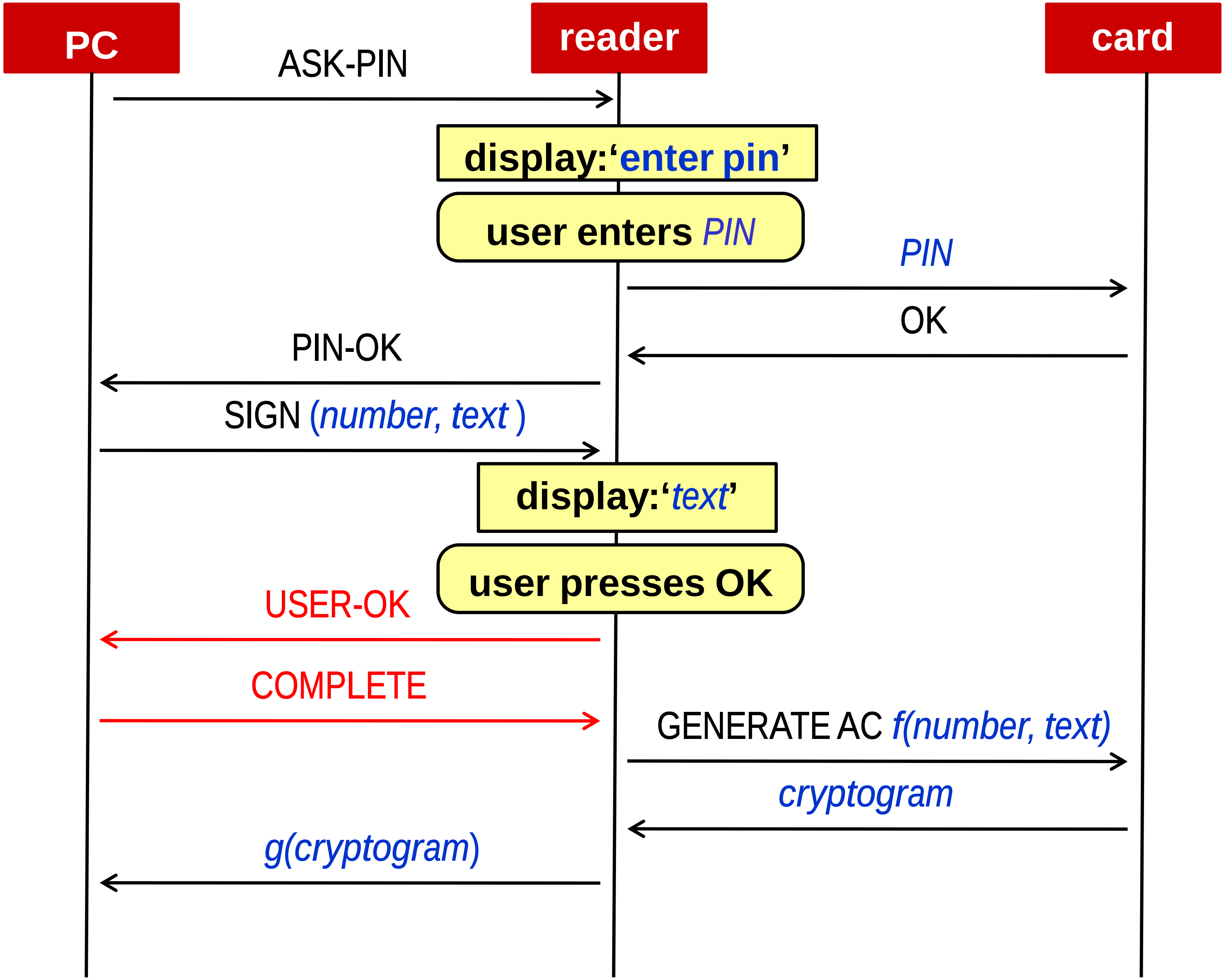
Reverse-Engineered Protocol



Spot the defect!

