# Software Countermeasures against Fault Attacks

Karine Heydemann



SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

# Fault injection attacks

**Fault injection means**

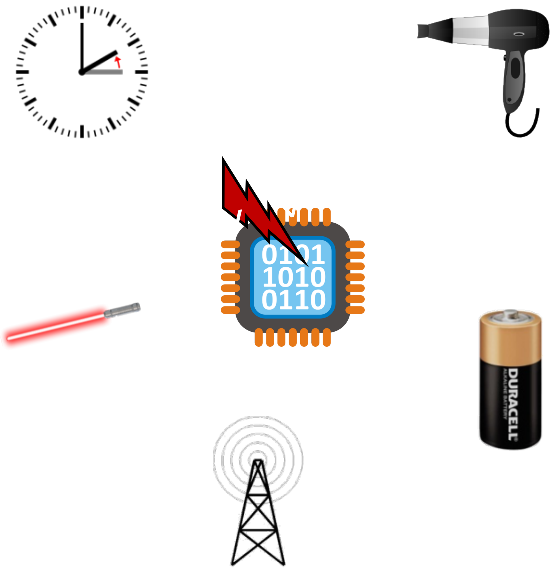
- Since 20 years via physical means: laser beam, electromagnetic pulse, clock or voltage glitch [El Bar et al., 2006]
- Recently via software means : row hammer, clock skrew

**Impact**

- Global : clock or voltage glitch [Yuce et al. 2017]
- Local : laser ou electromagnetic pulse [Dehbaoui et al. 2012]

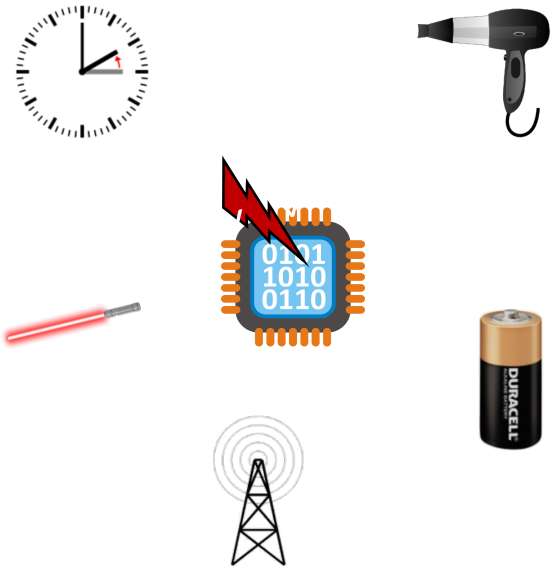
**Observed effects in storage elements**

- Bit(s) set or reset, bit flip(s)
- Transient ou permanent (stuck-at)



