Hardware Security
Course Notes

Jos Wetzels

a.l.g.m.wetzels@student.utwente.nl

1 Smartcard and RFID technology

1.1 Attacker Models for Embedded Security

Naïve Attacker Model

- Attacks on *channel* between communicating parties
- Exploiting lousy crypto or insecure protocols
- Protection by:
    - Mathematically strong algorithms and protocols
- Encryption and cryptographic operations as *black boxes*
- "*The mathematician's view"*

Improved Attacker Model

- Attacks on *channel* or *endpoints*
- Exploiting *sloppy users & software vulnerabilities*
- Protection by:
    - Mathematically strong algorithms and protocols
    - Secure software implementations
- *"The software engineer's view"*

Embedded Security Attacker Model

- Attacks on *channel*, *endpoints* or *exploitation of hardware weaknesses*
- Protection by:
    - Mathematically strong algorithms and protocols
    - Secure software implementations
    - Secure hardware
- Encryption and cryptographic operations as *gray boxes*
- *"The hardware engineer's view"*

Secure Implementations vs Secure Algorithms

Implementations of algorithms commonly deemed secure (eg. AES, RSA, etc.) can be completely insecure on given hardware hence:

- Never design your own cryptographic algorithms
- Never make your own implementations of these algorithms

This possibly goes for security protocols as well as for cryptographic primitives.

Access Control vs Cryptography
Access control:

- Involves *functionality*
- Can prevent people from reading a file by denying them to open it unless they have permission
- Can only regulate for access to resources *under our control*

Cryptography:

- Involves *data*
- Can prevent people from reading an (encrypted) file by making this impossible unless they have the right key
- Can (also) regulate access to data that are not under our control (eg. data on the network)

Problems with Cryptography
Any use of crypto *introduces* problems:

- **Key distribution**: How do we generate & distribute keys?
- **Key storage:** Where can we safely store keys (access control problem)?
- **Encryption/decryption:** Who do we trust to perform these operations?

Smartcards provide a possible solution.

1.2 Smartcards

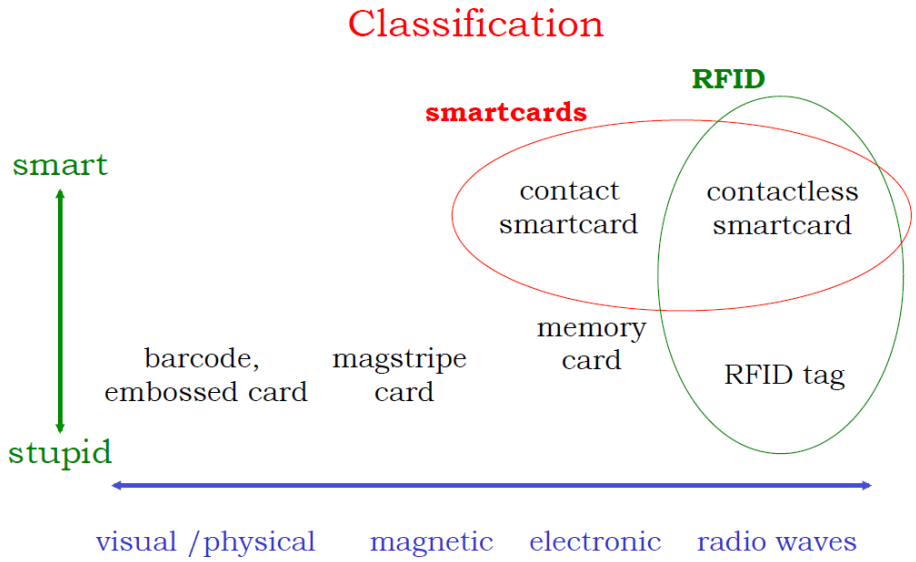
1.2.1 What is a smartcard

Smartcards (aka *chip cards* or *integrated circuit cards (ICC)*) are tamper-resistant computers with limited resources embedded in a piece of plastic which are capable of securely *storing* and *processing* data (unlike '*dumb* cards which are only capable of storing data).

Smartcards come in various forms

- Traditional credit-card sized (ISO 7816)
- Mobile phone SIM (cut down in size)
- Contactless cards (aka *proximity cards* or *RFID transpoder/tag*)

Classification of Smartcards



Security of Smartcards

- **Integrity:** Of data or functionality
- **Confidentiality:** Of data
- **Availability (resistance to DoS)**
- **Tamper-resistance**
- **Tamper-evidence**
- **Authenticity:** Resistance to copying/cloning

Functionality of Smatcards

1. Device just reports data (RO file-system)
2. Device reports data *after some access control* (protected FS, RO or RW)
3. Device can take part in crypto protocol (eg. for authentication)

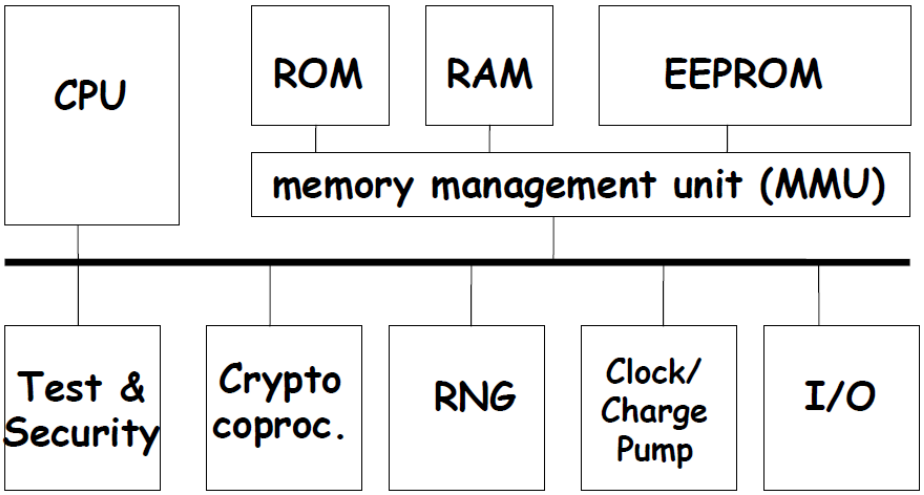
Stupid vs Smartcards

- **Memory cards ('stupid' smartcards)**
  - Essentially just a portable FS
  - Possibly with some access control or destructive writes as in old payphone cards.
  - Configurable functionality hardwired in ROM
  - RFID tags are often like old memory cards

- **Microprocessor cards ('smart' smartcards)**
  - Contain CPU
  - Possibly also crypto co-processor
  - Programmable

- Program burned into ROM or stored in EEPROM

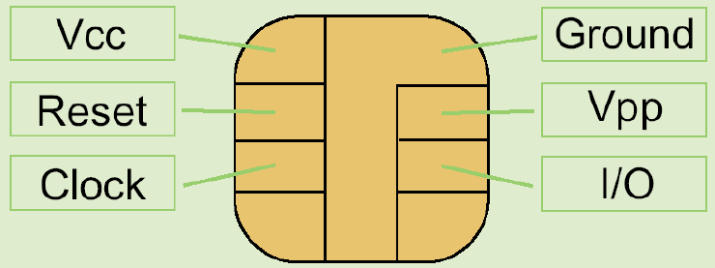
1.2.2 Smartcard Hardware



- **CPU:** 8/16/32 bit, steady need for more processing power (VMs & crypto)
- **Crypto co-processor**: DES, AES. For public key crypto ALU provides big-int arithmetic
- **Memory:** Volatile (*(S)RAM* holding workspace, stack, etc.) and non-volatile (*ROM* holding BIOS and OS + *EEPROM* holding harddisk). Typical card: 512 (typically 128-16K) bytes RAM, 16K ROM, 64K EEPROM, 13.5 MMz CPU.
- **Limited I/O**: just a serial port
- **RNG**: Typically software PRNG for key & nonce generation. Different cards of same production run should produce different RNG sequences (eg. custom seed stored in EEPROM)
- **No clock, no power!**

1.2.3 Smartcard Protocols and Operating Systems (NOT EXAM MATERIAL)

ISO 7816: Standard for contact smartcards



Contact smartcard pinout

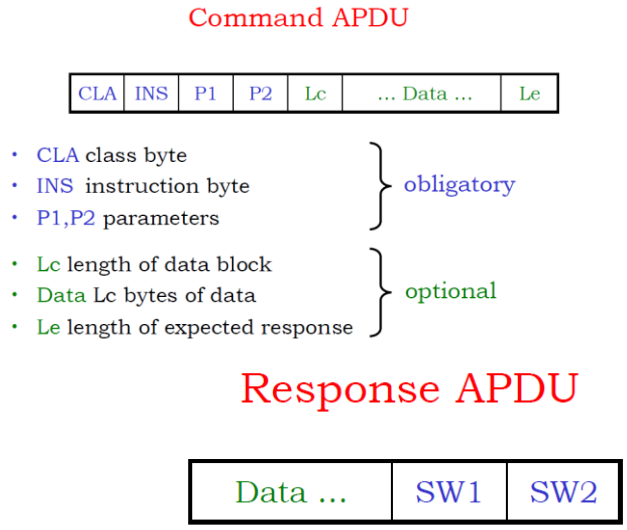
Smart Card Terminals

- Terminals (aka Card Acceptance Device (CAD)) and smartcard operate as master & slave (smartcards cannot initiate actions).

- **Problem**: No trusted I/O between user and card
  - No trusted display
  - No trusted keyboard
  - Etc.

- **Card Activation (ISO 7816-3)**
  - Terminal activates card
    - GND, VCC, CLK, RST
  - Card responds with Answer To Reset (ATR)
    - Info about protocol used, manufacturer info, (id of OS, version no. of ROM mask, etc.), last byte XOR checksum, supported I/O baudrate, etc.

- **Application Protocol Data Unit (APDU) Communications (ISO 7816-4)**
  - All subsequent communication via APDUs which are sequences of bytes in particular format.
  - Terminal sends command APDU, followed by card response APDU, etc.



