

Secure application programming in the presence of side channel attacks

Marc Witteman & Harko Robroch

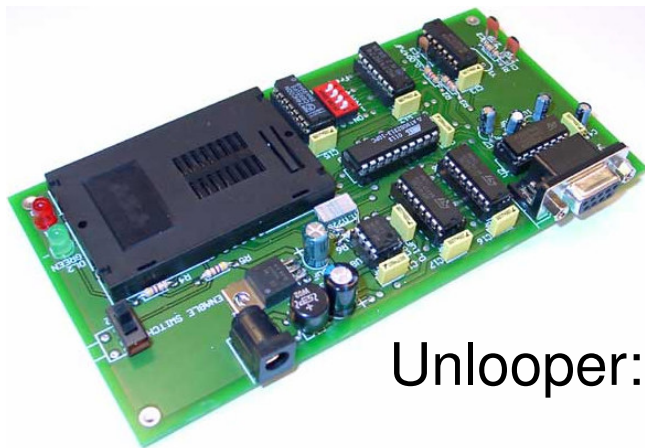
Riscure

04/09/08 | Session Code: RR-203

RSACONFERENCE**2008**

Attacks in the field...

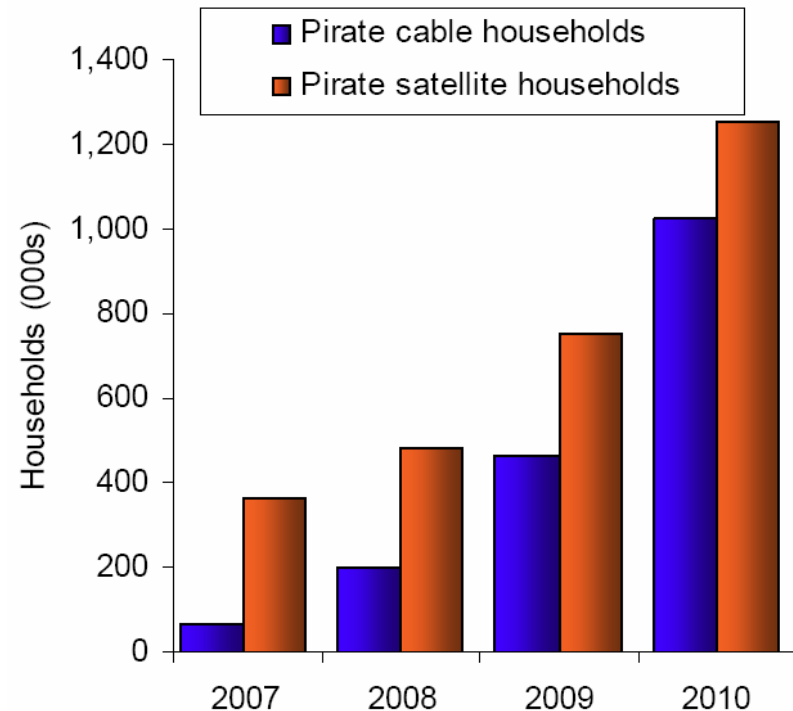
Survey 2007*, Hong Kong:
 Asia-Pacific Pay-TV piracy
 losses top US\$1.5 billion for
 2007



Unlooper: \$100

➔ Fault injection attack

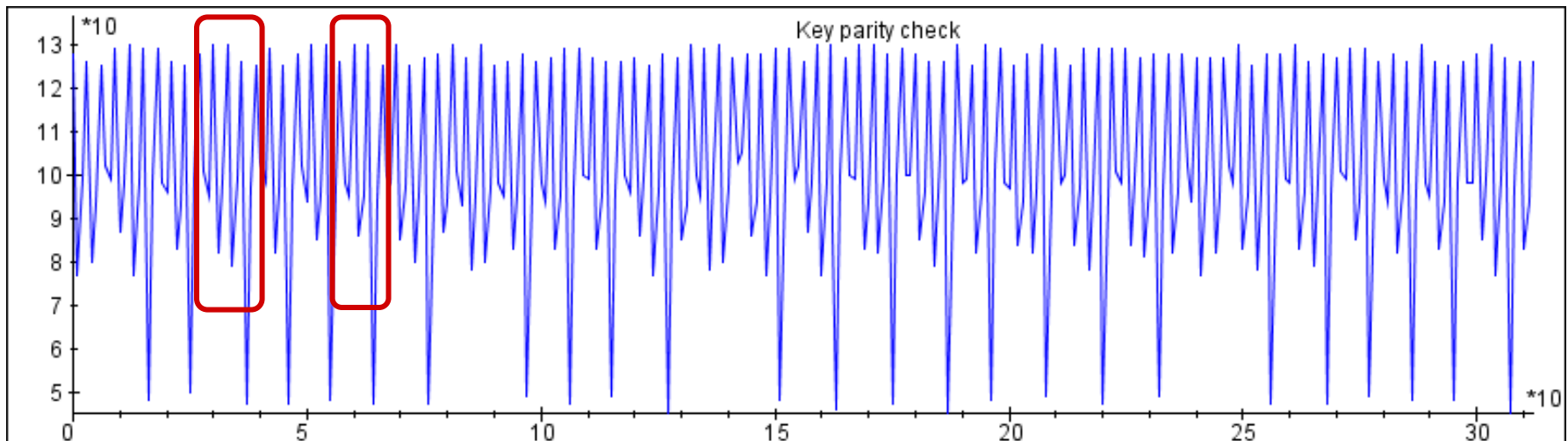
Pirate subscriber forecast in Eastern Europe



Source: Datamonitor

Side channel attack ...

on a standard 3-DES implementation (gnu-crypto)



1 0 1 0 0 0 1 0 0 0 1 1 1 1 1 0 1 1 1 1 1 0 0 0 1

0x45

0x08

0xEF

0x23

→ Time-power analysis

Outline

- Introduction
- What are side channel attacks?
- How to assess the risk?
- Secure programming patterns
- Conclusion

Objectives of this presentation

- Familiarize you with side channel threats
- Provide guidance for side channel protection at application level

Problem definition

- Are you **making** a
 - Set-top box?
 - Cell phone?
 - Printer?
 - Gaming console?
 - And focusing **on logical security**?
- You leave the device **vulnerable** to side channel attacks

Change your protection strategy

Common paradigm

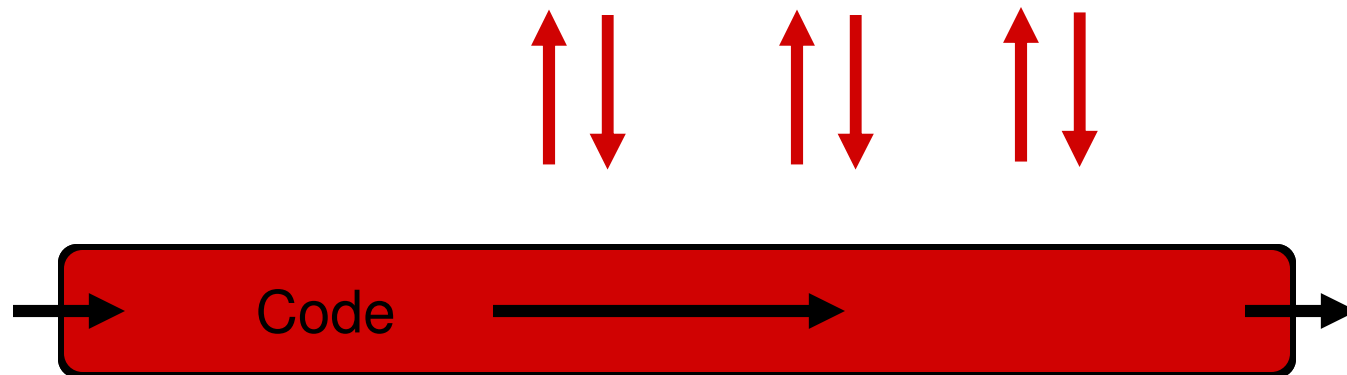
- Do not trust input
- Preserve confidentiality and integrity in output