# Protections against fault injection attacks

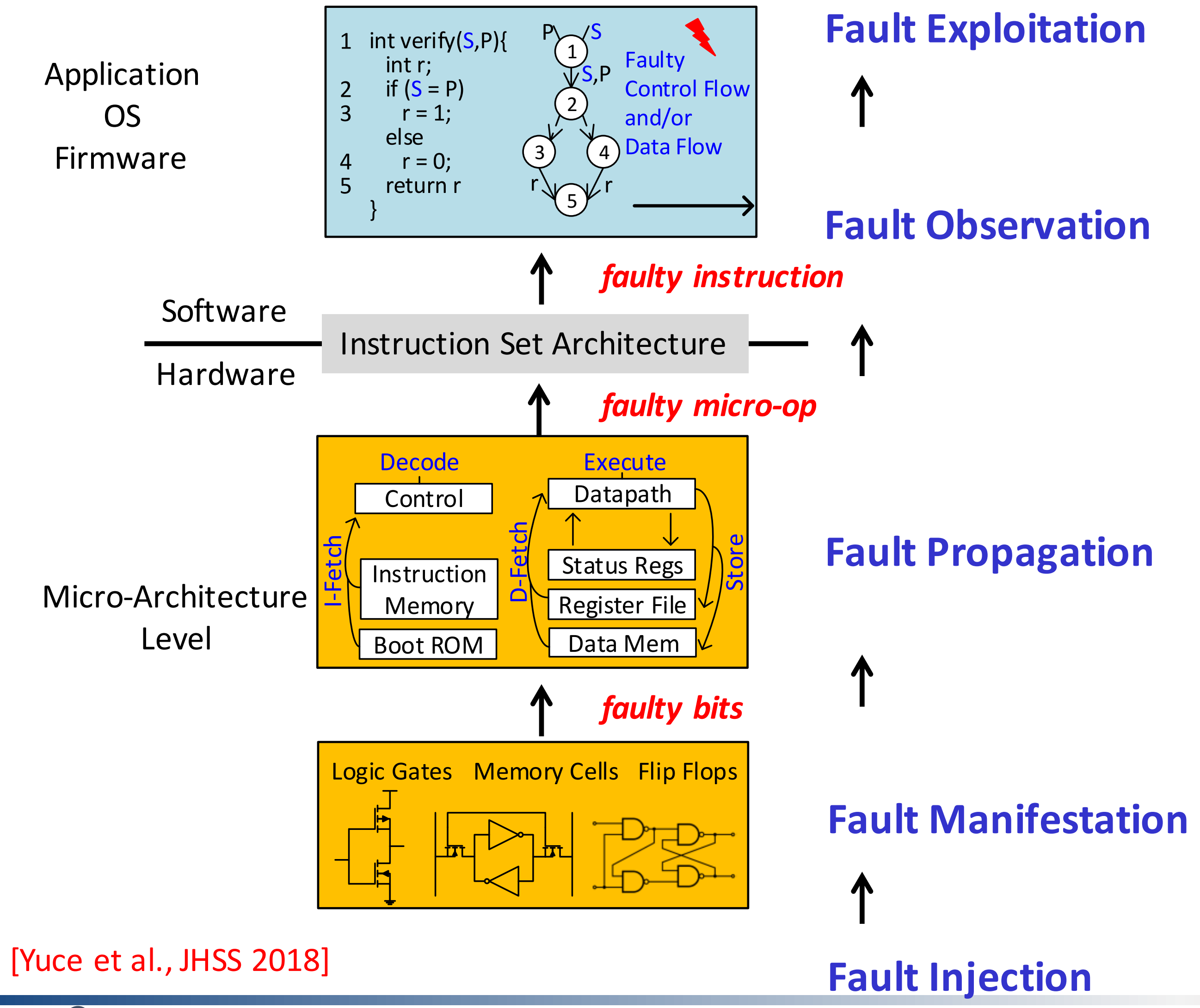
- **Hardware-based countermeasures** [El Bar et al., 2006]
  - Light sensor, glitch detectors [Zussa et al., 2014]
  - Redundancy [Karaklajic et al, 2013]
  - Error correcting codes (registers, memory)
- **Too expensive for small devices and no full guaranty**
- **Software-based countermeasures** [Verbauhede, 2011] [Rauzy et al., 2015]
  - Redundancy at function level
  - Algorithm-specific protection (e.g. RSA)
  - Ad-hoc protections designed by expert engineers
- In practice combination of both in secure elements



# SW protections against fault injection attacks

- **Manually added**
  - Tedious, error-prone
  - Highly expensive
  - Expertise needed
- **Need automation and capitalization**
  - Cost reduction, availability for non-experts
  - Adaptable to a specific product
  - Trade-off between security and performance
- **Need generic protections**
  - Not dedicated to a class of algorithms (crypto)
  - Against fault injection effects at software level…