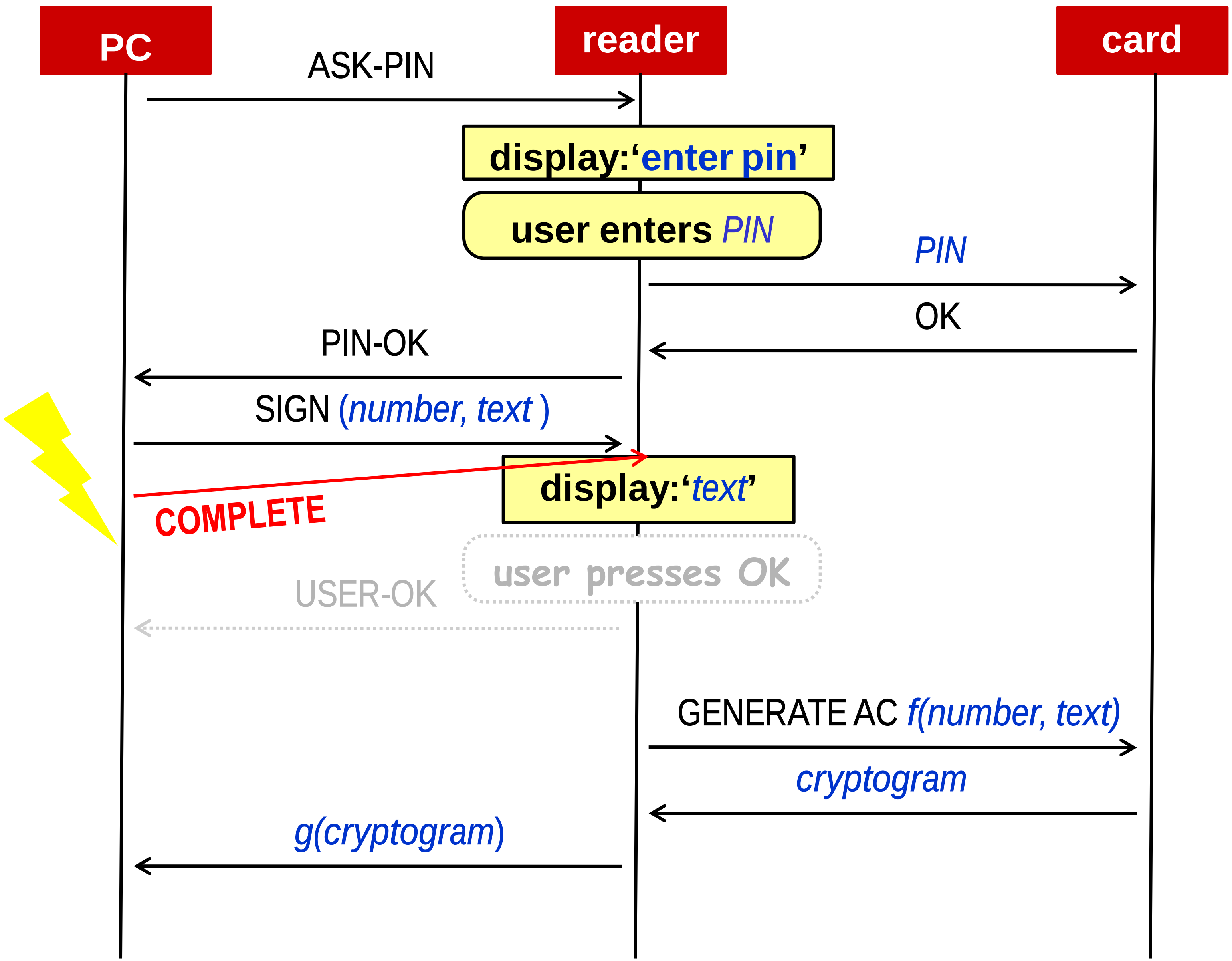


Suspicious...



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

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)

Reverse engineering the USB-connected e.dentifier

Can we fuzz

- USB commands
- user actions via keyboard

to find bug in ABN-AMRO

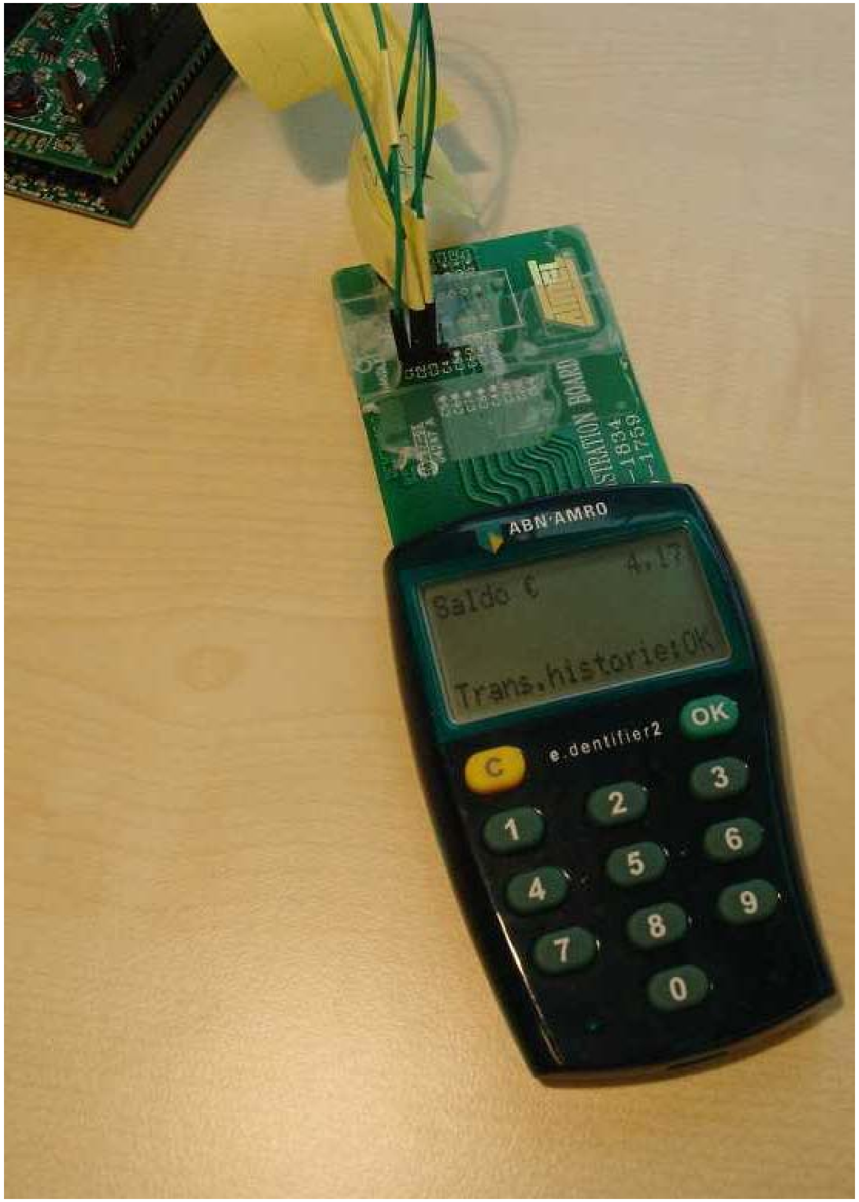
e.dentifier2

by automated learning?