Local: at input / output

But we also need to

- Beware of leakage
- Protect run-time integrity

Pervasive: throughout code



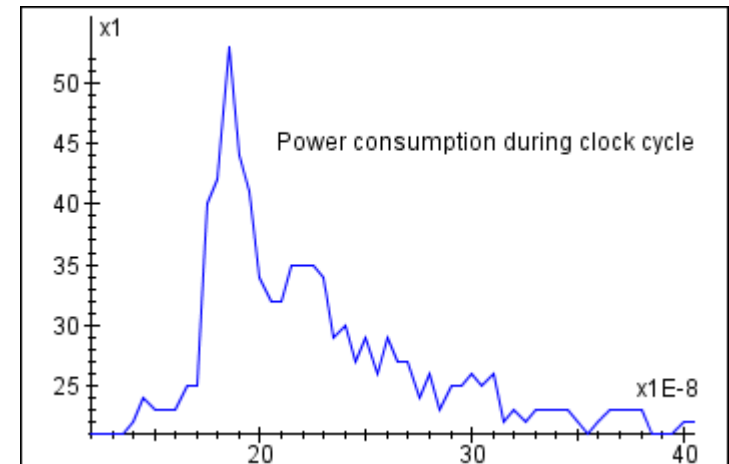
Outline

- Introduction
- What are side channel attacks?
 - Side channel analysis
 - Fault injection
- How to assess the risk?
- Secure programming patterns
- Conclusion

Side channel attacks

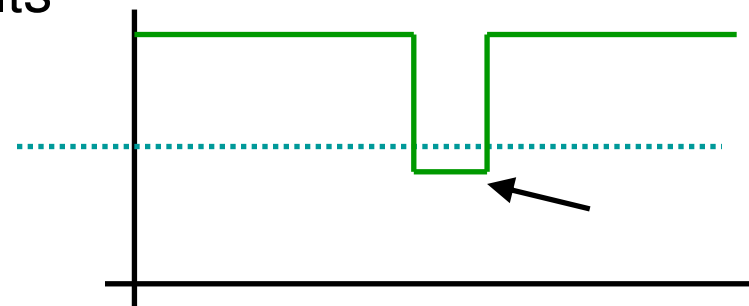
Leakage - Side channel analysis

1. Timing
2. Power consumption
3. Electro-magnetic emission



Fault - Side channel injection of faults

1. Power glitches
2. Optical pulses



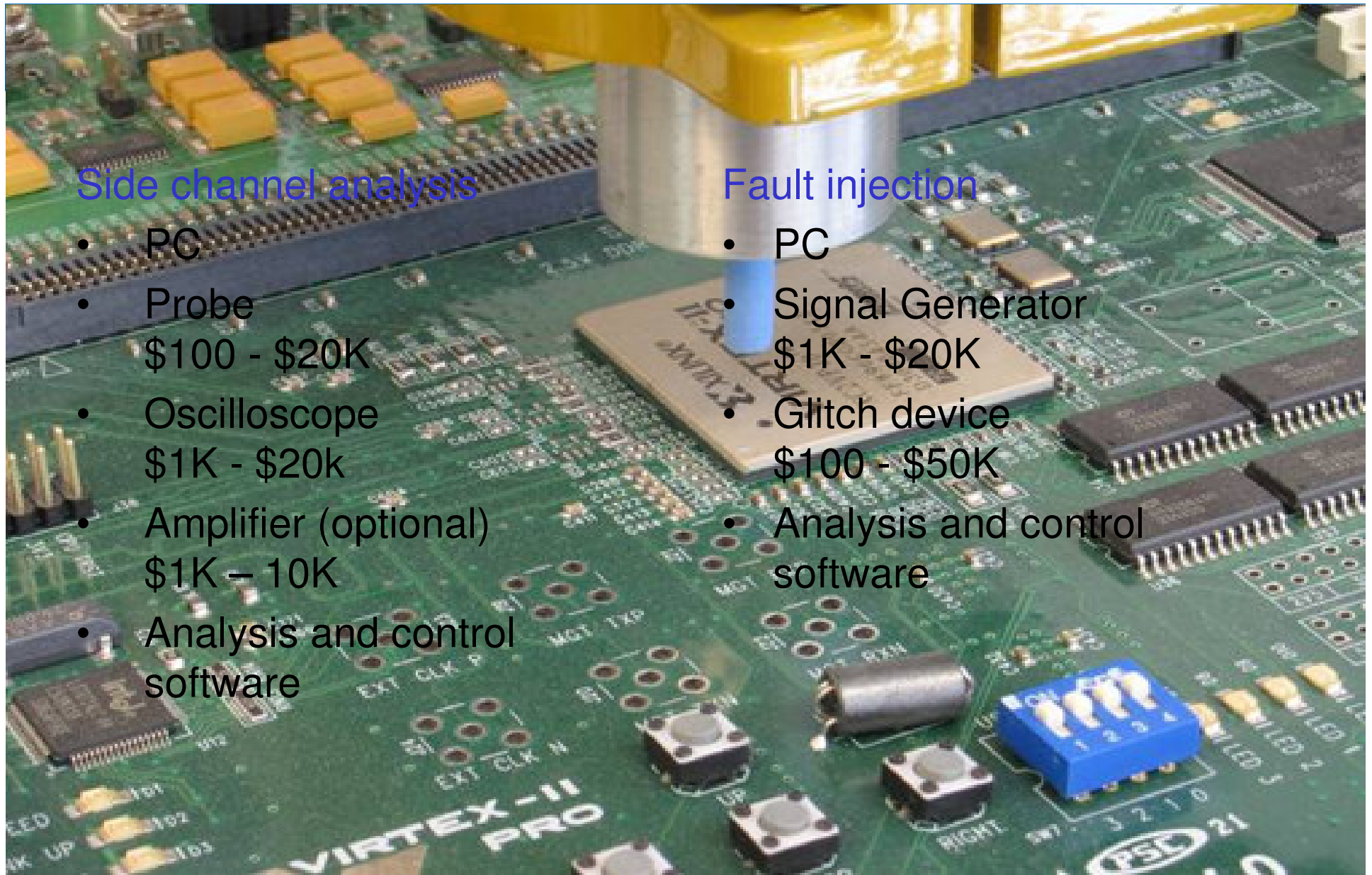
Impact on your device

Leakage

1. Key retrieval
2. Password / PIN retrieval

Fault injection

1. Change a program decision
E.g. unauthorized firmware update
2. Key retrieval
E.g. trigger computational fault



Side channel analysis

- PC
- Probe
\$100 - \$20K
- Oscilloscope
\$1K - \$20k
- Amplifier (optional)
\$1K - 10K
- Analysis and control software

Fault injection

- PC
- Signal Generator
\$1K - \$20K
- Glitch device
\$100 - \$50K
- Analysis and control software

Outline

- Introduction
- What are side channel attacks?
- [How to assess the risk?](#)
- Secure programming patterns
- Conclusion

Applicability to your device

If side channel threats apply, depends on

- Physical access?
- Access time window?
- Interfacing and control?
- Exploitation equipment \$?

First mitigate easier threats!