# Fault attacks at software level

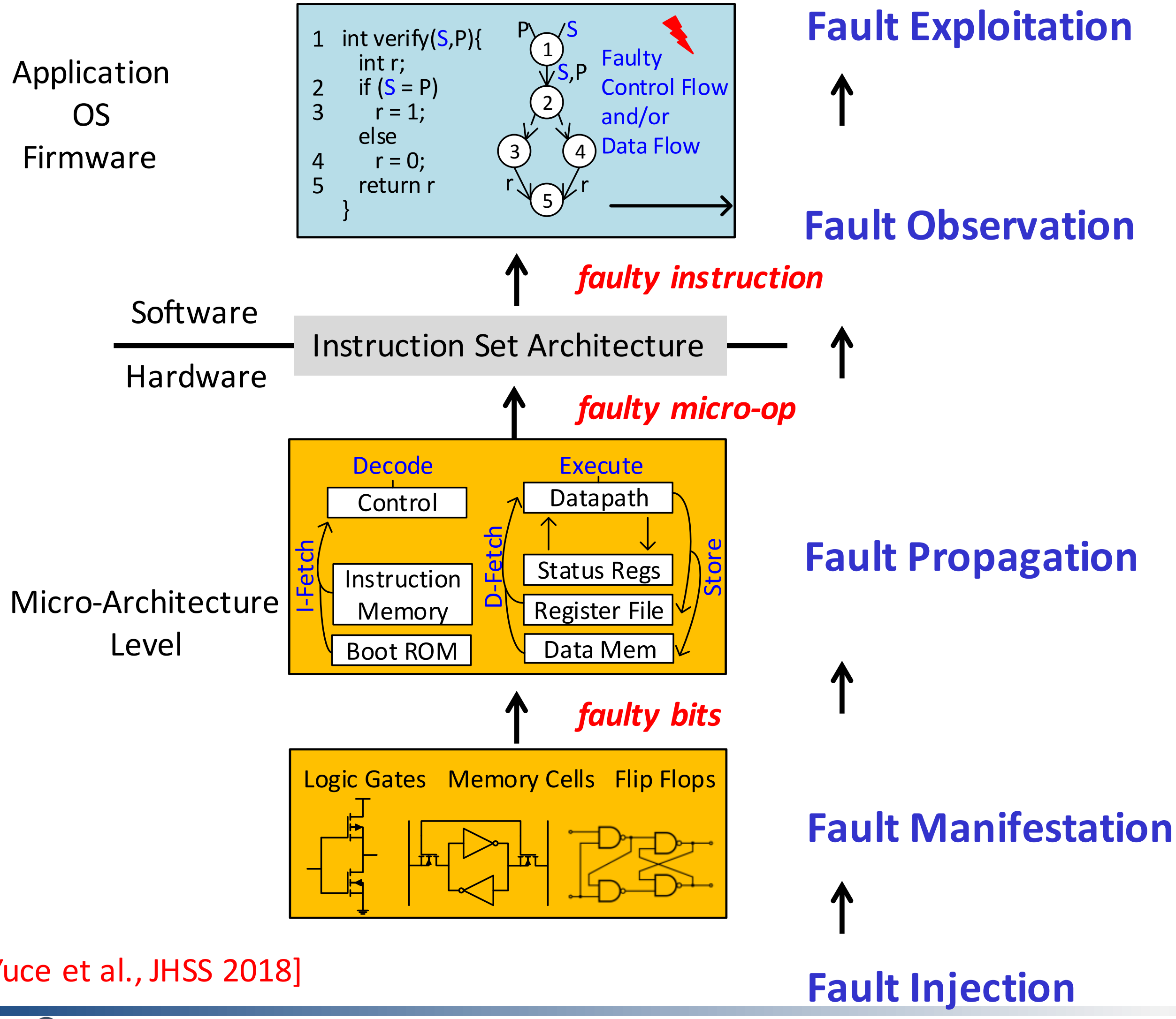



[Yuce et al., JHSS 2018]

## Fault observation depends on

- Fault injection means
- HW target
- Fault location / targeted part of the HW
- Running code

# Fault attacks at software level




[Yuce et al., JHSS 2018]