- **Logic channels**: Modern cards provide several logical channels to talk to multiple applications on the card concurrently.

Smart Card Software and Operating Systems

- Code is stored in ROM vs EEPROM
- Very primitive compared to normal OSes (no multi-programming but there is multi-threading as of JavaCard 3.0)
- **Tasks**:
  - Life-cycle management (of card, of applications)
  - Instruction processing
  - Memory management
  - I/O
  - Hardware error handling (incl. atomic EEPROM update support needed due to possibility of power failure due to card tear)

- **Lifecycle elements:**
  - Production of chip & card
    - Testing & removing test functionality
  - Card preparation
    - Completing OS
  - Application preparation incl. personalization
    - Init applications, individualization, etc.
  - Card utilization

        - (de)activation of applications
    - End of card utilization
        - De-activating applications
        - De-activating cards

- **Typical lifecycle elements:**
    - Installation of application (aka applet)
    - Personalization
        - Uploading data
        - Afterwards app starts in normal active life
    - End-of-Life
        - Disabling all functionality
        - Possibly leaving logging functionality enabled
        - Upon external command or when card notices something fishy going on

- **OS Completion**
    - Initially card contains ROM mask
    - Simple loader in ROM executed to load EEPROM
    - Checksum computed
    - Switch to mode where code in ROM and EEPROM can be executed

- **Modern multi-application cards**
    - Downloadable programs written in high-level language (multiapplication, post-issuance download)
    - Examples: MULTOS, JavaCard
    - Multi-application card vision: everyone carries one card with multiple applications. Not going to happen because of:
        - Trust (banks aren't going to trust other applications to run alongside them)
        - Marketing (whose logo on the plastic?)

ISO 7816-4 File System

- Master File (MF): root directory
- Directory Files (DF)
- Elementary Files (EF): external (for app use) or internal (for OS use only)
- File formats: binary, linear fixed or variable sized records,

ISO 7816 Commands

- Commands for:
    - FS management & access
    - PIN codes
    - Challenge-response based authentication
    - Crypto

- Related standards built on top of this:

- EMV for banking cards
- GSM 11.11 and its superset EN 726-3 for SIMs
- United Nations ICAO specs for e-passport

Example Authentication Protocol

- Authentication of card:
  - Internal authenticate, arguments: random, algorithm, key no.
  - Card returns: enc(key, random)

- Authentication of terminal:
  - Get challenge
    - Card returns random number
  - External authenticate
    - Arguments: enc(key, random), algorithm, key no.

- Mutual authentication
  - Get chip number:
    - Card returns chip number
  - Get challenge:
    - Card returns smart card random s_rnd
  - Mutual authenticate:
    - Arguments: enc(key, terminal random, s_rnd, chip number), algorithm, key no.
    - Card returns: enc(key, terminal random, s_rnd)

Key Diversification

- Standard technique to reduce hassle of key management with many symmetric keys
- Terminals or central back-end has master key M
- Card with unique card number X have an individual diversified key derived from master key M and X, eg. Mx = AES(M, x)

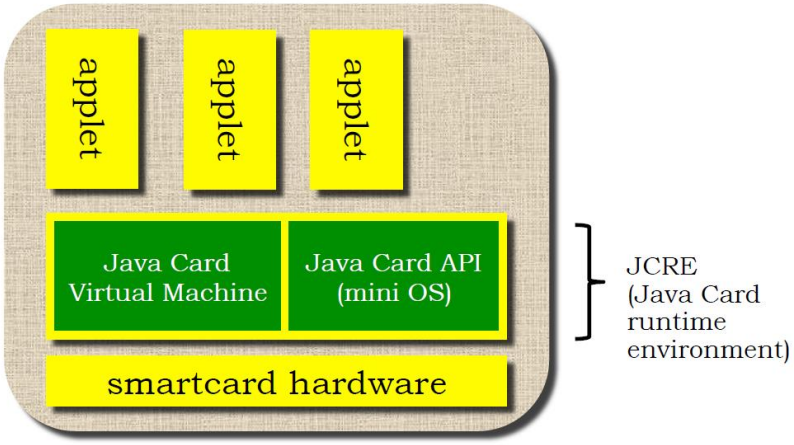
1.3 JavaCard

1.3.1 Architecture

Smartcard Operating Systems

- Old Smartcards
  - One program (applet)
  - Written in machinecode for specific chip
  - Burnt into ROM or uploaded to EEPROM
- New Smartcards
  - Applet in high-level language (eg. Java)
  - Compiled into bytecode
  - Stored in EEPROM

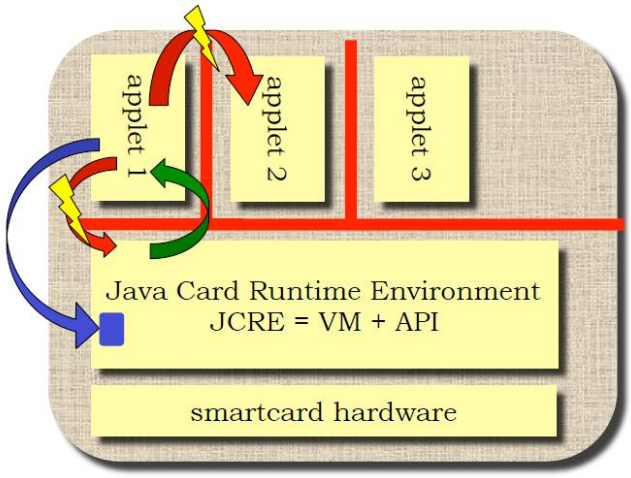
- Interpreted on card
- Multi-application, post-issuance



Javacard applets are executed in sandbox but with important differences from eg. browser applets:

- No bytecode verifier on most cards due to space constraints
- Downloading applets is controlled by digital signatures using global platform API
- Sandbox is more restrictive and includes runtime firewall to separate applets from eachother and from the OS

Javacard Firewall



- Runtime checks to prevent access to:
  - fields & methods of other applets

    - objects owned by other applets
    - even if these are public
- Exceptions:
    - Applets can access some JCRE-owned objects eg. APDU buffer
    - JCRE can access anything
    - Applets can expose functionality to eachother using sharable interfaces

- Advantages of Javacard
    - Vendor independence
    - Fast development & quick time-to-market
    - Easy to program (high level, standard functionality)
    - Open standard (no 'security through obscurity')
- Disadvantages of Javacard
    - Overhead (ample memory & CPU needed, more expensive cards needed)
    - Complexity (which brings security concerns)
    - Trust (how secure is the whole infrastructure?)
    - Ease of programming might be deceptive, invite non-experts to make trivial mistakes
    - Easy for attacker to experiment with card

Javacard Language

- Dialect of Java tailored to smartcards
- Subset of Java (due to hardware constraints)
    - No threads, doubles, strings, multi-dimensional arrays, restricted API, etc.
- With some extras (due to hardware peculiarities)
    - Communication via APDUs or RMI
    - Persistent & transient data in EEPROM & RAM
    - Transaction mechanism

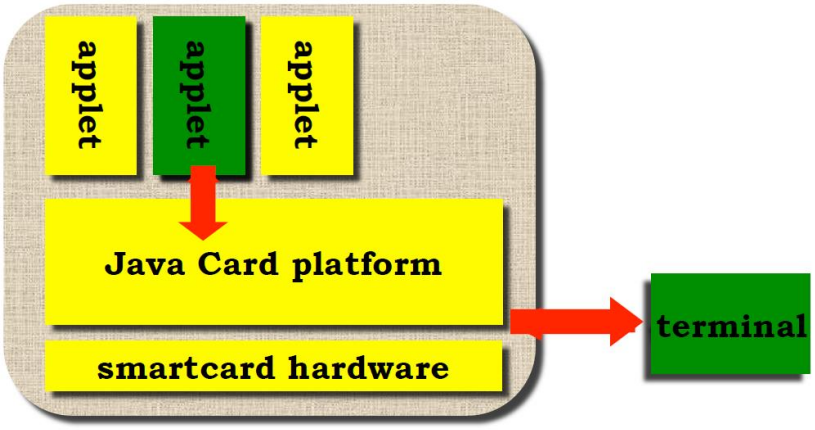
- Usually no Garbage collection, very limited memory hence no calls of **new** etc. except in installation phase. In particular, go easy on the OO

- Optimized form of class files called *cap-files*
    - Compiler translates .java to .class
    - Converter translates .class to .cap, compressing code, replacing method & field names by offsets, etc.

1.3.2 APDUs & RMI

Card-Terminal Communication

- Communication via APDUs
- Javacard OS sends command APDU to applet via API, APDU object contains byte array buffer
- Card sends response APDU back via API on same object



- Terminal sends command APDU incl. applet ID
- OS selects applet and invokes process method
- Applet executes
- Applet sends response APDU

RMI

Dealing with APDUs is cumbersome hence Remote Method Invocation (RMI) where terminal invokes methods on applet on the smartcard, platform translates them to APDUs

1.3.3 Transient & Persistent Data

Recall smartcard hardware:

- ROM: program code of VM, API, etc.
- EEPROM: program code of applets, **persistent** data storage (incl. objects with their fields)
- RAM: **transient** storage of data (eg. call stack, incl. method parameters and local variables)

Data stored in EEPROM is persistent and kept when power is lost, data stored in RAM is transient and lost upon powerloss.

By default:

- *Fields* of objects are stored in EEPROM
- *Local variables* and *method arguments* are in RAM
- Effectively: heap in EEPROM, stack in RAM

Only exception: API offers methods to allocate arrays in RAM rather than in EEPROM (only content of arrays is in RAM, array header incl. length is in EEPROM).

```
public class MyApplet extends javacard.framework.Applet{
 byte[] p; // persistent, i.e. in EEPROM
 short balance;
 SomeObject o;
 p = new byte[128];
 o = new SomeObject();
 balance = 500;

 public short someMethod(byte b) {
  short s; // transient, i.e. in RAM
  Object oo = new Someobject(); // DANGER: garbage?
  ...
}
```

Why use transient arrays?