How to identify weaknesses in your code

Leakage

- Where you **process a secret**
 - Processing → **access, use, verify**
 - Note: also data that depends on a secret

Fault

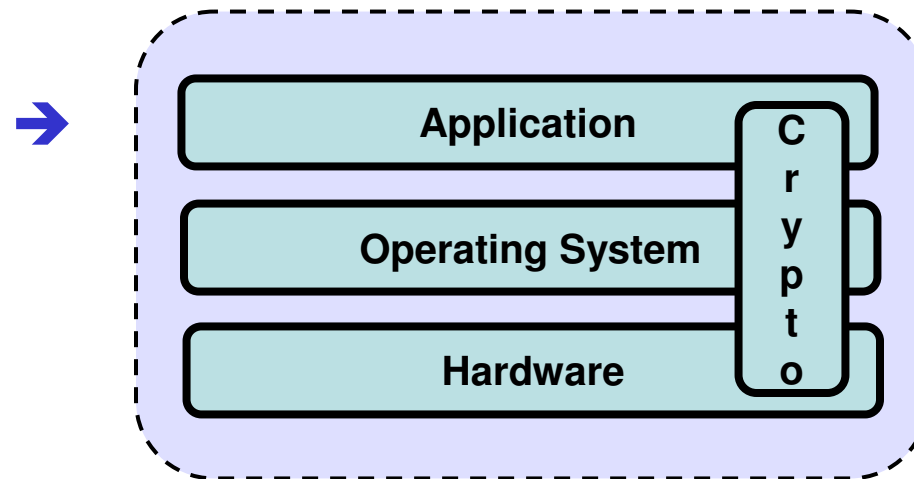
- Security critical **decision point**
 - Example: access control, branches (if, switch)
- Where you **manipulate** sensitive data
 - Example: state, key-dependent data

Outline

- Introduction
- What are side channel attacks?
- How to assess the risk?
- Secure programming patterns
- Conclusion

Approach

Countermeasures at each level of the device



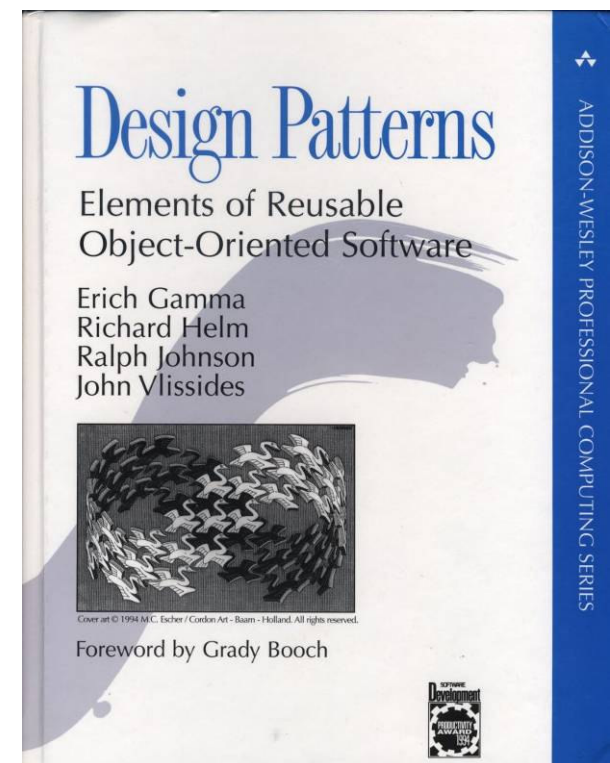
Here, we focus on the [application](#)

Patterns

Philosophy: find **repeatable solutions for common problems** that are optimal rather than perfect

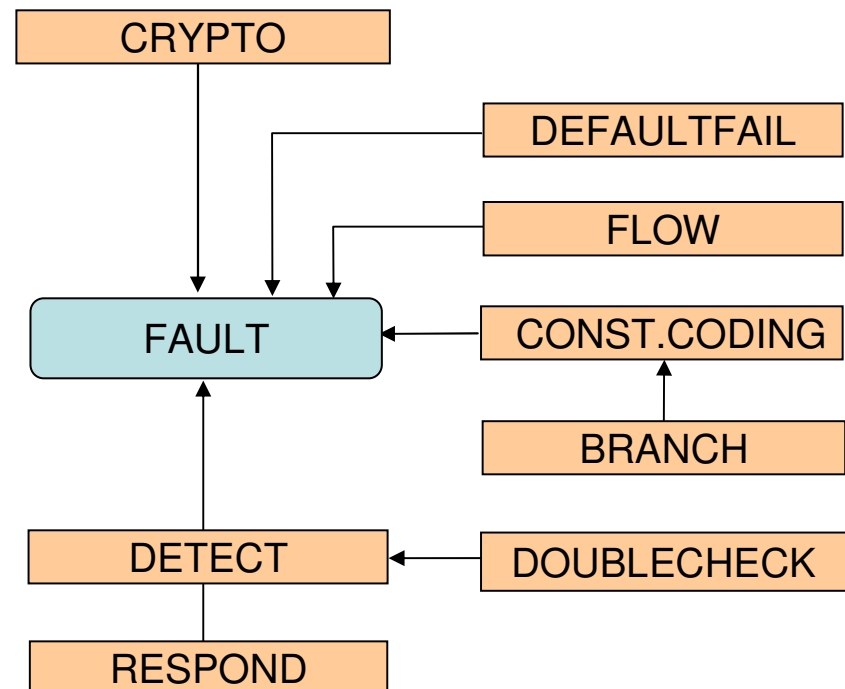
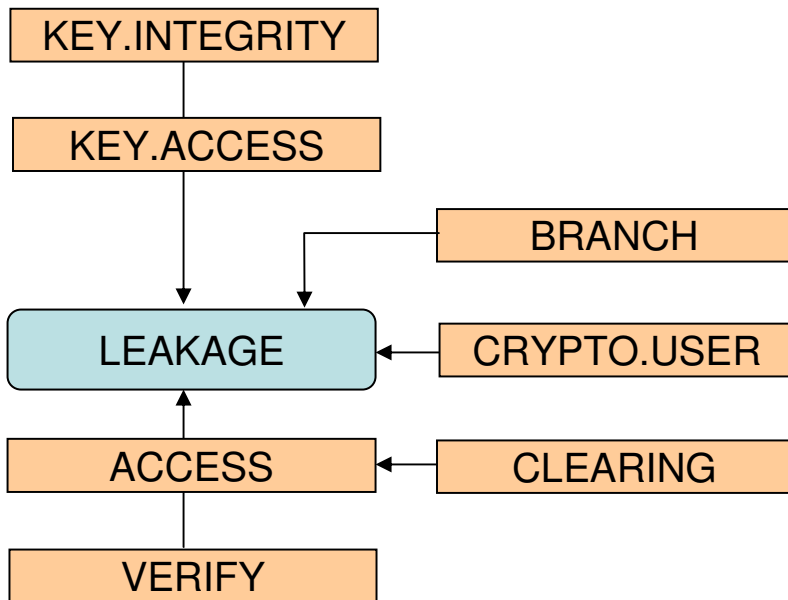
Design Patterns