## Fault exploitation

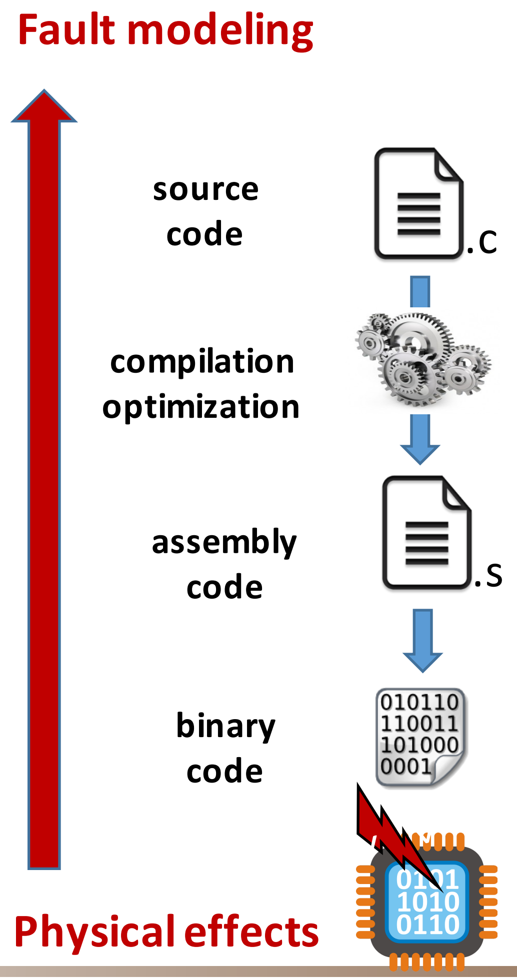
- **Macro view of fault attacks**
  - Cryptographic key retrieving [Dehbaoui et al., 2013] [Kumar et al., FDTC 2017]
  - Bypassing secure boot [Timmers et al., FDTC 2016]
  - Taking over a device [Timmers et al., FDTC 2017]
  - Privilege escalation [Vasselle et al. FDTC 2017]
- **Useful from an attacker point of view**

# Software fault characterization

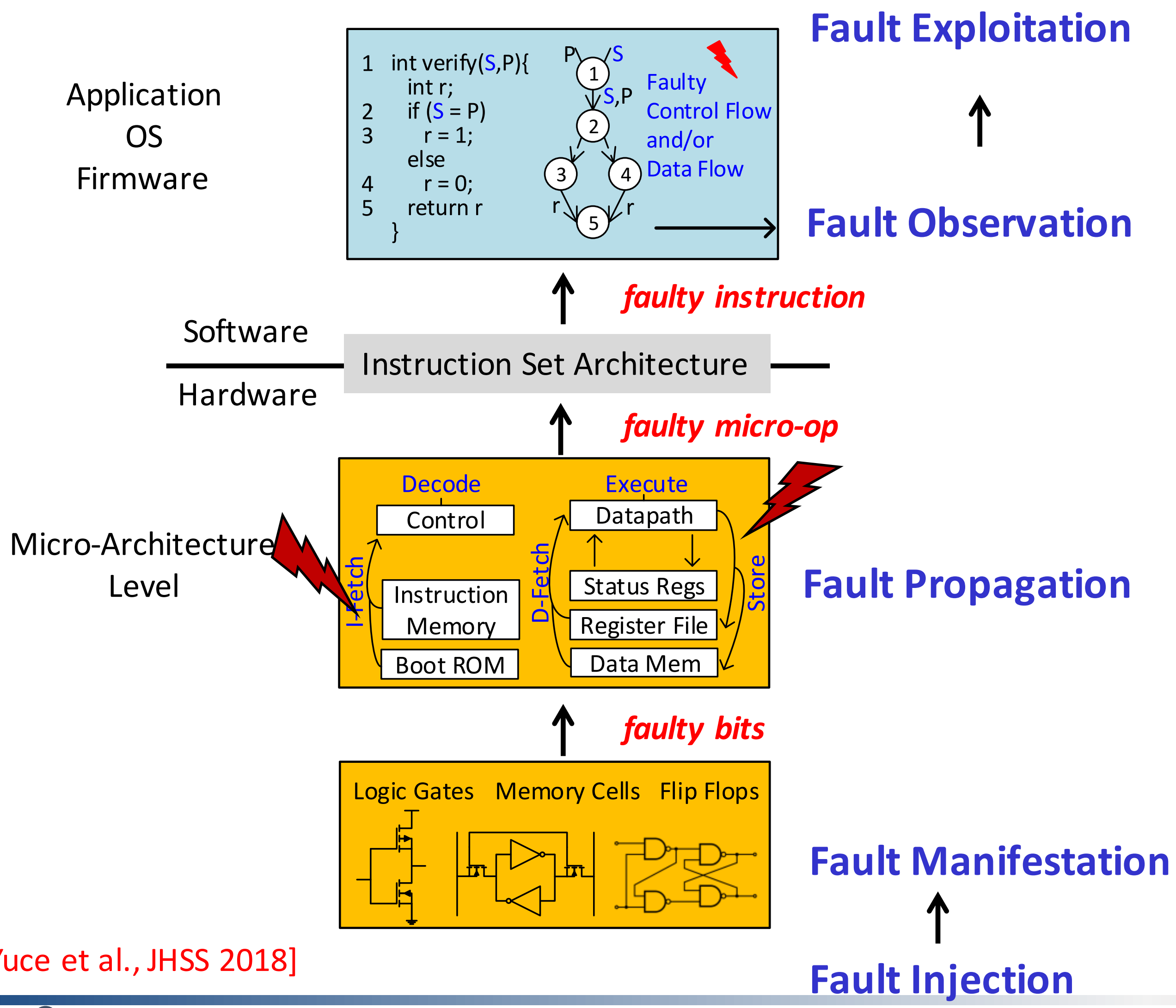
- Characterization of possible fault observations
- **Necessary to design software countermeasures**