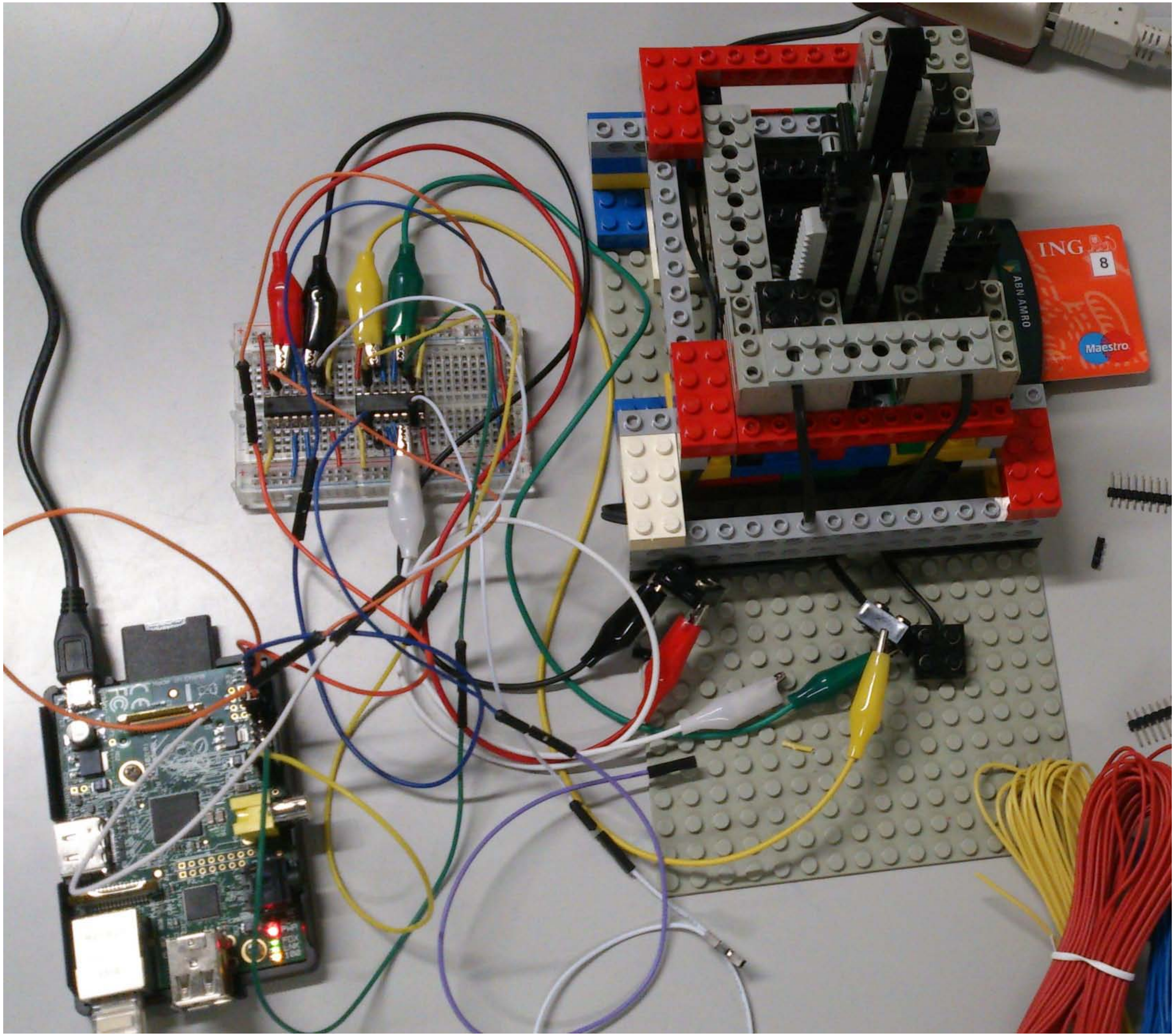
[Arjan Blom et al,

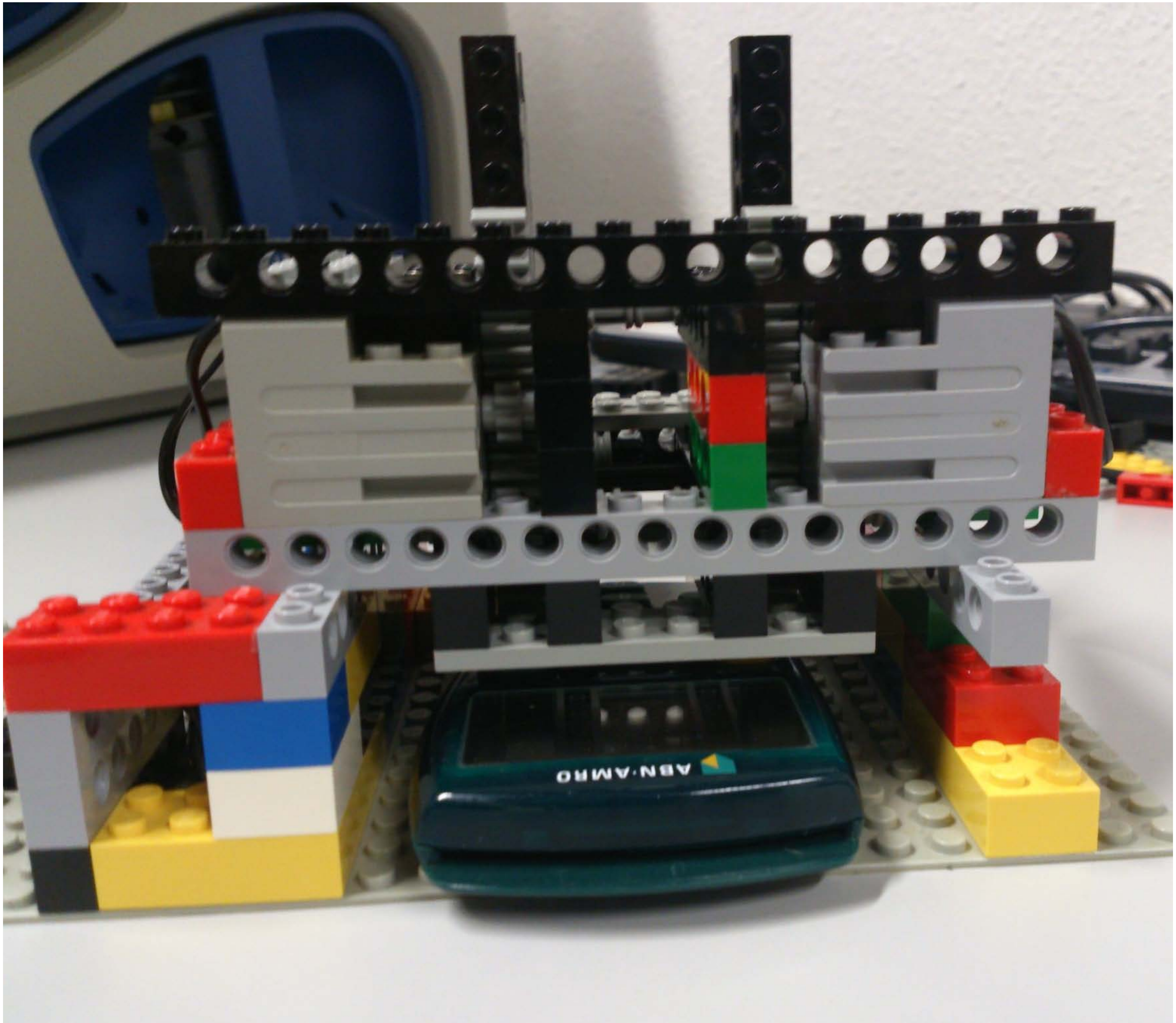
Designed to Fail: a USB-connected reader

for online banking, NORDSEC 2012]