- Book from 1994
- Recurring patterns in software design

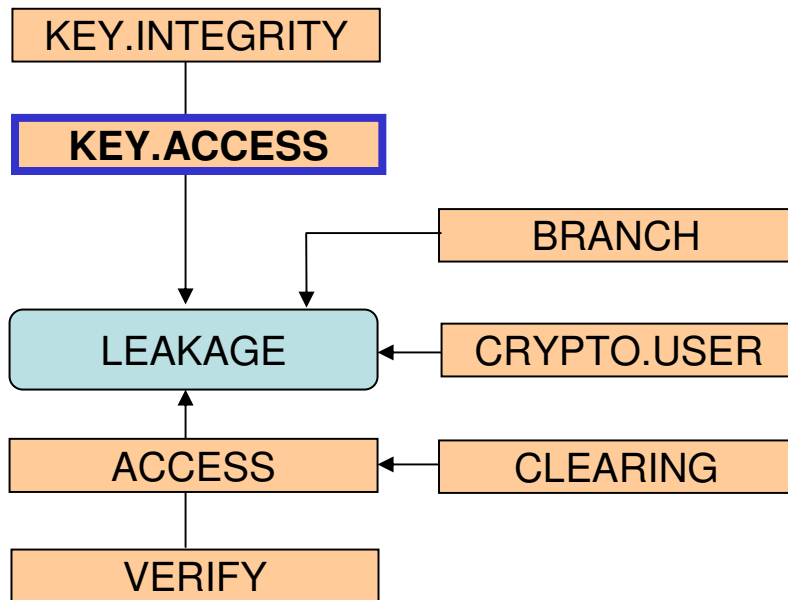


Side channel protection patterns

We define 15 related patterns



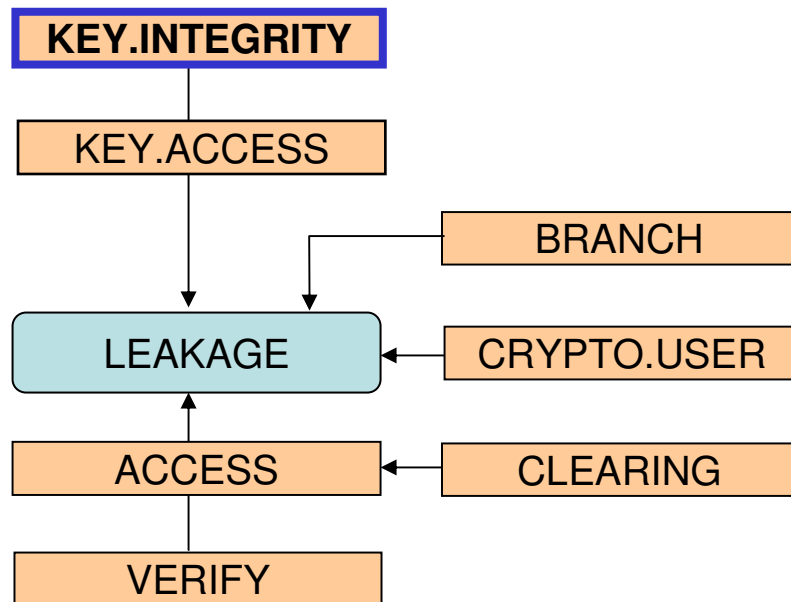
Side channel protection patterns



Use of a key value can cause data leakage

➔ Only use **pointers** to keys

Side channel protection patterns



Parity checking can leak the key

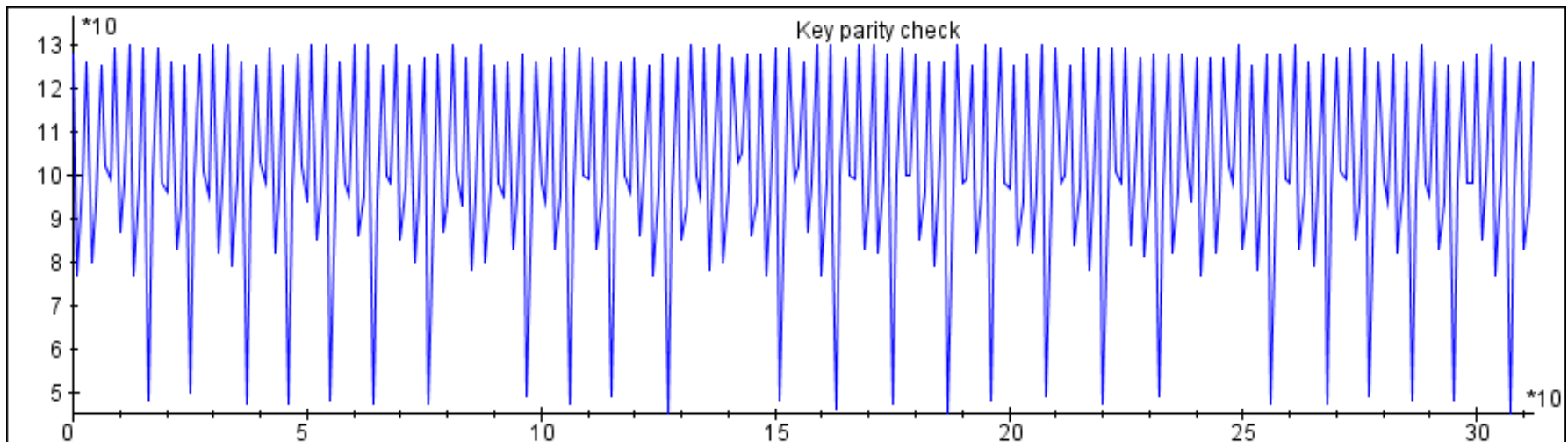
→ Make **time-constant** + **randomize** checking order

→ Or **don't check** the parity

A standard implementation (Gnu Crypto)

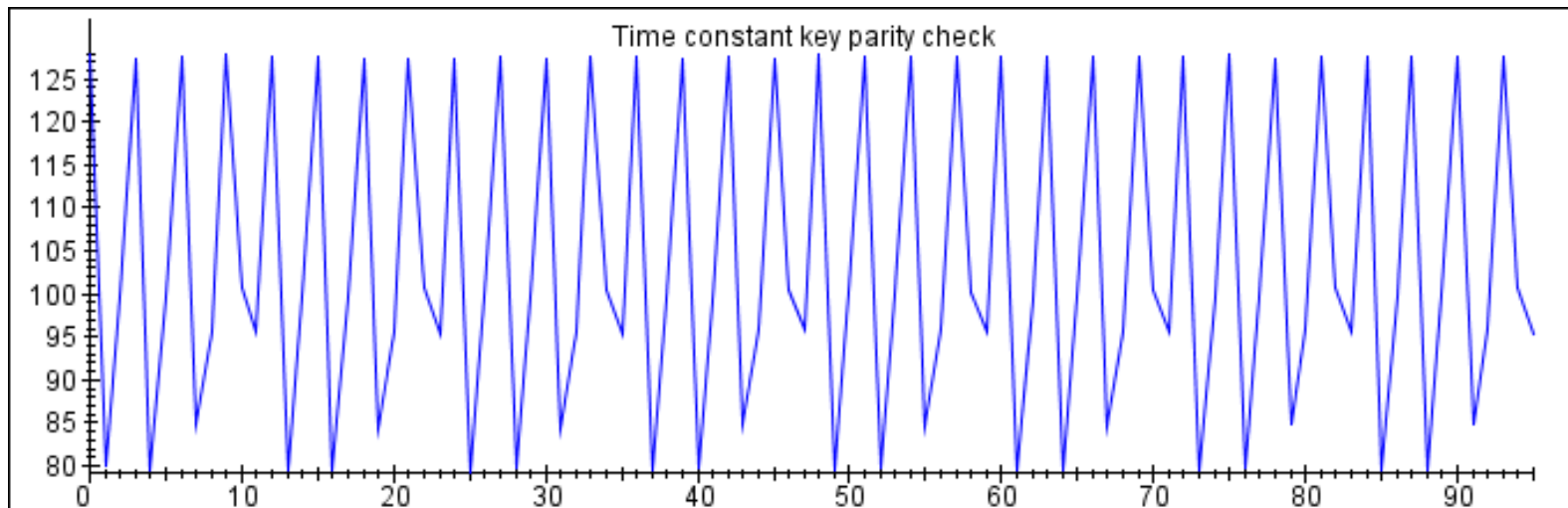
```
public static boolean checkParity ( byte[]key, int offset) {
    for (int i = 0; i < DES_KEY_LEN; i++) { // for all key bytes
        byte keyByte = key[i + offset];
        int count = 0;
        while (keyByte != 0) { // loop till no '1' bits left
            if ((keyByte & 0x01) != 0) {
                count++; // increment for every '1' bit
            }
            keyByte >>= 1; // shift right
        }
        if ((count & 1) == 0) { // not odd
            return false; // parity not adjusted
        }
    }
    return true; // all bytes were odd
}
```