**Fault model**

- **Simplified or abstracted representation** of a physical fault effects affecting an embedded software
- **At a given code level**: binary, assembly code, IR, source code



# Fault models at software level




[Yuce et al., JHSS 2018]

**Application of Attack Potential to Smartcards**
– Common Criteria, version 2.9. May 2013

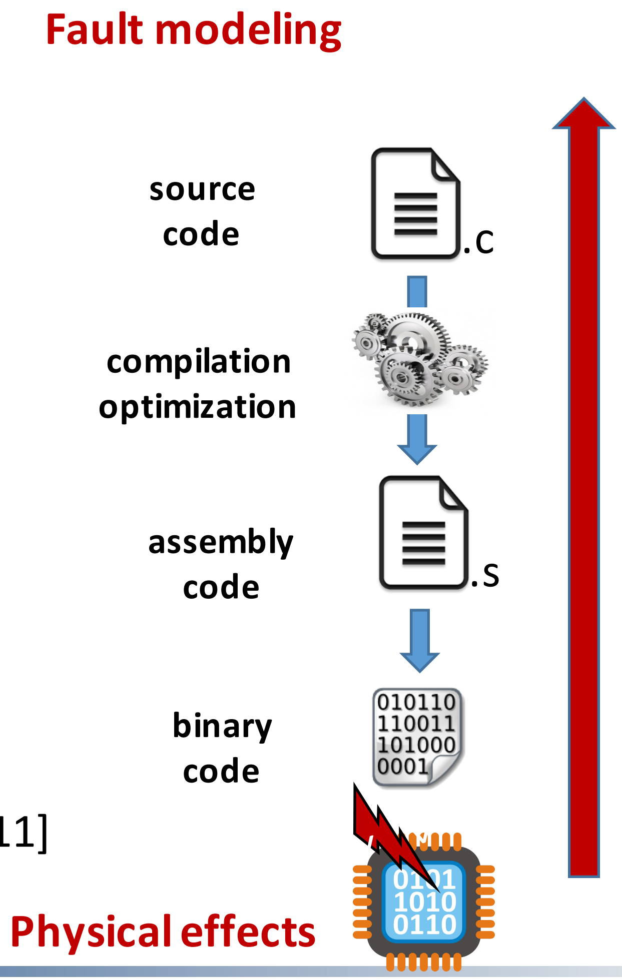
## Common instruction-level fault models

- Instruction skip
- Instruction replacement
- Test inversion
- Jump insertion
- Computation or register corruption
- Data memory corruption

# Fault models at software level

[Berthomé et al, 2010]
[Berthomé et al, 2013]