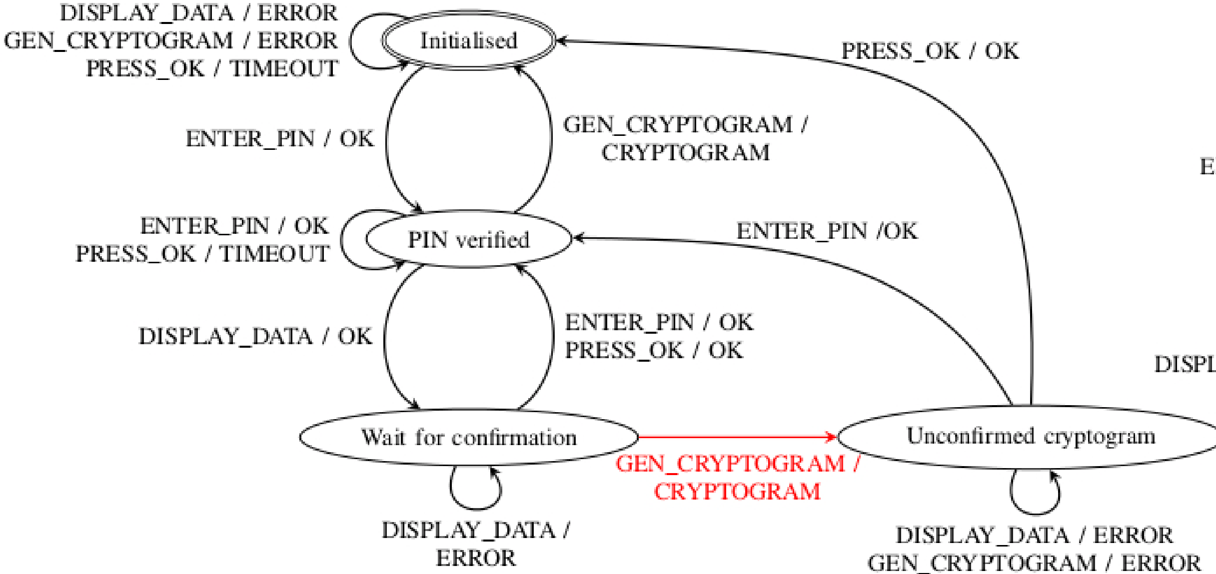


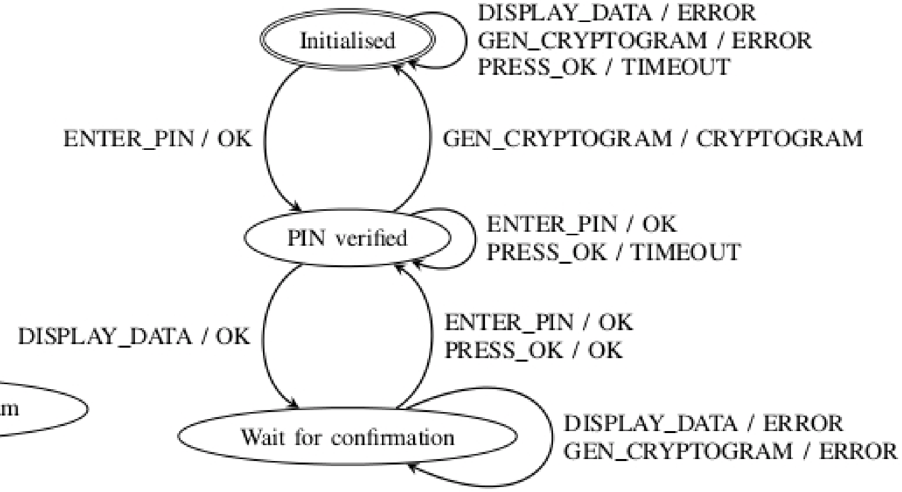
Operating the keyboard using of LEGO


ING
ABN AMRO
Maestro

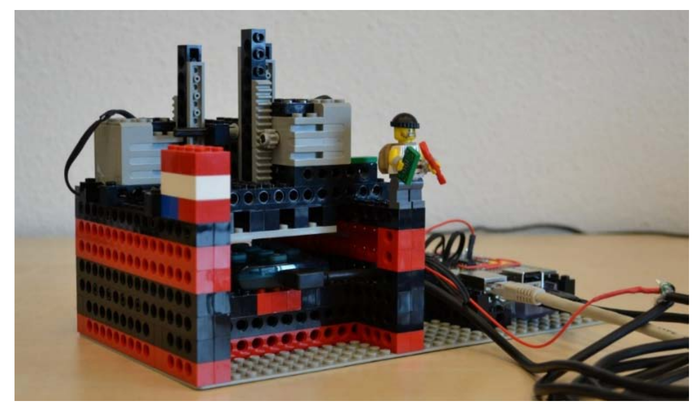

ABN·AMRO

The LEGO hacker let loose on 



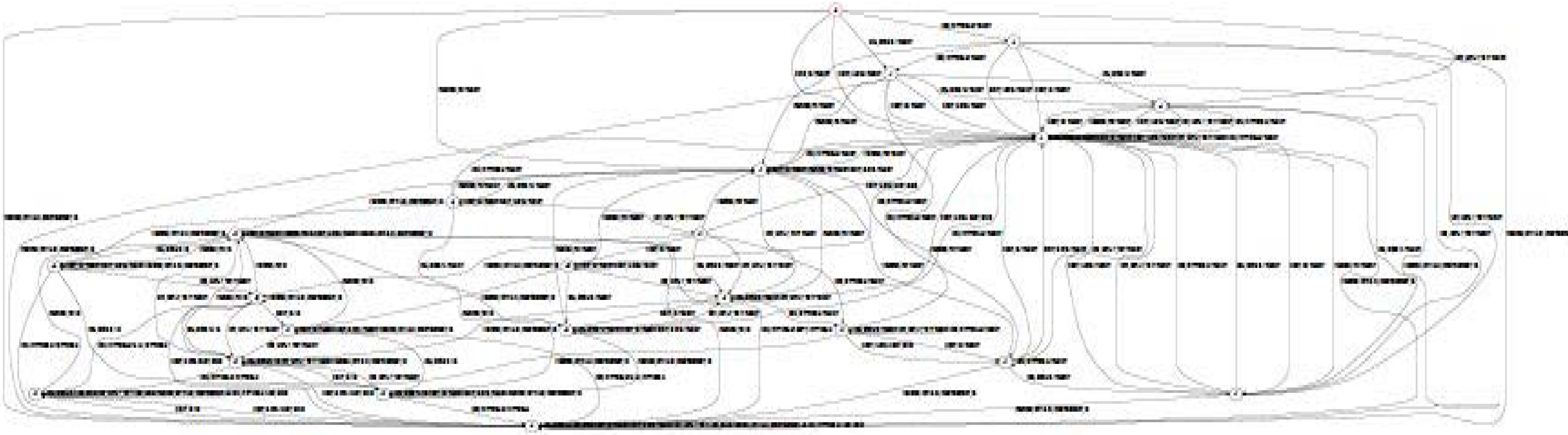