Power profile of parity check (Gnu Crypto)



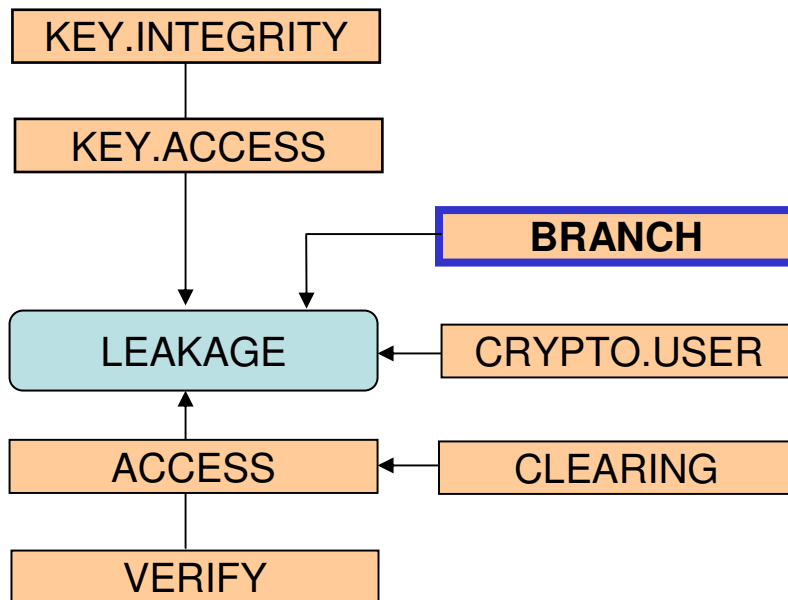
1 0 1 0 0 0 1 0 0 0 1 1 1 1 1 0 1 1 1 1 1 0 0 0 1

Make time-constant (OpenSSL)



Why would you now have to **randomize** the order?

Side channel protection patterns

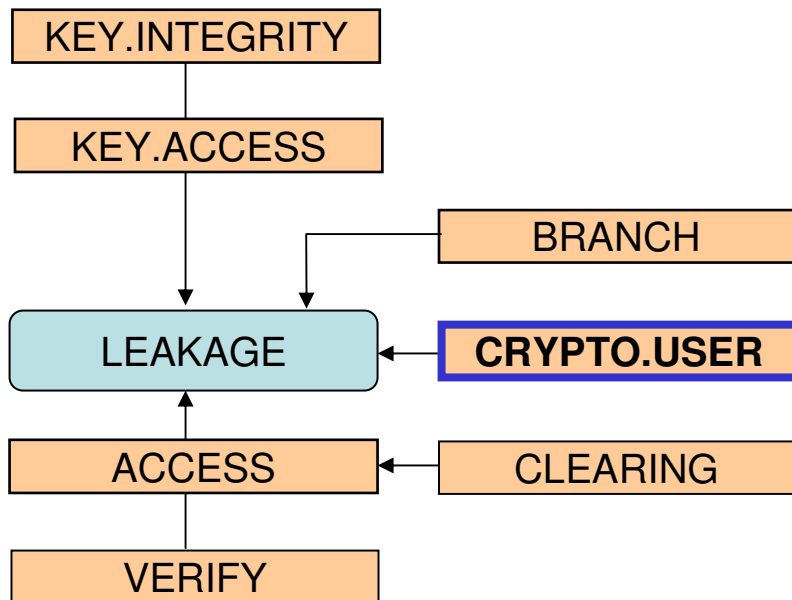


'if' and 'switch' statements leak information

→ **Avoid** "confidential" branches

→ Confidential decisions must be **time-constant** and in **identical** instruction sequence