- *Efficiency*: 'scratchpad' memory, RAM is faster & consumes less energy than EEPROM, EEPROM has limited lifetime
- *Security*: Automatic clearing of transient array on power-down or reset

```
public class MyApplet {
 boolean keysLoaded, blocked; // persistent state
 private RSAprivateKey priv;

 byte[] protocolState; // transient session state
  ...

 protocolState = JCSystem.makeTransientByteArray((short)1,
                        JCSystem.CLEAR_ON_RESET);
  // automatically reset to 0 when card starts up
 ...
```

1.3.4 Transactions

```
private int   balance;
private int[] log;
//@ invariant (* log[n] is previous balance *);
...

  // Update log
  n++;
  log[n] = balance;
  // Update balance
  balance = balance - amount;
```

what if a card tear occurs here ?

- **Card Tear:** Power supply of the card can be interrupted at any moment because of card tear. Javacard VM guarantees atomicity of bytecode instructions (like a normal VM except for operations on *doubles*)

- **Transcation**: Javacard API offers methods to join several instructions into one *atomic action*, ie. atomic update of the EEPROM. If power supply stops halfway through a transaction all assignments of that transaction are rolled back/undone which happens the next time the card powers up.

  Transactions affect all persistent data in EEPROM (ie. the heap) except some very specific EEPROM fields (eg. try-counter fields of PIN objects). Transactions do not roll back changes to transient data in RAM (ie. the stack)

```
private int   balance;
private int[] log;
//@ invariant (* log[n] is previous balance *);
...
JCSystem.beginTransaction();
  // Update log
  n++;
  log[n] = balance;
  // Update balance
  balance = balance - amount;
JCSystem.commitTransaction();
```

Complexity of transaction details introduce some potential problems:

```
Class C
{ private short i = 5;

   void m(){
      short j = 5;
      JCSystem.beginTransaction();
         i++; j++;
      JCSystem.abortTransaction();
    // What should the value of i be here?
    //        5, as assignment to i is rolled back
    // What should the value of j be here?
    //        6, as RAM-allocated j is not rolled back
   }

Class C
{ private short[] a;

   void m(){
      short[] b;
      JCSystem.beginTransaction();
        a = new short[4]; b = a;
      JCSystem.abortTransaction();
    // What should the value of b be?
    //       null, as creation of a is rolled back

    // Buggy VMs have been known to mess this up...
   }
```

```
Class C
{ private short[] a;

   void m(){
      short[] b;
      JCSystem.beginTransaction();
        a = new short[4]; b = a;
      JCSystem.abortTransaction();
       // Assume buggy VM does not reset b
      byte[] c = new byte[2000];
       // short array b and byte array c may be aliased!
       // What will b.length be?
       // What happens if we read/write b[0..999]?
       // What happens if we read/write b[1000..1999]?
}
```

```
Class SimpleObject {public short len; }
Class C
{ private short[];
   void m(){
      short[] b;
      JCSystem.beginTransaction();
        a = new short[4];  b = a;
      JCSystem.abortTransaction();
      // buggy VM forgets to clean up b
      Object x = new SimpleObject();
      x.len = (short)0x7FFF;
     // What will b.length be?
     // It might be 0x7FFF...
}
```

1.3.5 Crypto

Java Crypto Extension (JCE) provides classes

- SecureRandom
- MessageDigest
- Signature and MAC
- SecretKey, PrivateKey and PublicKey
- Cipher (object that performs crypto work)
- Certificate

Crypto in javacard works same except objects specified with short instead of string.

- Keys can be generated by card and/or terminal
- To pass keys from terminal to card or vice-versa you need to extract raw byte arrays from key objects

1.4 RFID

1.4.1 RFID Basics

Radio Frequency Identification (RFID) devices, called tags or transponders (with more powerful tags often referred to as contactless smartcards), use inductive coupling for:

- energy transfer to the card
- transmission of clock signal
- data transfer

Either in a one-way fashion (simple tags only supporting data transfer from tag to reader) or two-way fashion.

Various kinds of RFID tags:

- Animal identification RFID (only transmit permanently fixed code)
- Advanced transponders (have more data and support writing & write-protection)
- Contactless smartcards
    - Close coupling: a few mm
    - Proximity: less than 10cm
    - Vicinity: more than 10cm

'stupid' memory transponders

- Read-only (ie. tag just broadcasts serial number, one-way)
- Writable, no write-protection (1 byte to 64kb)
- Writable, some write-protection (password/key or more complicated authentication procedure, FSM OS, possible offering segmented memory each segment with its own key)

    - Important standard: MIFARE (Classic)
    - Others: DESfire, Calypso, ATMEL CryptoMemory, Legic

'smart' microprocessor transponders

- Like normal smartcard, ie. 'smart' but also wireless
- Lot less power (eg. 5-300 mW)
- Reduced resources for serious crypto or countermeasures
- Dual contact cards can allow different functionality via contacts and contactless

Near-Field Communication (NFC)

Implemented in mobile phones, phone can act as reader (active mode) or as tag (passive mode), 'next big thing' in mobile phones.

Pros and Cons of contactless

- **Pro**
    - Ease of use
    - No wear & tear of contacts on card & terminal
    - Less maintenance
    - Less susceptible to vandalism
- **Con**
    - Easier to eavesdrop on communication (terminal comm easier to eavesdrop than tag comm)
    - Communication possible without owner's consent (for replay or mitm attacks)
    - Cheap tags have limited capabilities to provide security (eg. amount of data, access control model, crypto, etc.)

Passive vs Active attacks on RFID

- **Passive**
    - Eavesdropping on communication between passport & reader
    - Possible from several meters
- **Active**
    - Unauthorised access to tag without owner's knowledge (possible up to ~25 cm, requires powerful field, 'virtual pickpocketing')
    - Variant: relay attack

Anti-Collision

Additional complexity of contactless cards is fact that multiple cards may be activated by reader, anti-collision protocol is needed for terminal to select one card to talk to.

1.4.2 MIFARE

Widely used proprietary standard by NXP, several versions (MIFARE ultralight, standard, DESfire, etc.).

Common weaknesses regardless of crypto:

- 75% of MIFRARE FID applications use default (transport) keys or keys used in examples in documentation

MIFARE Ultralight

- No keys or crypto to protect memory access
- Relies on RO and Write-once memory for security
- Memory organized in 16 pages of 4 bytes
    - First part is RO
        - Includes 7 byte serial number
    - Second part is One Time Programmable (OTP)
        - You can write 1s not 0s
        - Includes data for locking
    - Third part is readable & writable
- In sum security only provided by: OTP, locking pages and having signed/encrypted data in pages where crypto is done by terminals not the tag

Fundamental weakness: No way to protect against spoofing of tags.

MIFARE Ultralight memory layout

	Page	byte 0	byte 1	byte 2	byte 3	
read only	0	sn0	sn1	sn2	checksum	serial no
	1	sn3	sn4	sn5	sn6	
	2	checksum	???	lock 0	lock1	OTP
	3	OTP 0	OTP 1	OTP 2	OTP 3	
read/ write	4					application data
	5					
	6					
	7					
	8					
	9					
	10					
	11					
	12					
	13					
	14					
	15					

2 Smartcard Attacks & Countermeasures

Smartcards are not 100% secure, older cards are easily broken today. So if we field cards with long lifetimes we need to determine what risks are acceptable (do the cost to the attacker outweigh the potential gains?). Threats here depend on the application (eg. PayTV cloning vs SIM cloning).

Fundamental attack classes

- Logical
- Physical
- Side-Channel

General preparatory steps

1. Depackaging
    a. Remove chip from card with sharp knife
    b. Remove epoxy resin encapsulation (nitric acid + heat of infrared lamp, rinse with acetone)
2. Optical reverse-engineering (microscope images with different layers in different colors, before and after etching)

Cat-and-Mouse Game

- < 1990: Eavesdropping communications
  - Secure messaging (encryption)
- ~1990: Removing passivation layer
  - Passivation detectors
- ~1990: Manipulation of data transfer
  - Secure messaging (authentication)
- ~1991: Erasing the EEPROM using UV light
  - Light sensors
- ~1991: Equivalent circuits for memory cards
  - Challenge response for authentication
- ~1992: Interrupting supply voltage
  - Increment retry counter before PIN test
- ~1993: Stopping the clock
  - Underfrequency detector
- ~1993: Manipulating with a laser cutter
  - Shield on microcontroller
- ~1995: Timing attack
  - Implementations not dependent on secret data
  - "Noise-free" cryptographic implementation
- ~1995: Bus tapping with microprobes
  - Scrambling busses on microcontroller die
- ~1996: FIB manipulation of microcontroller
  - Shields on microcontroller
- 1997: Exhaustive keysearch on DES
  - Use of Triple-DES
- 1998: Power analysis (SPA/DPA)
  - Random delays, constant power consumption
- 2001: Software attacks by downloading executable code
  - Card verifies code authentication before execution
- 2005: Relay attack with RFID devices
  - Distance bounding

Classification of attacks

- Cost (time, equipment, know-how)
- Impact
- Tamper-evidence

- Logical vs Physical
  - Logical: Use the card as it is
  - Physical: Attacker underlying hardware
- Passive vs Active
- Invasive vs Non-invasive
  - Non-invasive: can happen in few minutes, violate tamper-evidence and tamper-resistance

    - Invasive: requires hours to weeks and lab equipment, are tamper-evident so only violate tamper-resistance

Side-Channel Attacks
Many physical attacks exploit side-channel attacks. Either passively (through observation) or actively (through fault introduction):

- Timing
- Power consumption
- EM radiation
- Radio Frequency Analysis (RFA) for contactless

Confidentiality and Integrity
Attacks on

- Confidentiality: eg. to get access to keys stored on card, clone card, etc.
- Integrity: change data stored on card or change behavior of card

Confidentiality and integrity not only important for crypto keys and PINs but also for software on the card and logic implemented in hardware on the card.