[Georg Chalupar et al
Automated Reverse Engineering using LEGO,
WOOT 2014]



http://tinyurl.com/legolearning

Would you trust this to be secure?

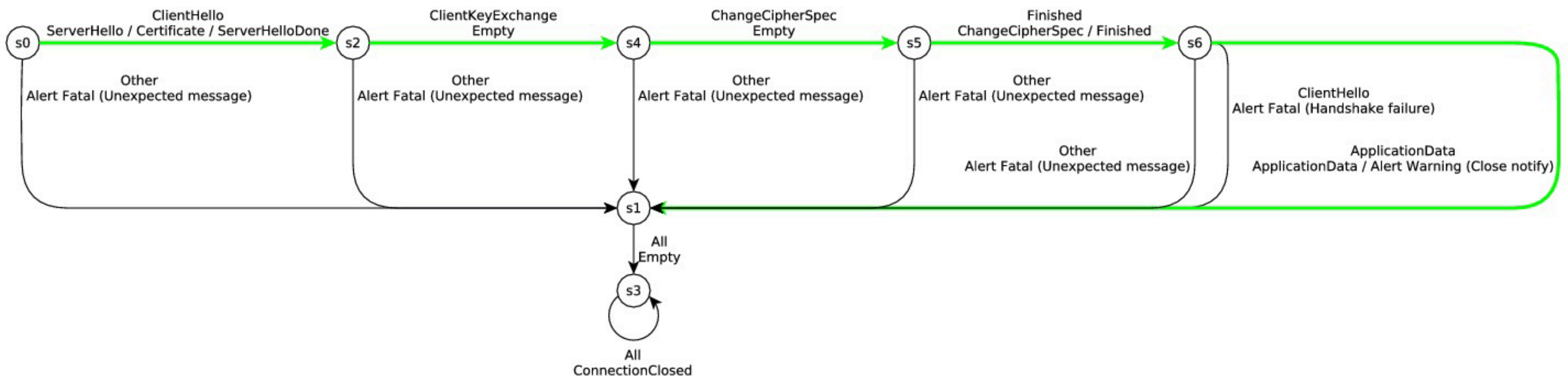


More detailed inferred state machine, using richer input alphabet.