Side channel protection patterns

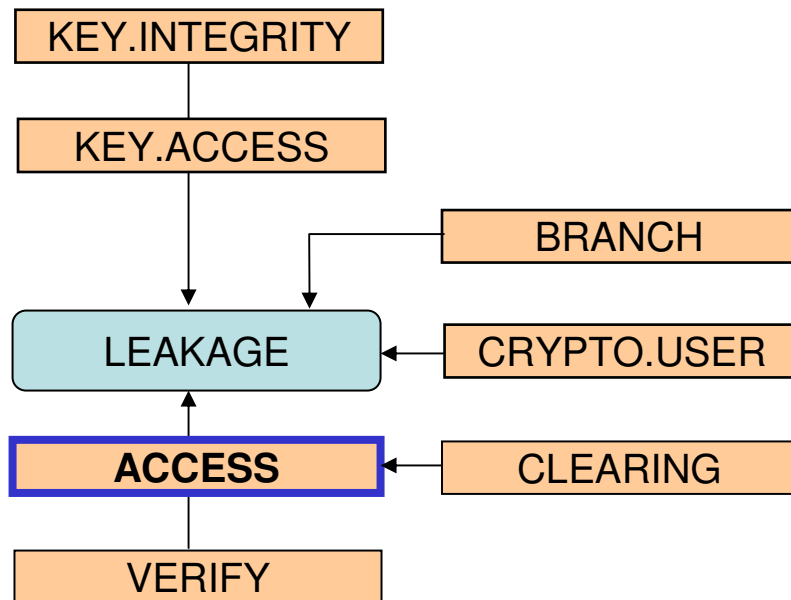


Implementing crypto is very tricky

→ **Use** - don't implement - crypto in an application

→ The **O/S and hardware** take care of this

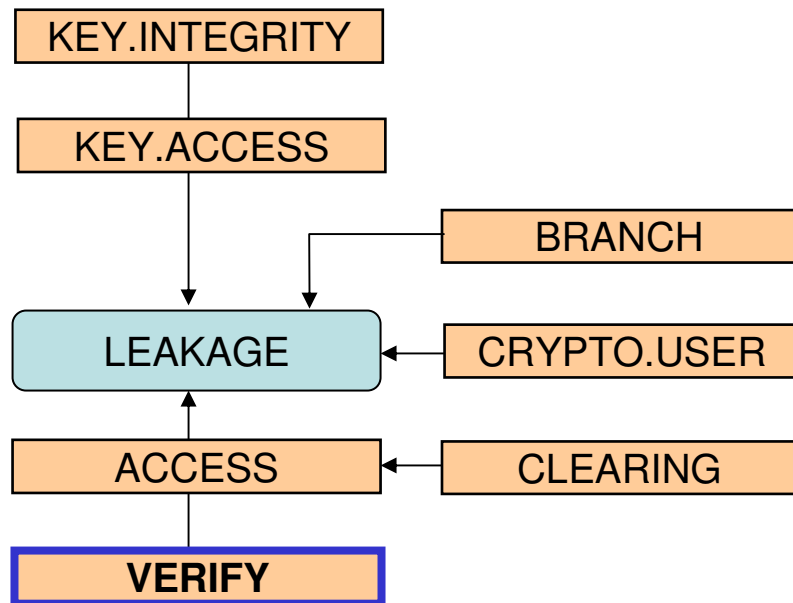
Side channel protection patterns



Accessing an array is prone to differential analysis

→ Use **dynamic offset** to traverse array

Side channel protection patterns

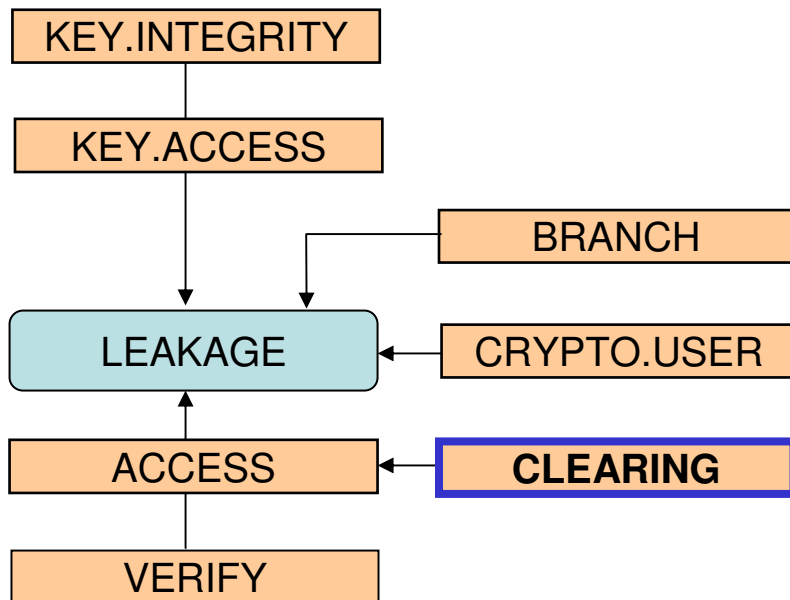


Verification of e.g. password leaks data on the stored value

- ➔ Compare **entire data value** and use LEAKAGE.ACCESS
- ➔ Better: compare **encrypted / hashed** value

`strcmp` and `memcmp` are sequential

Side channel protection patterns



Clearing an array leaks information

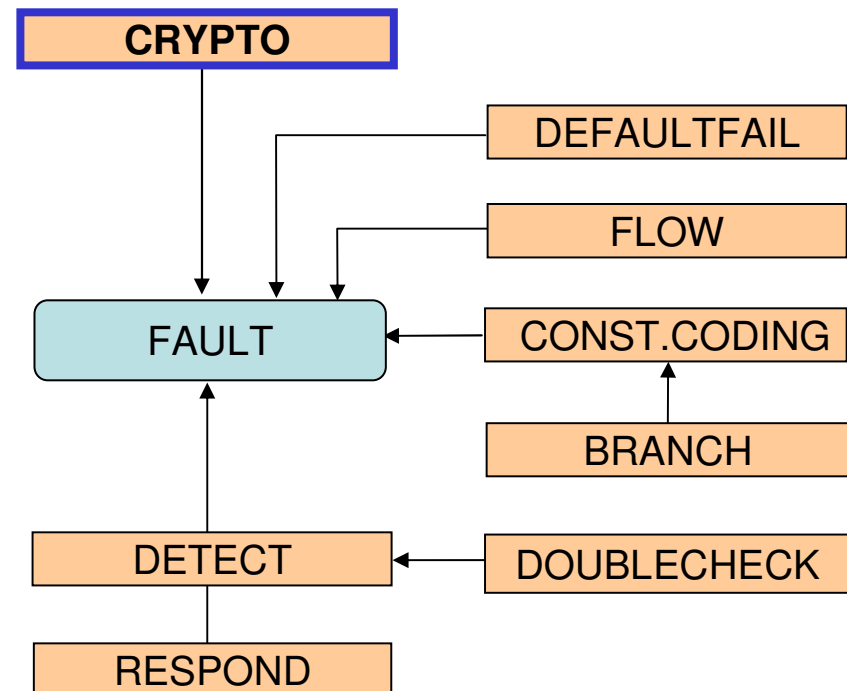
→ Overwrite with **random data**, and then clear it