Smartcard Attacks – Cost

- Logical Attacks
    - 100$ of equipment
    - RE may take weeks, final attack will be real-time
- Side-Channel Attacks (DPA)
    - 5K$ of equipment
    - Again lots of time to prepare but attack almost real-time
- Physical Attacks
    - 100K$
    - Several weeks

Smartcards - Future

- Moral of story: Hard to keep secrets from motivated & well-funded attackers.
- Ongoing arms-race
- Some physical attacks are becoming harder due to improved countermeasures and smaller circuitry
- But side-channel attacks are here to stay
- And increasing complexity of software may introduce more opportunities for logical attacks

What to worry about?

- Choosing secure crypto primitives & key lengths is the easy part
- Hard part is:
    - Insecure implementations of primitives esp. in the face of side-channel attacks
    - Insecure protocols using these primitives

    - Software bugs in general or software weaknesses wrt fault injections

3 Side-Channel Attacks

3.1 Introduction to Physical attacks and Side-Channel attacks

Goals of attackers:

- Secret keys/data
- Unauthorized access
- IP/piracy
- (location) privacy
- (theoretical) cryptanalysis
- Reverse engineering
- Finding backdoors
- …

We need both:

- Efficient implementation
    - Within power, area, timing budgets
- Secure implementation
    - Resistant to attacks
    - Active attacks: probing, glitches, JTAG scan chain, cold-boot
    - Passive attacks: power consumption, electromagnetic emanation, sound, temperature, etc.

Side-channel attacks based on (non-intentional) physical information leakage. Often optimized implementations enable leaks:

- Cache: faster memory access
- Fixed computation patterns (rounds)
- Square vs Multiply (RSA)

Physical attacks (gray box, physics) are not cryptanalysis (black box, math).

Attack categories

- Side-channel attacks
- Faults
- Microprobing
- Reverse engineering

3.1.1 Side-Channel Analysis Basics

Analysis capabilities

- Simple attacks (few measurements, visual inspection)

- Differential attacks, multiple measurements, use of statistics, signal processing, etc.
- Higher-order attacks (nth order is using n different samples)
- Combining two or more side-channels
- Combining side-channel attack with theoretical cryptanalysis

Devices under attack

- Smartcard
- FPGA, ASIC
- RFID, PDAs
- Phones, USBs

Practical issues

- Quality of measurements
    - Noise issues
        - Sources: power supply, clock gen.
        - Algorithmic, sampling, external, intrinsic, quantization
        - Averaging multiple observations helps

- Aligning the measurements
    - Due to time randomization, permuting execution or hardware countermeasures

3.1.2 Power Analysis Attacks

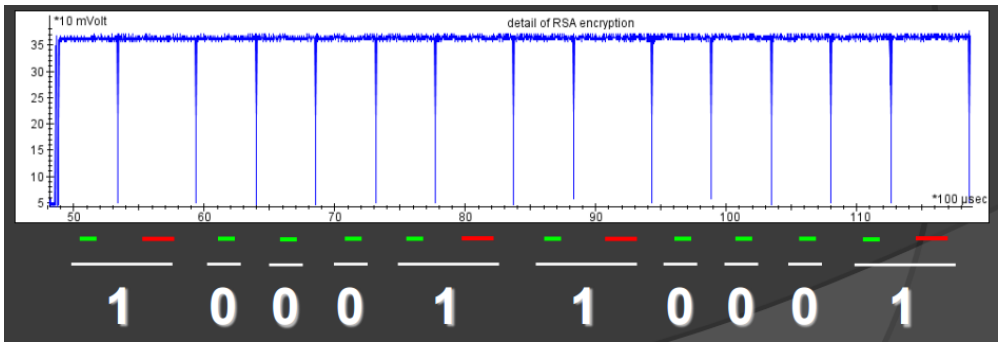
3.1.2.1 Simple Power Analysis (SPA)

- Based on one or a few measurements
- Mostly discovery of data-(in)dependent but instruction-dependent operations
- Symmetric
    - Number of rounds (resp. key length)
    - Memory accesses (usually higher power consumption)
- Asymmetric
    - Key (if badly implemented RSA/ECC)
    - Key length
    - Implementation details (eg. RSA w/wo CRT)
- Search for repetitive patterns

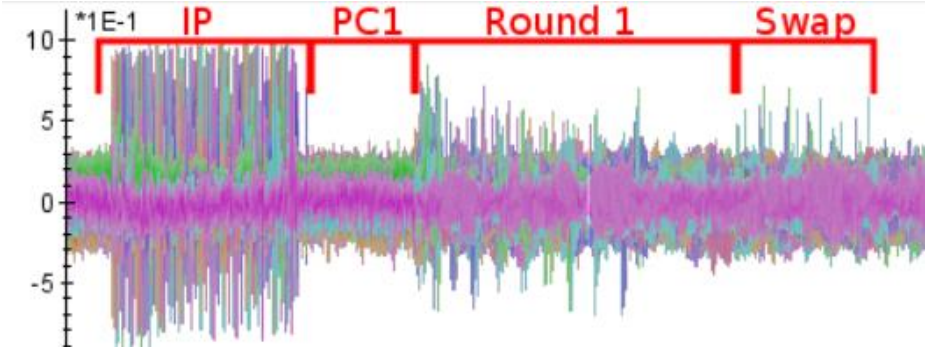
RSA modular exponentiation

```
In: message m, key e(l bits)
Output: m^e mod n

A = 1
for j = l - 1 to 0
    A = A^2 mod n  /* square */
    if (bit j of k) is 1 then
    A = A x m mod n  /* multiply */
Return A
```



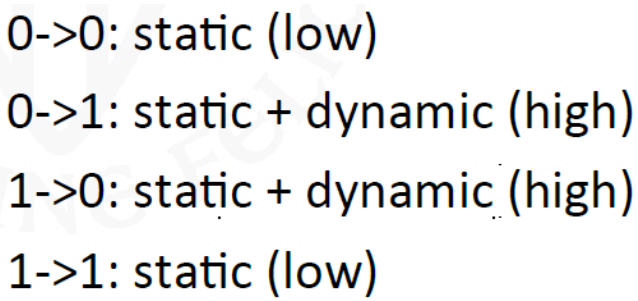
SPA can be used to find a good place for other attacks (eg. DPA).



3.1.2.2 Differential Power Analysis (DPA)

Static CMOS

- Most popular circuit style
- Power analysis attacks explore fact that instantaneous power consumption depends on data and instructions being processed
- Power consumed when an input signal switches is much higher



- Hence adding 0xA7 to 0xB9 may result in more bitflips (and correspondingly transistor switches) than when adding 0x01 to 0x00.

Traces

A trace is a sequence of measurements taken across a cryptographic operation or sequence of operations. Trace data is captured by putting a resistor in series with the device's ground line and using an oscilloscope to measure the voltage at the ground input, larger measurements represent higher power consumption. Rounds of symmetric ciphers (and other cryptographic operations) are often (but don't necessarily have to be) clearly identifiable from visual inspection of the trace.

A trace represents the power consumption corresponding to a particular key and input combination. Hence in DPA we seek to attack a scenario where we have an unknown but static secret key and either a chosen plaintext or chosen ciphertext scenario.

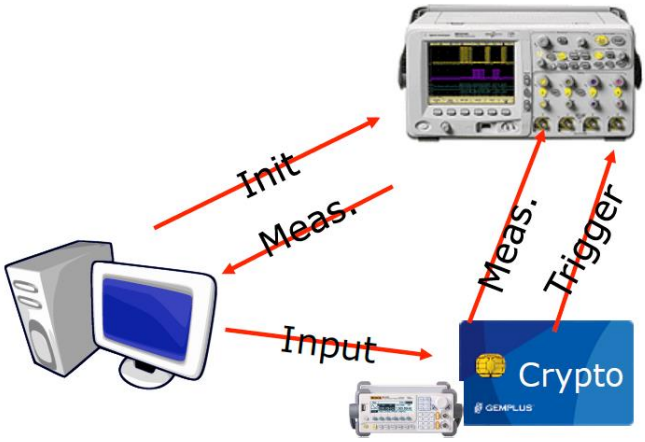
DPA Basics

The basic method involves partitioning a set of traces into subsets and then computing the difference of the averages of these subsets. If the choice of which trace is assigned to each subset is uncorrelated to the measurements contained in the traces, the differences in the subsets' averages will approach zero as the number of traces increases. Otherwise, if the partitioning into subsets is correlated to the trace measurements, the averages will approach a nonzero value. Given enough traces, extremely tiny correlations can be isolated – no matter how much noise is present in the measurements.

Note the following:

- Information revealed by a DPA test is determined by the choice of *selection function*
- Selection function is used to assign traces to subsets (eg. divide based on LSB of S-Box output) and is typically based on educated guess as to possible value for one or more intermediates
- If final DPA trace shows significant spikes cryptanalyst knows selection function output is correlated to (or equals) a value actually computed by target device, if no correlation is observed then selection function output was not correlated (or too small to observe).
- Selection functions may be predicted value of a single bit (eg. output bit of S-Box or multiplier). More complex functions (eg. predicted difference between value of a bit in a register and value of a bit that overwrites it) may also be used. Selection functions can also be functions of multiple bits.

Measurement Setup



- Cryptographic device under attack
- PC
- Power measurement circuit (or EM probe)
- Power supply and clock generator (for smartcard clock is around 4MHz)
- Control and analysis software
- Oscilloscope

DPA Steps

Assume we have a device performing (symmetric, block cipher) encryption operation $E(x, k)$ for a known x and we want to determine k. In most common block ciphers multiple rounds are applied where a subkey k_i tends to be split into several (individually brute-forcable) fragments and is combined with the input, eg. through use of S-Boxes. A typical DPA attack then involves the following stages:

1. Choose Intermediate Result for targeting: We choose an intermediate result $f(x, k_i)$ of the cryptographic algorithm E where, for a given round j, an easily bruteforcable part of the subkey k_i is combined with our input x.