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

Case study in state machine learning (3): TLS

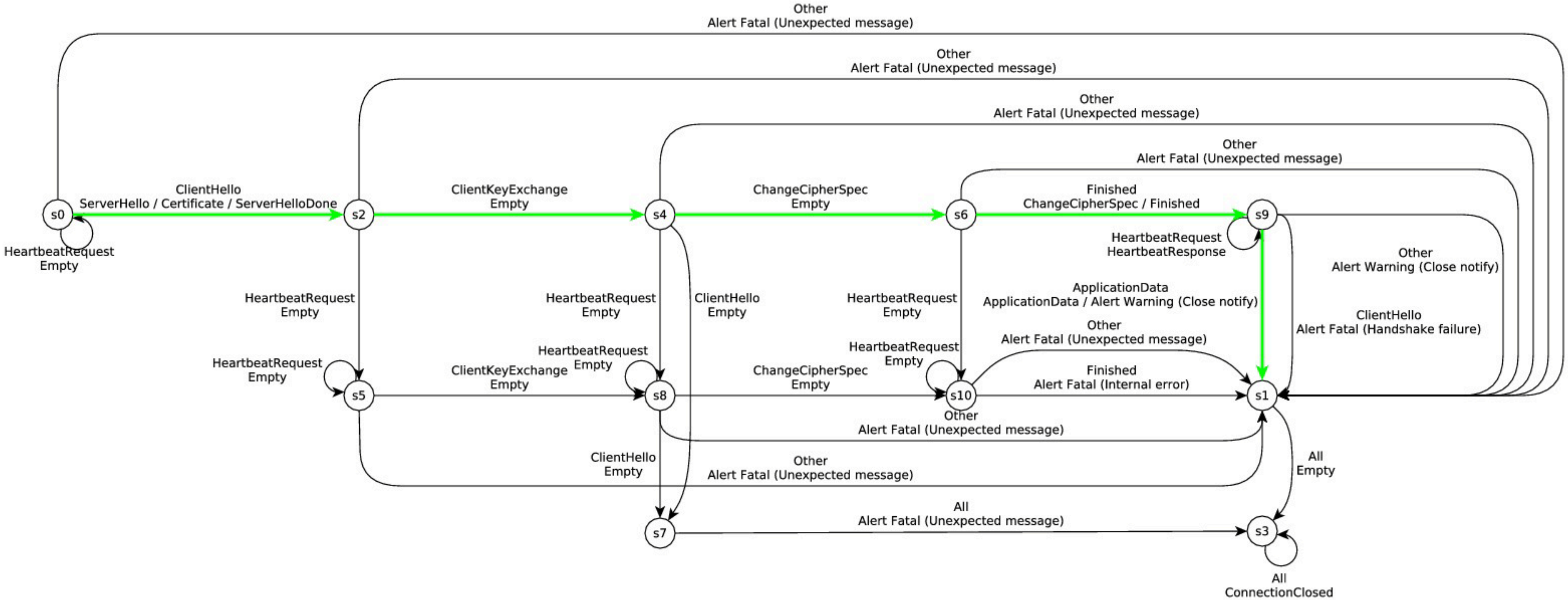
NSS implementation of TLS



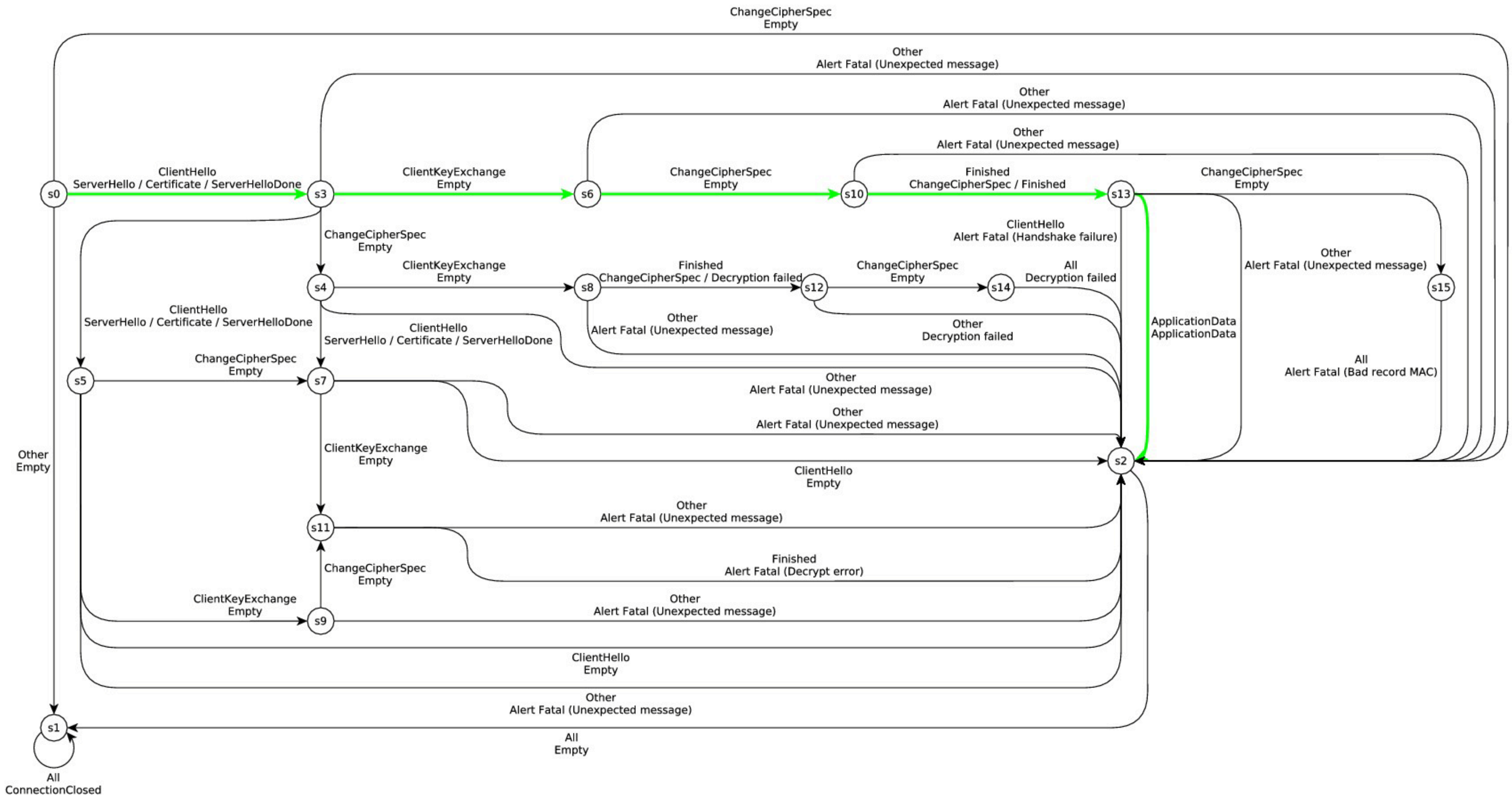
State machine inferred from NSS implementation

Comforting to see this is so simple!