Side channel protection patterns

Differential fault analysis reveals secret key

→ **Verify** ciphered data **before** transmitting it

→ Don't transmit if **inconsistent**

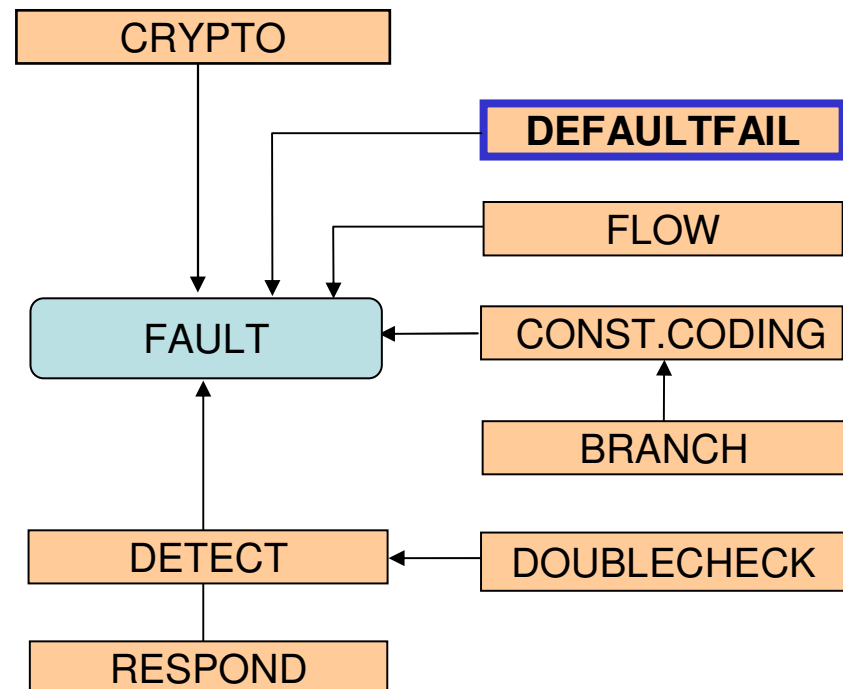


Side channel protection patterns

Assuming a parameter has a value eases fault injection

→ Also **verify** the last possible value when verifying conditions

→ By default, a comparison should lead to **a failure**

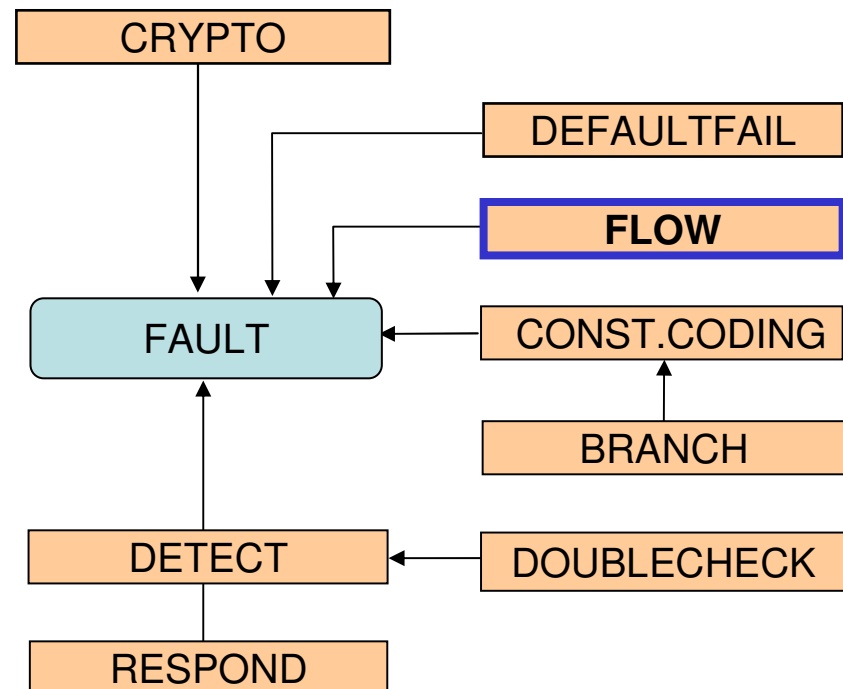


Side channel protection patterns

Program flow can be manipulated → code hi-jacking

→ Use a **shadow stack** and **program counter**

→ **Verify** these after branches



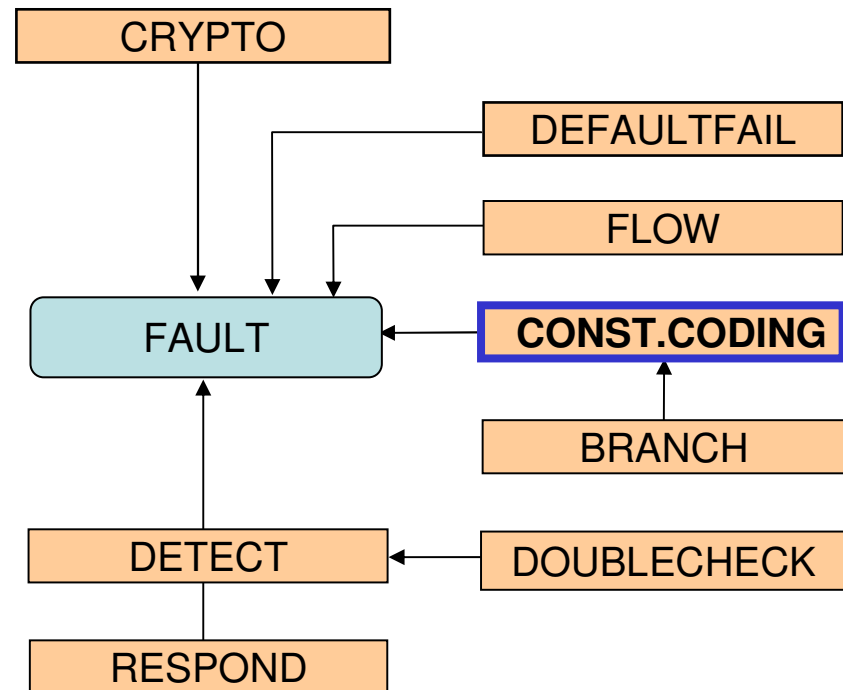
Side channel protection patterns

Simple data values are easy to manipulate

→ Use **non-trivial** values

→ Preferably with maximal **hamming distance**

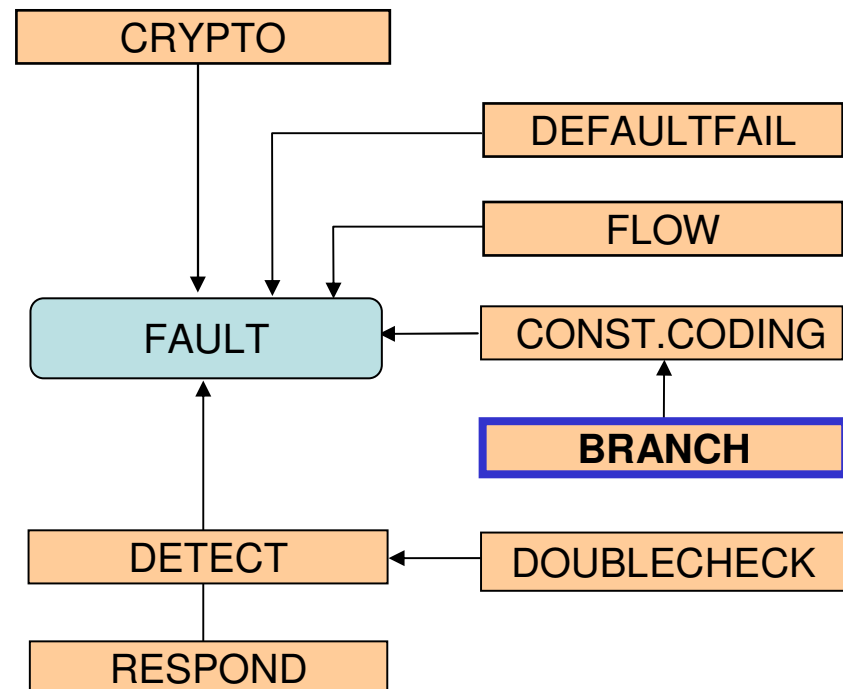
STATE_INIT = 0 ~~X~~
0x3C



Side channel protection patterns

A Boolean is easy to corrupt

→ Use **non-trivial** values for sensitive decisions

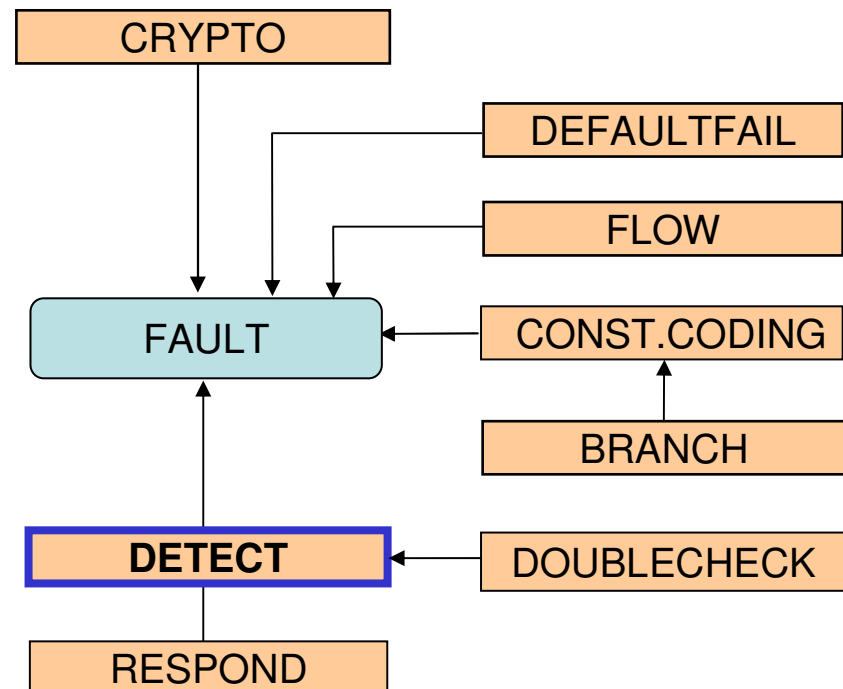


Side channel protection patterns

Detect if you get hit

→ Keep checksums and **verify each time** data is used

→ Use “**integrity-controlled objects**”