    - Let us choose $f(x, k_j) = S(x' \oplus k_j')$, ie. some S-Box mapping part of input x and part of subkey k_j to an output value.

2. Device instrumentation: We set up up means to communicate with device, invoke cryptographic operations and record responses.

3. Measurement: We collect power traces using known input, key guesses. We now have a set of N power traces $p_i(t), 1 \leq i \leq N$, with corresponding inputs x_i as our input vector $X = (x_1, .., x_N)$. The power traces of size T each can be represented as a power trace vector $p_i = (p_{i_1}, .., p_{i_T})$ and hence we have a measurement matrix M of N x T.

4. (Optional) Signal processing: This optional stage involves processing power traces in software to remove alignment errors, isolate features of interest, highlight signals, reduce noise, etc.

5. Decide on Selection Function and Distinguisher: First we decide on a suitable **selection function** and **distinguisher**.

    The **selection function** is more or less used to emulate a 'power consumption model' and is denoted as $select(o)$ and selects a (bit of a) sensitive intermediate value. In other words $select\left(f(x_i, k_j)\right)$.

    - Let us choose the Least Significant Bit (LSB) as our selection function.

    The **distinguisher** is used as a test to find the right key.

    - Let us choose the *Distance-of-Means (DoM)* as described below.

6. Calculate Hypothetical Intermediate Values: We now use the above choices to calculate hypothetical intermediate values for every possible (partial) subkey choice where we write the L possible choices as a vector of subkey hypotheses $K = (k_1, .., k_L)$ effectively representing the brute-force keyspace for (part of) k_j. Given data vector X and key hypotheses vector K we can calculate hypothetical intermediate values $f(x_i, k_j)$ for all runs and for all hypotheses resulting in prediction matrix V of size N x L where $V_{ij} = f(x_i, k_j), 1 \leq i \leq N, 1 \leq j \leq L$.

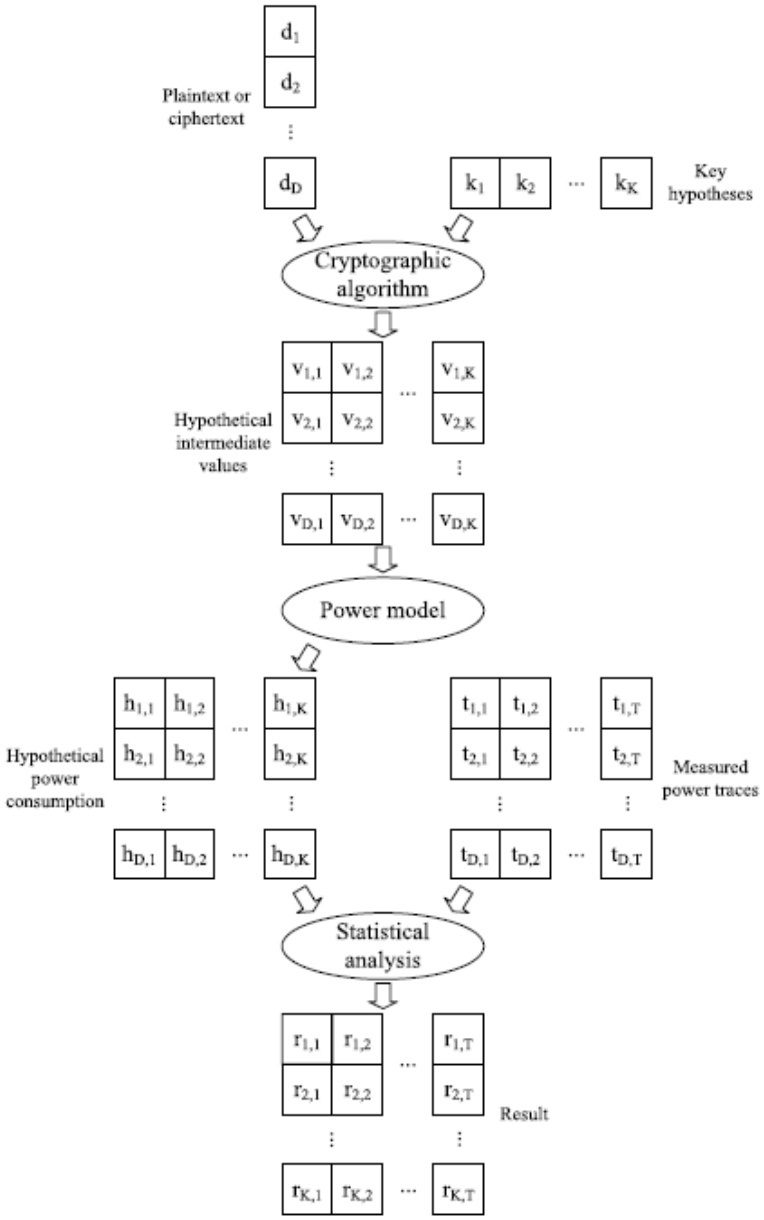
    Here column j of V contains the intermediate prediction results calculated based on key hypothesis k_j. Given that we exhaustively enumerate all k_j, one column of V will have to contain the intermediate values calculated in the device during our measured runs. The goal of our DPA attack is to find out which column of V corresponds to what has been processed during our runs and hence retrieve the corresponding (partial) round key.

7. Mapping Hypothetical Intermediate Values to Hypothetical Power Consumption Values: We now map our matrix of hypothetical intermediate values V to a matrix of hypothetical power consumption values H using our **selection function** as follows: $H_{i,j} = select(V_{i,j})$. Note that in practice this step can often be combined with the previous one by directly building H through applying the selection function immediately to the intermediate function result rather than building V first.

8. Comparing Hypothetical Power Consumption Values: We now have to column-wise (as each column represents a subkey candidate) compare our hypothetical matrix H with our measured matrix M resulting in comparison matrix R of size L x T. We do this by applying our **distinguisher** as discussed below.

   All correlation-coefficient based distinguishers (used in CPA, see below) have the property that the higher the value of $R_{i,j}$ the better columns H_i and M_j match. Hence the correct (partial) subkey can thus be revealed since the highest value of the matrix R will be $R_{ck,ct}$ where ct is the moment in time where the intermediate value we target is being processed and ck is the correct (partial) subkey. We can thus select the highest value from the matrix and find the correct subkey and processing moment in time by index.

   For other distinguishers this is not necessarily the case and the meaning of values of R depends on the distinguisher.



Note that the above approach targets only part of the subkey and hence would have to be repeated targeting other intermediate values in order to recover the full subkey and, subsequently, would be repeated again for other subkeys to recover the full key.

In step 5 we have to decide on a **selection function** and a **side-channel distinguisher** both of which have multiple variants:

Selection functions:

- MSB (1 or more bits)
- LSB (1 or more bits)
- Etc.

Side-channel distinguishers:

- Difference-of-Means
    - Here we determine the relation between the columns of prediction matrix H and measurement matrix M. The matrix M is split into two sets of rows. The split is done based on the columns of the prediction matrix H. Thus for every key hypothesis (ie. each column H_i in H) we split M into M_0 and M_1 with M_0 consisting of all rows in M whose indices correspond to the zeros in H_i and M_1 consisting of the rest (thus effectively sorting by our selection function).

      We then calculate the mean-vector $\mathrm{mean0}$ of the rows in the first set M_0 and the mean-vector $\mathrm{mean1}$ of the rows in the second set M_1. Each of these mean-vectors will be of size T. We then assign $R_{i,j} = |mean0[j] - mean1[j]|, 1 \leq j \leq T$ (ie. each row of result matrix R represent a difference of means trace.

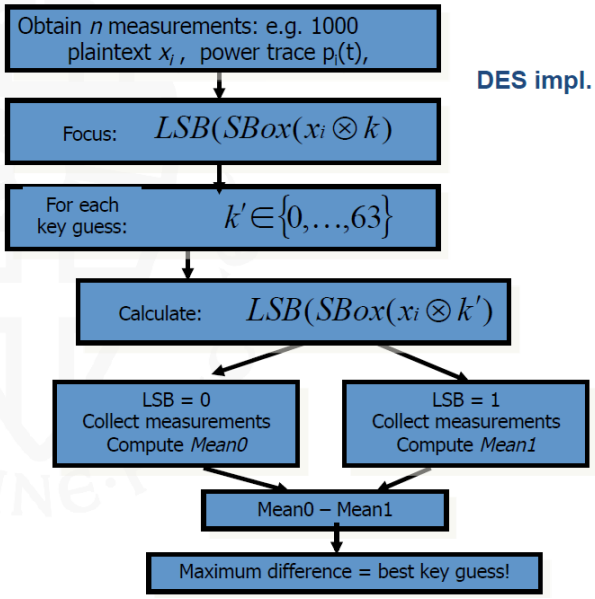
      If key hypothesis k_i is correct there is a significant difference between the corresponding M_0 and M_1. When identifying the correct (partial) subkey we look for the entry in R with the highest difference of means and the resulting $R_{i,j}$ will give us index i indicating the best candidate partial subkey and index j indicating the moment in time our target intermediate result was likely executed.

      Note, however, that the difference-of-means only considers differences of values and not variances. The Distance-of-Means does take variances into account.

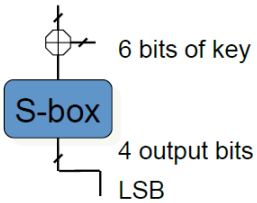
- Distance-of-Means
    - The Distance-of-Means (DoM) is an improvement over the Difference-of-Means in that it takes variance into account. We now calculate our result matrix R as $R_{i,j} = \frac{m_{1i,j} - m_{0i,j}}{S_{i,j}}$ where $S_{i,j}$ is the standard deviation of the difference distribution of the two sets.