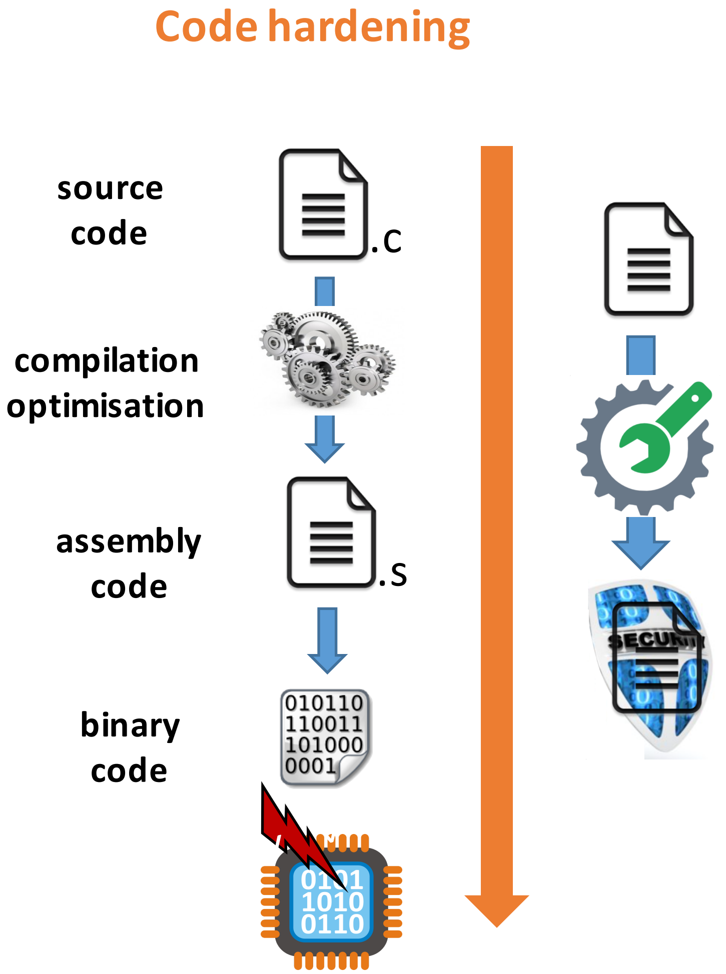
[Kelly et al., 2017]
[Yuce et al., 2017]
[Timmers et al., 2016]
[Dureuil et al., 2015]
[Rivière et al., 2015]
[Moro et al, 2013]
[Balash et al., 2011]
[Verbauwhede et al., 2011]
[El Bar et al., 2006]




- At source code level
  - Control-flow disruption
  - Variable corruption
  - Combination

- At assembly level
  - Instruction(s) skip
  - Instruction(s) replacement
  - Corruption of loaded data
  - Register(s) corruption(s)

# Software protection against fault attacks




- **At which code level?**
  - **Source**
    - Code review, portability, independent from tools
    - Fault models, compilation optimization
  - **Compilation**
    - Adaptability and/or control over code optimization
    - No existing compilation tool
  - **Assembly**
    - Realistic fault models, low level information available
    - Target specific, potential lack of source code information
  - **Binary**
    - Attacked code, global view, availability of library codes
    - Even more lack of semantic information

➜ Multiple needs

# Outline

- **Principle of software countermeasures**
  - Data integrity
  - Code integrity
  - Control-flow integrity
- **Compiler-assisted code hardening**
  - Protection against instruction skip
  - Loop hardening scheme

# Countermeasures for data integrity

## Fault model

- Data corruption: register corruption, load corruption

## Redundancy-based protections

- Duplication of instructions involved in a computation
- Comparison of results of duplicated computations
- Detection of
  - Register corruption (r1 or r2)
  - Load corruption
- Need available registers

```
add   r1, r0, #1
```

duplicate and compare →

```
add   r1, r0, #1
add   r2, r0, #1
cmp   r2, r1
bne   fault_detection
```

```
ldr   r1,  [r0]
```

duplicate and compare →

```
ldr   r1, [r0]
ldr   r2, [r0]
cmp   r2, r1
b.ne  fault_detection
```

A. Barenghi et al. *Countermeasures against fault attacks on software implemented AES.* 5th Workshop on Embedded Systems Security (WESS'10)

# Countermeasures for data integrity

**Fault model**

- Data or data-related computation corruption: register corruption, load and memory corruption

**Redundancy-based protections**

- Data duplication in addition to instruction duplication
- Detection of
  - Memory corruption
  - Load corruption
  - Register corruption
- High overhead: performance and memory footprint

```
ldr   r1, [r0]
```