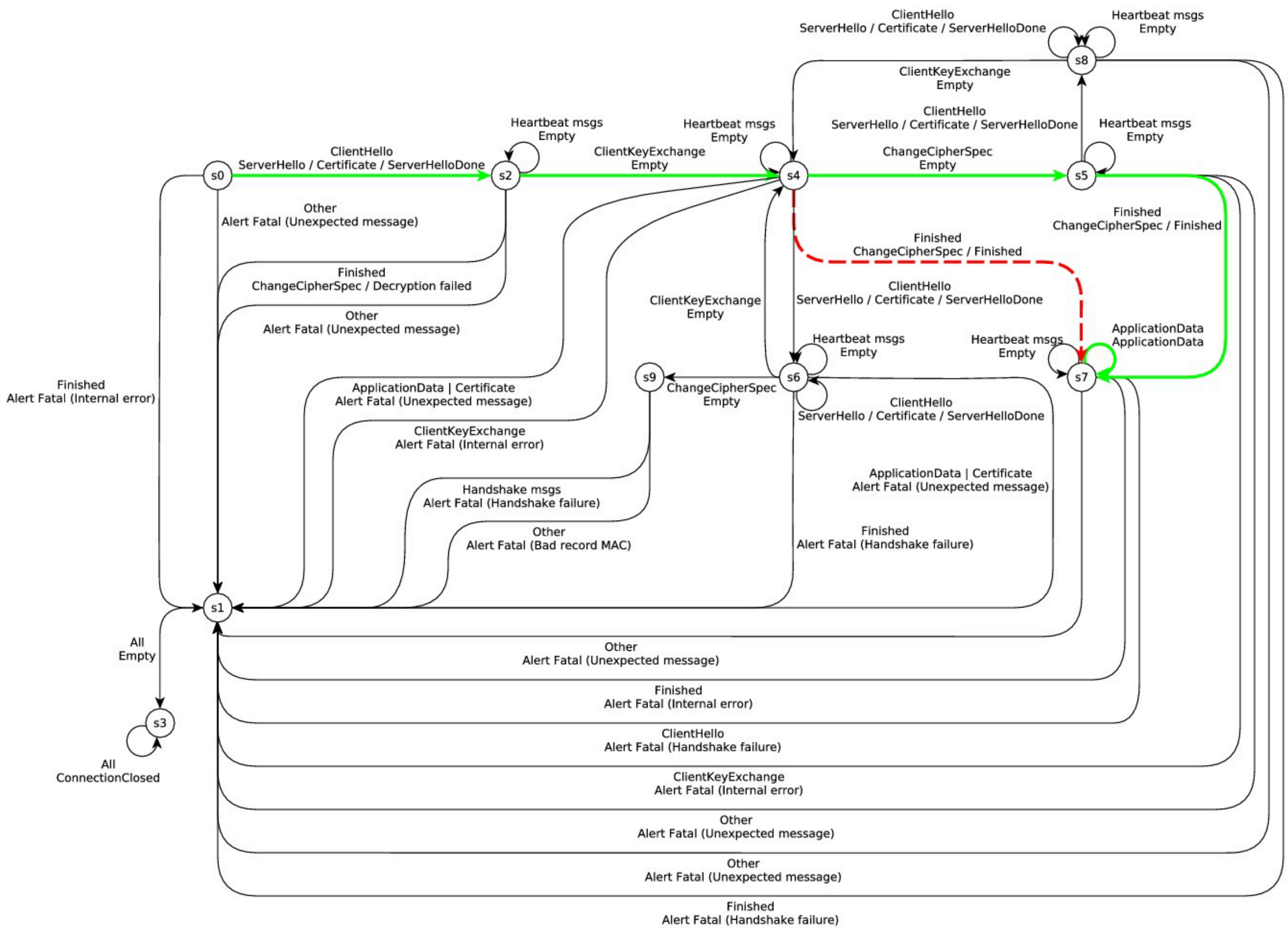
TLS... according to GnuTLS



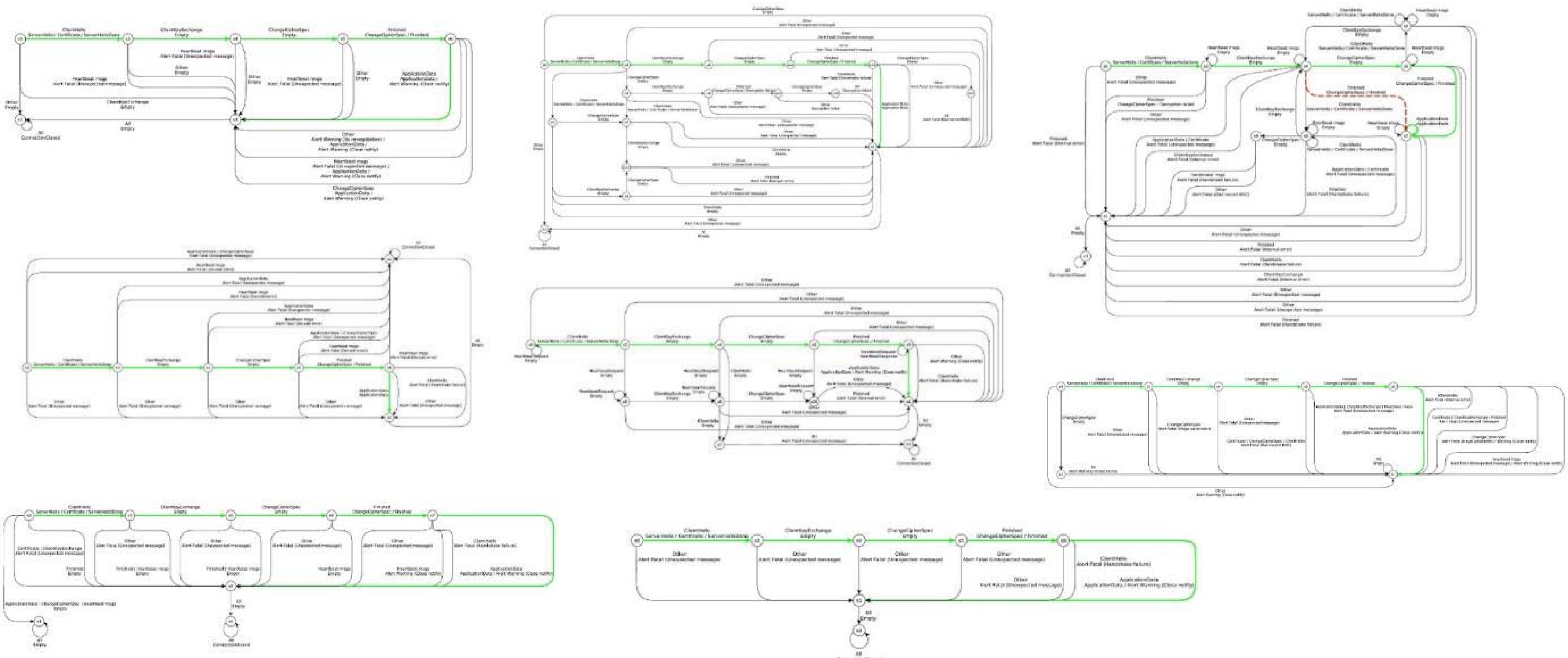
TLS... according to OpenSSL



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]

Conclusions: Dynamic Security Analysis

More reverse engineering approaches

There are many techniques for reverse engineering based on testing.

- *passive* vs *active* learning, ie passive observing vs active testing
  - Active learning involves a form of fuzzing
  - These approaches learns different things:
    - passive learning produces statistics on normal use,

      and can be basis for anomaly detection
    - active learning will more agressively try our strange things
- black box vs white box

  ie only observing in/output or also looking inside running code

Cool example of what afl (american fuzzy lop) fuzzer can do

http://lcamtuf.blogspot.com.br/2014/11/pulling-jpegs-out-of-thin-air.html

Different forms of fuzzing

1. original form of fuzzing: trying out long inputs to find buffer overflows

2. message *format* fuzzing:
   trying out strange inputs, given some format/language
   to find flaws in input handling

3. message *sequence* fuzzing
   trying out strange sequences of inputs
   to find flaws in program logic

   3a. given a protocol state machine, or

   3b. to infer the protocol state machine from an implementation

2 & 3a are essentially forms of model-based testing

3b is a form of automated reverse engineering

(Warning: there is no really standard terminology for these various kinds of fuzzing)