- Generalized Maximum-Likelihood Testing
- Etc.

Classical 1-bit DPA on DES using DoM



DES impl.



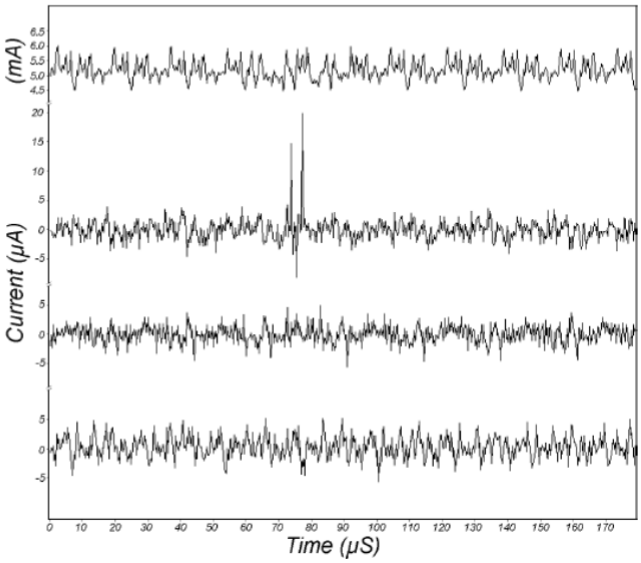
100 measurements *
time window *t* *
64 key guesses

[Kocher et al.]

DPA Result Example

average power consumption

- Δ with correct key guess
- Δ with incorrect key guess
- Δ with another incorrect key guess

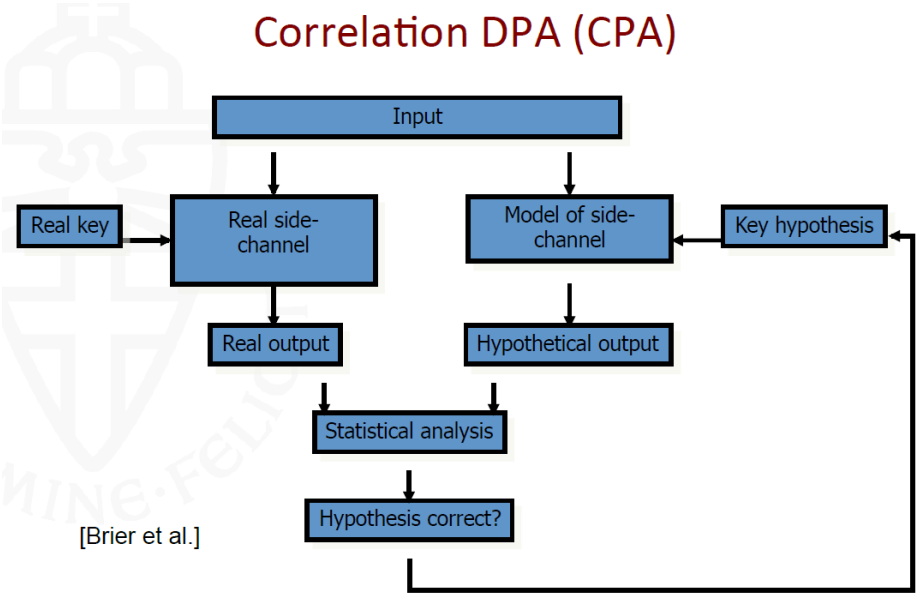


3.1.2.3 Correlation Differential Power Analysis (CPA)

Correlation Power Analysis (CPA) is an extension of DPA where a model of the power consumption is created for use in the analysis phase of an attack, differing from regular DPA in its choice of **selection function** and **side-channel distinguisher**. Here instead of using the binary-only mappings as **selection function**, a more realistic and unconstrainted power consumption model is introduced. The idea is that power consumption is only indirectly related to the data that is being processed, but this can be estimated better by taking into account the type of device. For instance, for register outputs in ASICs, only the number of transitions from 0 to 1 and from 1 to 0 are relevant. All bits contribute equally to the power consumption, both kinds of transitions consume the same amount and static power can be ignored. This is called the *Hamming distance model* and typically applies to hardware implementations. More precisely, let r denote the previous register value. Then $V(x_i, k_0) = \mathrm{HD}(S(x_i \oplus k_0), r)$. For other power consumption models see below.

The model needs to approximate the power consumption of the target cryptographic device during an encryption operation. The resulting power predicted by the model will then be correlated to the actual measured power consumption using a key hypothesis. The highest peak of the correlation plot gives the correct key hypothesis.

Another difference is that CPA uses a correlation coefficient (usually Pearson's correlation coefficient) as a **distinguisher** (see below).



Power Consumption and Power Models
Simple model for power consumption:

$$P(t) = \sum_{g} f(g,t) + N(t)$$

f(g,t) power consumption of gate *g* at time *t*

Total power consumption of a cell is the sum of its static and dynamic power consumption: Ptotal = Pstatic + Pdynamic. Dynamic power consumption is the dominant factor in Ptotal for CMOS.

- Hamming weight model
  - Assuming power is proportional to the number of ones in the processed value
  - Ignoring data processed before and after
  - Typically for pre-charged busses, ie. all bits are set to 0 (in v0) before processing value v1: $HD(v_0, v_1) = HW(v_0 \oplus v_1) = HW(v_1)$.
  - $HW(v)$ = digital sum of all 1s in v (ie. # of symbols different from 0)
- Hamming distance model
  - Most commonly used by attackers
  - Counts # bitflip (1->0, 0->1) transitions
  - Assuming same power consumed for both
  - Typically for register outputs in ASIC's
  - Ignoring static power consumption
  - $HD(v_0, v_1) = HW(v_0 \oplus v_1)$ (ie. # of positions at which corresponding symbols are different)
  - Requires knowledge of preceding or succeeding v
  - Typically v_0 is some (constant) reference state *R* which is not necessarily 0 (as this would reduce the model to the Hamming Weight model) but is the same for each power trace provided the target intermediate result is executed at the same moment in time.
- Weighted hamming weight/distance models
- Signed hamming distances (0->1 neq 1->0)
- Dedicated models for combinational circuits

Selection Functions and Side-Channel Distinguishers for CPA
Selection functions (power consumption model):

- See above

Side-channel distinguishers:

- Pearson's Correlation Coefficient
  - Pearson's Correlation Coefficient reflects the degree of linear relationship between two variables X and Y.
    Let X and Y denote variables, let E denote the expectation value, μX the mean of X, and σX the standard deviation of X. Then Pearson's correlation coefficient ρ can be computed as $\rho(X,Y) = \frac{covariance(X,Y)}{\sigma X,\sigma Y} = \frac{\mathbb{E}((X-\mu X)(Y-\mu Y))}{\sigma X \sigma Y} = \frac{\sum_{i=1}^{n}(X_i-\mu X)(Y_i-\mu Y)}{\sqrt{\sum_{i-1}^{n}(X_i-\mu X)^2}\sqrt{\sum_{i-1}^{n}(Y_i-\mu Y)^2}}$ with $-1 \leq \rho \leq 1$.

    When using this as a distinguisher we are given our hypothesis matrix H and our measurement matrix M and for each key hypothesis $k_i, 1 \leq i \leq L$ we take the corresponding column H_i and then, for each moment in time $1 \leq j \leq T$, we take the corresponding column M_j and compute Pearson's Correlation Coefficient over the columns $\rho(H_i, M_j)$ and assign its absolute value to result matrix $R_{i,j}$.

- Student's t-test
- Mutual Information
- Principal component analysis
- Etc.

3.1.2.4 Higher-Order DPA

Consider an (intermediate) cryptographic result protected against regular DPA by boolean masking (discussed later):

$$x' = x \oplus m$$

For example given plaintext input x and key k we generate a random mask m to mask the result of an intermediate operation *f*:

$$c = f(x \oplus m, k)$$

Masking will protect against first-order DPA attacks, but higher-order attacks that combine multiple samples from within a power consumption trace are still possible and have been quite successful. More than one mask can be added to protect against second-order attacks. In general, an n^{th} order masking scheme will protect against an n^{th} order DPA attack (and lower), but can be broken by an n^{th}-order attack (or higher).

Higher-Order DPA (HODPA) correlates power consumption more than once per computation and generalized (first-order) DPA attacks by considering simultaneously k samples, within the same power consumption trace, that correspond to k different intermediate values. These HODPA attacks are based on the joint statistical properties

of multiple aspects of the signal, typically joint analysis of the power consumption at two (or more) points in time. In masking individual measurements are correlated to the parts but uncorrelated to the variable. A higher-order function can combine a measurement correlated to the first part with a measurement correlated to the second part, so that the combination is correlated to the sensitive variable. For example if two points are correlated to A and B respectively and the secret intermediate is $A \oplus B$ the product of points A and B will show correlation to $A \oplus B$ and can be used to test hypotheses about the secret. Most established HODPA techniques rely on a pre-processing step to map the multivariate problem to a univariate problem before attacking the result with a standard DPA attack.

The mounting point for 2nd order attacks is the fact that the side-channel leakage Y of a masked value depends on a predictable value X and an unpredictable value M. The core idea is to jointly analyze the leakage of the masked value and the mask (or a second value masked with the same mask) to establish a relation to the predictable X or its predictable leakage.

For example, consider the leakage of the masked output $L_{K,M}$ and the leakage of the mask L_M then 2nd order DPA requires a suitable pre-processing that combines the leakage of two masked RVs to construct a signal that is correlated to the unmasked intermediate result (or a function thereof) and can be attacked with 1st order DPA. More formally that is: one looks for functions $g(L_M, L_{K,M})$ and $h(X)$ that yield highly correlated values. These values can be attacked with 1st order DPA.