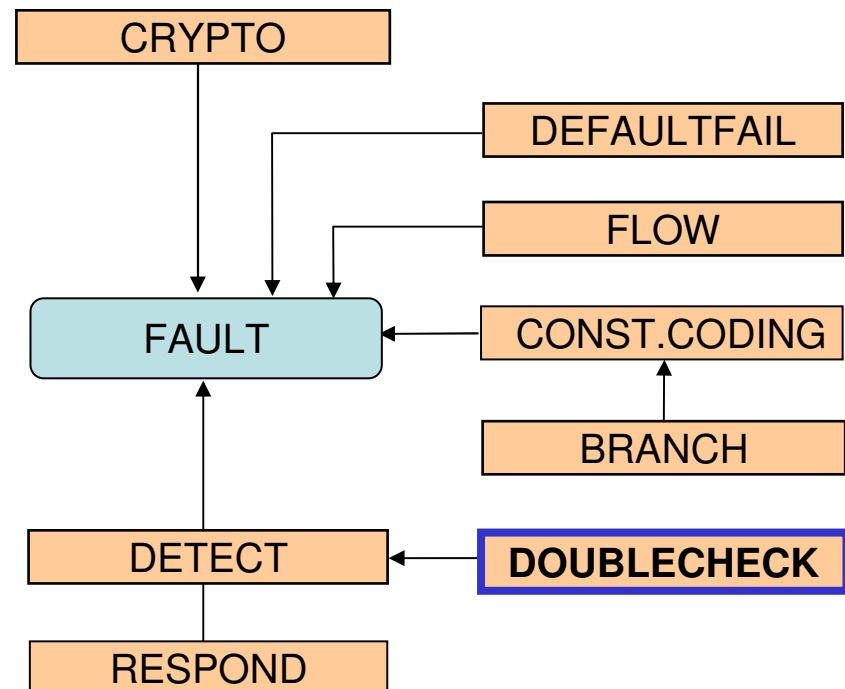


Side channel protection patterns

A wrong decision can be enforced

→ Verify data used in a decision **twice**

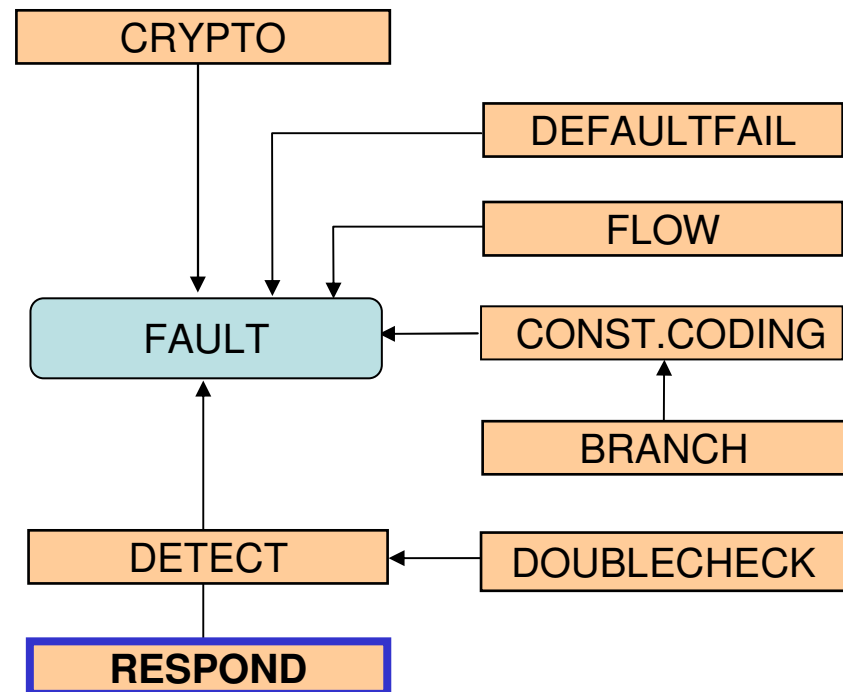
→ 2nd check should be **complementary to** 1st



Side channel protection patterns

Immunity is hard; respond after you detect it

- **Log incidents**
- **Disable** functionality
- **Wipe** secrets

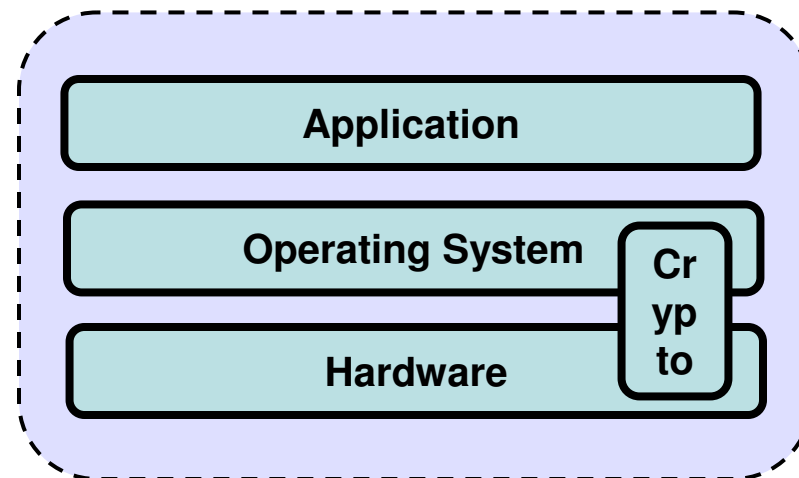


Outline

- Introduction
- What are side channel attacks?
- How to assess the risk?
- Secure programming patterns
- Conclusion
 - Discussion
 - Developer checklist
 - Further research
 - Concluding

Discussion

- Relationship with hardware & O/S protection?
 - Underlying level OK → Excellent, apply all patterns
 - Underlying level not OK → Prioritize what patterns to apply



Discussion (*cont.*)

- Implementing sufficient protection **is a challenge**
 - A **compiler** may undo countermeasures
 - Weakness identification requires **experience**
 - Protecting **sensitive decisions** is tedious
 - Incident response may conflict with **reliability** requirements
 - Trade off between secure programming and **maintainability**

Device developer checklist

- ✓ Does the device **need protection**?
- ✓ Understand the resistance of the **hardware & O/S**
- ✓ **Identify potential weaknesses** in design
- ✓ **Apply** the patterns
- ✓ Understand your **compiler**
- ✓ Code it
- ✓ **Test** the resistance of the device

Further research

- **Generic code solutions** for patterns
 - Secure comparison of arrays
 - Integrity-controlled objects
 - Shadow stack and program counter
 - Traps
- **Computer-aided** inspection & protection
 - Find weaknesses in code
 - Explore flow alternatives and privilege handling
 - Conversion to machine language (compiler)

Concluding

- **Be aware** that most existing software has little side channel resistance
- On our experience, **these patterns** improve the side channel resistance of devices

→ Read more on this in our paper (www.riscure.com)

References & resources

1. Design patterns: elements of reusable object-oriented software, ISBN 0-201-63361-2
2. Advances in smart card security, www.riscure.com/1_general/articles/ISB0707MW.pdf
3. Differential power analysis, www.cryptography.com/resources/whitepapers/DPA.pdf
4. Power analysis attacks: revealing the secrets of smart cards, ISBN 978-0-387-30857-9
5. OpenSSL, www.openssl.org
6. GNU Crypto, www.gnu.org/software/gnu-crypto/
7. Smart Card Alliance, <http://www.smartcardalliance.org/>
8. Secure application programming in the presence of side channel attacks, www.riscure.com

Thank you. More questions?

Marc Witteman
Chief Technology Officer
 witteman@riscure.com

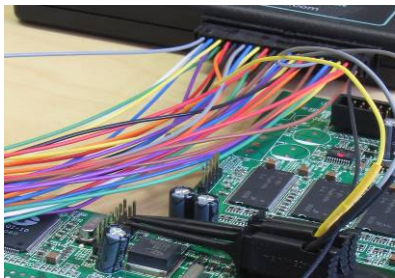
Harko Robroch
Managing Director
 robroch@riscure.com



Riscure B.V.
 Rotterdamseweg 183c
 2629 HD Delft
 The Netherlands

Phone: +31 (0)15 2682664
[Http://www.riscure.com](http://www.riscure.com)

Visit Riscure at
 booth 2838



**CHALLENGE
 YOUR SECURITY**