Protocol State Machines

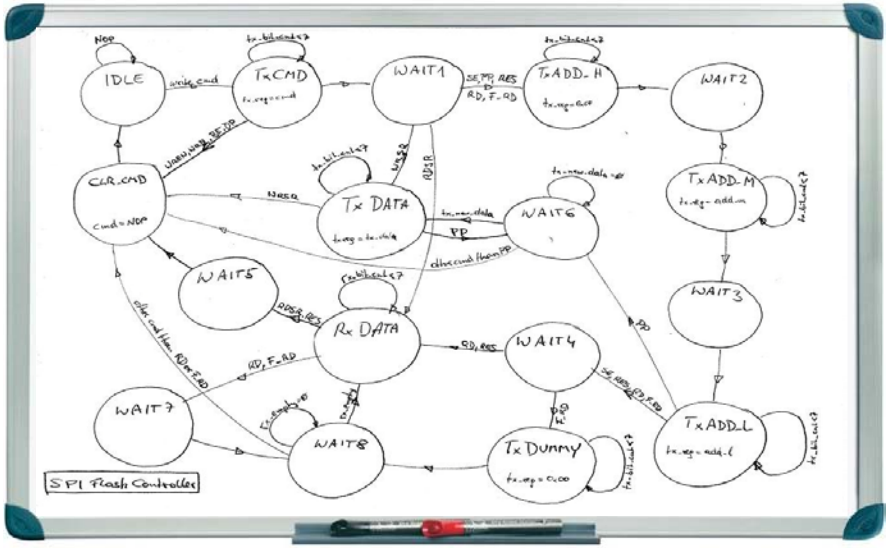
- **State machines** are a great specification formalism
  - easy to draw on white boards ☺, but typically omitted in official specs ☹

  and you can extract them for free from implementations
  - using standard, off-the-shelf, tools like LearnLib

  Useful for security analysis of protocol implementations

The people writing the specs, coding the implementation, and doing the security analyses may all be drawing state machines on their white boards...

But will these be identical?



[Protocol state machines and session languages,
SErik Poll, Joeri de Ruiter, and Aleksy Schubert, LangSec 2015]

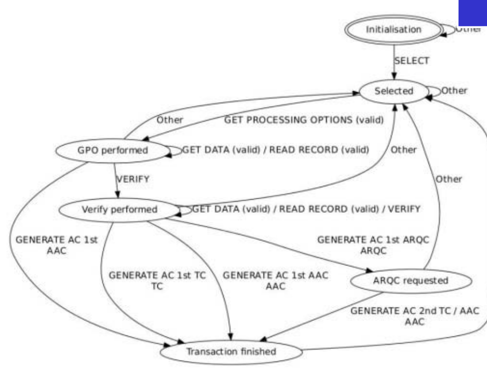
specs

implementing

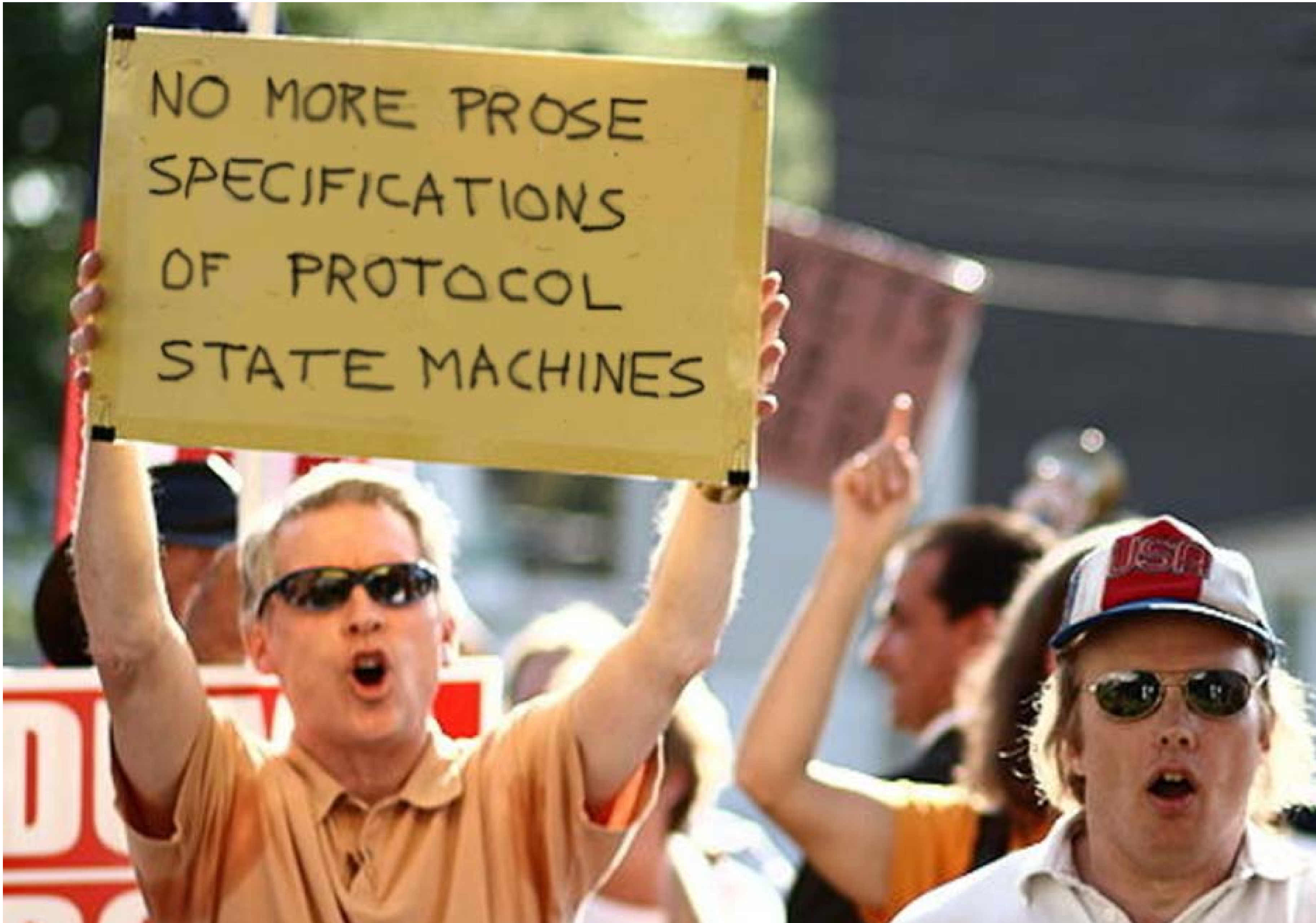
code

model-based testing

automated learning



model


NO MORE PROSE
SPECIFICATIONS
OF PROTOCOL
STATE MACHINES