3.1.2.5 DPA / CPA Summary

- Attack has 2 parts
  - 'cryptanalysis': target intermediate result for which exhaustive key search is feasible
  - Engineering statistics: provide access to an oracle that verifies sub-key hypotheses using power traces
- Working principle
  - Acquisition part: collect a set of traces with varying inputs
  - Select sensitive intermediate variable
  - For each key hypothesis
    - Compute hypothetical values of sensitive variable, sort curves into subset
    - Compute differences between subsets
- Intuition
  - Wrong key guesses -> no correlation P vs model, no peak
  - Correct guesses -> good correlation P vs model

Practical attacks: platforms & distinguishers

- Platforms
  - 4-bit, 8-bit, 32-bit, contactless Java cars
  - ASICs, FPGAs

- All algorithms: secret key, public key, stream ciphers, MACs
- Side-channel distinguishers
    - Used as the selection function but also to assist other attacks eg to find interesting points in time
    - Distance of means (DoM) with single-bit or multi-bit, pearson correlation coefficient, student's t-test, principal component analysis (PCA), variance based, etc.

3.1.2.6 Advanced Attacks

Template Attacks

- Assuming attacker has access to same device as one under attack
- Consists of 2 phases: characterization and key recovery
- Characterization phase
    - Templates for certain sequences of instructions
    - Obtaining templates for every pair of data and key

Collision Attacks

Exploits the fact that in two algorithm executions, a certain intermediate value Vij can be the same. Vij = f(Di, Kj) = f(Di*, Kj).

3.1.3 Countermeasures

Purpose of counter-measures is to destroy the link between intermediate values and power consumption.

- Masking / Blinding
    - A random mask concealing every intermediate value
    - Can be on all levels (arithmetic -> gate level)
- Hiding
    - Making power consumption independent of the intermediate values and of operations
    - Special logic styles, randomizing in time domain, lowering signal-to-noise ratio

Software Countermeasures

- Time randomization: the operations are randomly shifted in time
    - Use of NOP operations / add random delays
    - Use of dummy variables / instructions
- Register renaming and nondeterministic processor
    - Processor selects instruction and memory access randomly
- Permuted execution
    - Rearrange instructions / S-boxes. By randomizing the order in which manipulations are being done an attacker cannot assume anything about the order of data being processed (eg. *n* bytes being XORed with *n* key bytes) which prohibits determining which byte has been XORed at which time and aids in defense against eg. DPA.

- Masking techniques
- Eliminating secret data-dependent branching
- Masking/Blinding
- Application-level countermeasures
    - Retry counters that limit the number of samples that an attacker can take. A PIN verification that blocks after eg. 3 successive failures is a useful protection against DPA.
    - Limited control and visibility of input and output to cryptographic operations. An attack may not be able to carry out DPA if only part of the input can be chosen or only part of the result is returned.

Hardware Countermeasures

- Noise Generation
    - HW noise generator would include use of RNG
    - Total power consumption is increased (problem for handheld devices)
- Desynchronization
    - Introducing some fake clock cycles during computation or using a weak jitter
- Power Signal Filtering
    - Ex.: RLC filter smoothing the power consumption signal by removing high frequency components
    - One should use active components (transistors) in order to keep power cons. Relatively constant – problem for mobile phones
    - Detached power supplies - Shamir
- Novel Circuit Designs
    - Special logic styles (using constant amount of power)
- Dynamic and different logic (pre-charged dual rail)
    - Duplicate logic
    - Bits are encoded as pairs eg. 0 = (1,0) and 1 = (0,1)
    - Circuit is pre-charged eg. to all zero (0,0)
    - Each DRP gate toggles exactly once per evaluation
    - Hence number of bitflips is constant and data-independent
- Masking/Blinding

3.1.3.1 Masking/Blinding

Masking seeks to break the relation between algorithmic and intermediate values. A random mask conceals every intermediate value (which can be done on all levels from arithmetic to gate level) by randomizing them. Since new masks are randomly chosen for each new run simple statistical analysis of power consumption does not lead to secret recovery. However 2nd and higher order DPA (as discussed above) are still possible by looking at the joint probability distributions of multiple samples from within a trace. Some examples of masking are:

- *Boolean masking*: $x' = x \oplus r_x$

- *Arithmetic masking*: $x' = x - r_x \ mod \ 2^w$
    - *Multiplicative masking*: $x' = x \otimes r_x$

Note that in general boolean masking is less (or not) suitable for non-linear functions. In linear functions $f(x * y) = f(x) * f(y)$ while in non-linear functions (eg. S-Boxes) $f(x * y) \neq f(x) * f(y)$. Also note how multiplicative masking cannot conceal intermediate value 0.

Consider the following example of masked RSA (protecting against regular DPA) where message *m* is randomized:

$n = p * q$
$e * d = 1 \ mod \ \phi(n)$
Pick random $r < n$
$m_r = m * r$
$v = m_r^e \ mod \ n$
$u = r^e \ mod \ n$
$c = v * u^{-1} \ mod \ n$

Or this one where private key *d* is randomized:

$n = p * q$
$e * d = 1 \ mod \ \phi(n)$
Pick random $r < n$
$d_r = d + r * \phi(n)$
$s = m^{d_r} \ mod \ n$

Regarding the susceptibility to regular or higher-order DPA consider the following:

$f(p)$:
$result = p \oplus secret$
...
$return \ c$

$f(p)$:
$mask = rand()$
$mp = p \oplus mask$
$result = mp \oplus secret$
...
$return \ c$

The example on the left is vulnerable to regular DPA while the example on the right isn't. The example on the right is, however, vulnerable to 2nd order DPA. Consider the points:

$$t_1 : mask = rand()$$
$$t_2 : mp = p \oplus mask$$
$$t_3 : mp \oplus secret = (p \oplus mask) \oplus secret$$

The joint distribution of the power consumption traces in points t_1, t_3 allows bit-by-bit derivation of the secret. Usually additive arithmetic masking is the best solution as calculation of the mask correction (to *unmask* the value when needed) is always linear when using composite fields.

Masking does have some issues however as it requires a TRNG and could possibly leak due to glitches if not implemented correctly. In the absence of TRNG (as is the case with many devices) the mask will have to be drawn from a (CS)PRNG which introduces a 'chicken and egg' problem in that the attacker might choose to target the (CS)PRNG (eg. a block cipher in CTR mode with a secret seed) using a side-channel attack.

Attacks on Masking/Blinding
Masking/Blinding is vulnerable to more advanced Power Analysis attacks such as Higher-Order DPA (see above) or template-based attacks.

3.1.3.2 Hiding

The goal of hiding countermeasures is to make the power consumption of cryptographic devices independent of the intermediate values and independent of the operations that are performed. There are essentially two approaches to achieve this independence. The first approach is to build devices in such a way that the power consumption is random. This means that in each clock cycle a random amount of power is consumed. The second approach is build devices that consume an equal amount of power for all operations and for all data values. Hence, equal amounts of power are consumed in each clock cycle.

The goal of perfect randomization cannot be reached in practice but getting close to this goal can be done in roughly two ways:

- Randomizing power consumption by performing cryptographic operations at different moments in time during each execution, affecting only the time dimension.
- Changing the power consumption characteristics of the performed operations, affecting the amplitude dimension.

Examples are:

- Time Dimension
    - *Dummy Operations*: nop operations, random delays, redundant data representations, the use of dummy variables and instructions, etc.
    - *Shuffling*: Changing the order of independent instructions to reduce correlation.

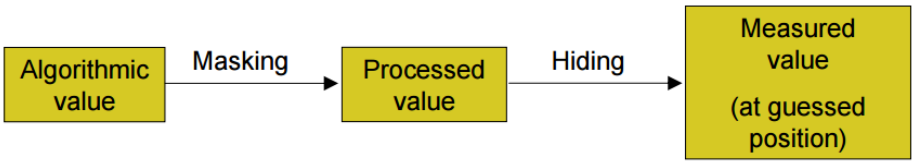
- Amplitude Dimension

This largely consists of *changing the Signal-to-Noise Ratio*: Noise generation (random switching activity dominates power consumption), Reducing the signal (device where all operations require equal amount of power). Usage of active components (eg. transistors) in order to keep power consumption relatively constant, usage of a detached power supply, RLC filter to smooth the power consumption signal by removing high frequency components, etc.

Examples of signal reduction:

- *Special logic styles*: duplicate logic, pre-charged circuitry, bits encoded as pairs eg. 0=(1,0) and 1=(0,1), etc. The number of bit flips is constant and data-independent.

- *Power Signal Filtering*: Usage of active components (eg. transistors) in order to keep power consumption relatively constant, usage of a detached power supply, RLC filter to smooth the power consumption signal by removing high frequency components, etc.



Attacks on Hiding

- Dealing with misaligned traces: Misaligned traces can be dealt with through realignment which is done in two steps: first a pattern in the first trace is selected, secondly the attacker tries to find this pattern in the other power traces and determines the position where the pattern fits best, shifting the other traces in such a way that the pattern occurs at the same position in all traces.

3.1.4 Conclusion and Open Problems

- Physical access allows many attack paths
- Trade-offs between assumptions and computational complexity
- Requires knowledge in many different areas
- Combining SCA with theoretical cryptanalysis

3.2 Fault Injections & Analysis

Passive vs Active

	Passive	Active
Operating	within specification	possibly outside specification
Signal	hidden	insertion
Attack	observe	influence & observe
Example	eavesdropping, side channel	fault injection, FIBbing

Invasive vs Non-Invasive

- Non-invasive: violates tamper-evidence
  - Low-cost
  - Power/EM measurements
  - Data remnance in memory
  - Glitching: clock/power supply
  - Counter-measures: sensors for V, T, f
- Invasive: strongest type
  - Expensive
  - Microprobing, Focused Ion Beam (FIB) probing, optical reverse engineering
  - Long preparation phase
  - Counter-measures: feature size, multi-layering, protective layer (active grid, seemingly non-correlated signals), sensors, bus scrambling, glue logic
- Semi-Invasive: device is depackaged but no contact to the chip
  - Eg. Optical attacks that read out memory cells
  - Faults/glitches by voltage, power supply, clock, EM, etc.
  - Using UV light, laser, imaging, optical fault injection

Fault Injection

Mess with environmental conditions to induce fault in execution eg.:

- Clock frequency
- Voltage
- Temperature

May require depackaging but is not always tamper-evident.

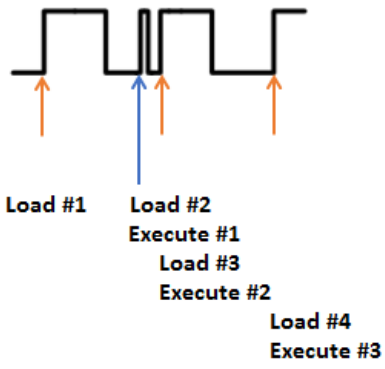
3.2.1 Methods for Fault Injection

Methods

- Card Tears

- Physical
  - Putting a 0 or a 1 on a databus line
- Glitching
  - Causing one or more flipflops or instruction jumps
  - Affect EEPROM & ROM
  - Skipping instructions on a javacard VM seems possible
  - Most useful practical fault-injection attack

  - Variation in supply voltage
    - Underpowering or overpowering
    - Well-timed power spikes or ‘brown-outs’ can cause a processor to skip instruction
    - Circuit injecting fault must be driven by same clock as target
    - Difficulty increases with clock speed
    - Actively investigated by smartcard industry
  - Variation in external clock
    - Shorten length of single cycle through feeding premature clock cycle toggle
    - May cause data misread or an instruction miss
    - Can only target chips with external clock line (eg. smartcards) not those with own clock line since disconnecting clock line from circuit is difficult
  - Change in temperature
    - Change in RAM content
    - Write operations work better
  - White light (camera flash)
    - Photons induce faults
  - X-rays and ion beams
    - FIBs can be used to induce alterations in circuit by physically modifying their structure (reconstructing missing busses, cutting wires, milling through layers or reconstructing them, etc.)



Fault injection on clock causing instruction miss by shortening cycle

Table I
FAULT INJECTION TECHNIQUES SUMMARY

Technique	Accuracy [space]	Accuracy [time]	Technical skill	Cost	Hindered by technological advances	Requires knowledge of the implementation	Damage to the device
Underfeeding	high	none	basic	low	no	no	no
Clock glitch	low	high	moderate	low	yes	yes	no
EM Pulses	low	moderate	moderate	low	no	no	possibly
Heat	low	none	low	low	partial	yes	possibly
Power supply glitch	low	moderate	moderate	low	no	partial	no
Light Radiation	low	low	moderate	low	yes	no	yes
Light Pulse	moderate	moderate	moderate	moderate	yes	yes	possibly
Laser beam	high	high	high	high	yes	yes	possibly
Focused Ion Beam	complete	complete	very high	very high	yes	yes	yes

Goals

- Insert computational faults
  - Null key (exploiting two keys being combined in the wrong way)
  - Wrong crypto result (Differential Fault Analysis – DFA)
- Change software decisions
  - Force approval of false PIN
  - Reverse life cycle state
  - Enforce access rights
- Etc.

Targets

- Attacks can be on code or data
  - Including CPU functionality and data, eg. 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
  - Result in random values
- Targeting crypto
  - Can cause processor to skip instruction
  - Some crypto-algorithms are sensitive to bitflips (classic example is RSA)
- Targeting other functionality
  - Any security-sensitive part of the code or data can be targeted
  - Smartcard platform can take care of some of this but every programmer may have to ensure this for each program seperately

Practical Complications

- Many parameters for attacker to play with
  - When to do card tear?
  - When to glitch and for how long?
  - When and where (x and y dimension) to shoot laser?

    - For how long, how strong and which colour?
- Multiple faults?
    - Multiple glitches are possible
    - Multiple laser attacks are harder
- Fault attacks can be hit-and-miss process

3.2.2 BellCoRe Attack and Differential Fault Analysis (DFA)

The BellCoRe attack is a fault injection attack (a form of *Differential Fault Analysis* (DFA) to be precise) targeting RSA-CRT (RSA with Chinese Remainder Theorem (CRT)) implementations and consists of revealing primes p and q by faulting the computation. It is quite powerful as it works even with very random faulting. Consider RSA-CRT:

$$\boldsymbol{in}: message\ m, private\ key\ (p, q, d_p, d_q)$$
$$\boldsymbol{out}: signature\ s$$

$$\boldsymbol{Precompute}$$
$$d_p = d\ mod\ (p-1)$$
$$d_q = d\ mod\ (q-1)$$
$$\mathrm{k} = p^{-1}\ mod\ q$$

$$\boldsymbol{Exponentiation}$$
$$s_p = (m)^{d_p}\ mod\ p$$
$$s_q = (m)^{d_q}\ mod\ q$$

$$\boldsymbol{Recombination}$$
$$s = \Big(\big((s_q - s_p) * k\big)\ mod\ q\Big) * p + s_p$$

Suppose s_p or s_q is computed with a fault. Assume that the resulting faulty signature s' together with the correct signature s are known to the attacker. Then he can retrieve the private key by computing $gcd(s - s', N)$ for the publicly known RSA modulus N. Alternatively the attacker can recover the private key from s' and m by computing $gcd(m - ((s')^e\ mod\ N), N)$ where e is the public exponent.

This holds since, for an injected fault in s_q resulting in ${s_q}'$ and corresponding s' we have:

$$s = \Big(\big((s_q - s_p) * k\big)\ mod\ q\Big) * p + s_p$$
$$s' = \Big(\big(({s_q}' - s_p) * k\big)\ mod\ q\Big) * p + s_p$$
$$s - s' = \Big(\big((s_q - s_p) * k\big)\ mod\ q\Big) - \Big(\big(({s_q}' - s_p) * k\big)\ mod\ q\Big)$$

Hence

$$p = \gcd(n, s - s')$$

And for an injected fault in s_p resulting in $s_p{}'$ and corresponding s' we have:

$$s = \Big(\big((s_q - s_p) * k\big) \ mod \ q\Big) * p + s_p$$

$$s' = \Big(\big((s_q - s_p{}') * k\big) \ mod \ q\Big) * p + s_p{}'$$

$$\begin{aligned} s - s' &= (s_p - s_p') + \Big(\big((s_q - s_p) * k\big) \ mod \ q\Big) * p - \Big(\big((s_q - s_p) * k\big) \ mod \ q\Big) * p \\ &\equiv (s_p - s_p') + (p * k \ mod \ p)(s_q - s_q \ mod \ q) - (p * k \ mod \ p)(s_q - s_q{}' \ mod \ q) \\ &\equiv 0 \ mod \ p \ \{since \ (p * k \ mod \ p) = (p * p^{-1} \ mod \ p) = 1 \ mod \ p\} \end{aligned}$$

Hence

$$q = \gcd(n, s - s')$$

3.3 Fault Injections & Defensive Coding

Physical Countermeasures

- Prevention: Make it hard to attack a card
- Detection: Include detector that can notice an attack, eg. light detector or power supply dip detector. Starts another arms race.

Logical Countermeasures

- Program defensively: To not leak info or to resist faults for Javacard, this can be at platform or applet level

Example

```
class OwnerPIN {
 boolean validated = false;
 short tryCounter = 3;
 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 should be a transient boolean array
  - Ensuring automatic reset to false
- Checking & resetting PIN tryCounter in a safe order
  - To defeat card tear attacks
- Does timing of arrayCompare leak how many digits of the PIN code we got right?
  - Read javadocs for arrayCompare

More paranoid:

- Checking for illegal values of tryCounter
  - Eg. negative or greater than 3
- Redundancy in datatype representation
  - Eg. record tryCounter*13 or use error-detecting/correcting code
- Changing order of tests
  - Thinking of how this looks in bytecode
- Keeping two copies of tryCounter
  - Even better keep one of these in RAM, initialized on applet selection hence attacker must attack both RAM and EEPROM synchronously
- Doing security sensitive checks twice

```
if (pinOK)   { // allow acces
                    ...
             }
       else { // error handling
                    ...
             }
```

Better

```
if (!pinOK) { // error handling
                    ...
            }
      else { // allow access
                    ...
            }
```

Even more paranoid

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

Better to branch to the good (ie. dangerous) case if faults can get the card to skip instructions.

Defensive Coding Tricks

- Avoiding use of special values such as 0x00 or 0xFF (don't use javacard Booleans)
- Using restricted domains and testing against them, ideally domains that exclude 0x00 and 0xFF and elements with equal hamming weights
- Introduce redundancy
  - When storing data and/or performing computations
  - Forcing attacker to synchronize attacks (eg. on EEPROM and RAM)
- Jump to good (ie. dangerous) cases
- Make sure code executes in constant time
- Additional integrity checks on execution trace
  - Doing same computation twice and checking results or, when dealing with cryptography, computing the inverse of the result and comparing it against the input to prevent attacks inducing the same fault twice.
  - For asymmetric crypto: use the cheap operation to check validity of expensive one
- Check control-flow integrity
  - Add ad-hoc trip wires & flags in code to conform integrity of run
- Store sensitive data as javacard.security.keys
  - Presumable the (naïve) implementation of this class uses (hardware-specific?) measures to protect integrity & confidentiality of objects

    - A key can also be read or written as byte array, so we can store byte array as a key
- More generally: API implementations must be tuned to hardware vulnerabilities

Coding Trick to remove timing sensitivity

- Instead of

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

do

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

Conclusion

- Any physical device has side-channels
- Information redundancy can increase vulnerability of cryptographic circuit to power analysis
- We not just have to implement right security but implement right security securely
- Technology for fault attacks is improving
- Counter-measures may be at level of hardware, platform & APIs and the application code
- Security through obscurity does have its merits: knowing code of implementation or layout of data in memory really helps attacker with fault attack, obscurity makes life of attacker harder.