Duplicate data, instruction, and compare

```
ldr    r1, [r0]
ldr    r2, [r0+offset]
cmp    r2, r1
b.ne   fault_detection
```

Reis et al. *SWIFT: Software Implemented Fault Tolerance.* International Symposium on Code Generation and Optimization. 2005

# Countermeasures for code integrity

## Fault model

- Instruction corruption

## Redundancy-based protections

- Instruction duplication with detection
- Detection of
  - One instruction skip
  - Some instruction replacements

```
ldr   r1,  [r0]
```

duplicate and compare →

```
ldr     r1, [r0]
ldr     r2, [r0]
cmp     r2, r1
b.ne    fault_detection
```

A. Barenghi et al. *Countermeasures against fault attacks on software implemented AES*. 5th Workshop on Embedded Systems Security (WESS'10)

# Countermeasures for code integrity

## Fault model

- Instruction skip

## Redundancy-based protections

- Instruction duplication without detection
  - Tolerance to one instruction skip
  - Only for idempotent instructions
  - Transformation of non-idempotent instructions

```
add   r1, r0, #1
```

duplicate

```
add   r1, r0, #1
add   r1, r0, #1
```

```
add r1, r1, r2
```

Transform into idempotent instructions

```
add   rX, r1, r2
mov   r1, rX
```

duplicate

```
add   rX, r1, r2
add   rX, r1, r2
mov   r1, rX
mov   r1, rX
```

Moro et al. *Formal verification of a software countermeasure against instruction skip attacks.* Journal of Cryptographic Engineering 2014.

# Countermeasures for code integrity

## Fault model

- Instruction skip

## Redunda

- Instr
  - T
  - C
  - T

No software-only protection for full code integrity
(i.e. against all kinds of instruction replacement or disruption)

```
add r1, r1, r2
```

idempotent instructions

```
mov r1, rX
```

duplicate

```
mov r1, rX
mov r1, rX
```

Moro et al. *Formal verification of a software countermeasure against instruction skip attacks.* Journal of Cryptographic Engineering 2014.

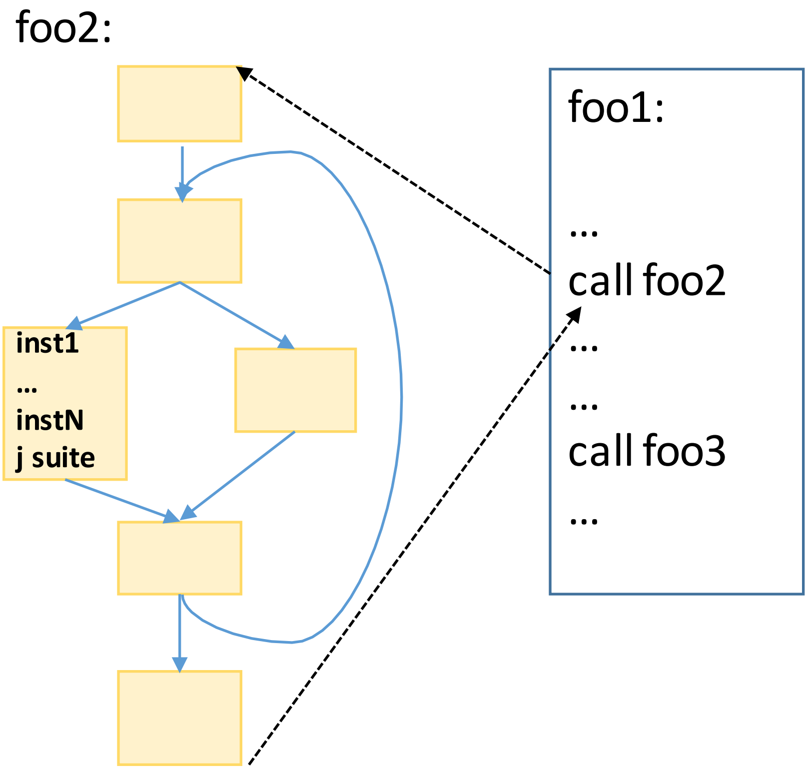
# Control flow integrity

**Fault model**

- Jump insertion

**Different levels of control-flow integrity**

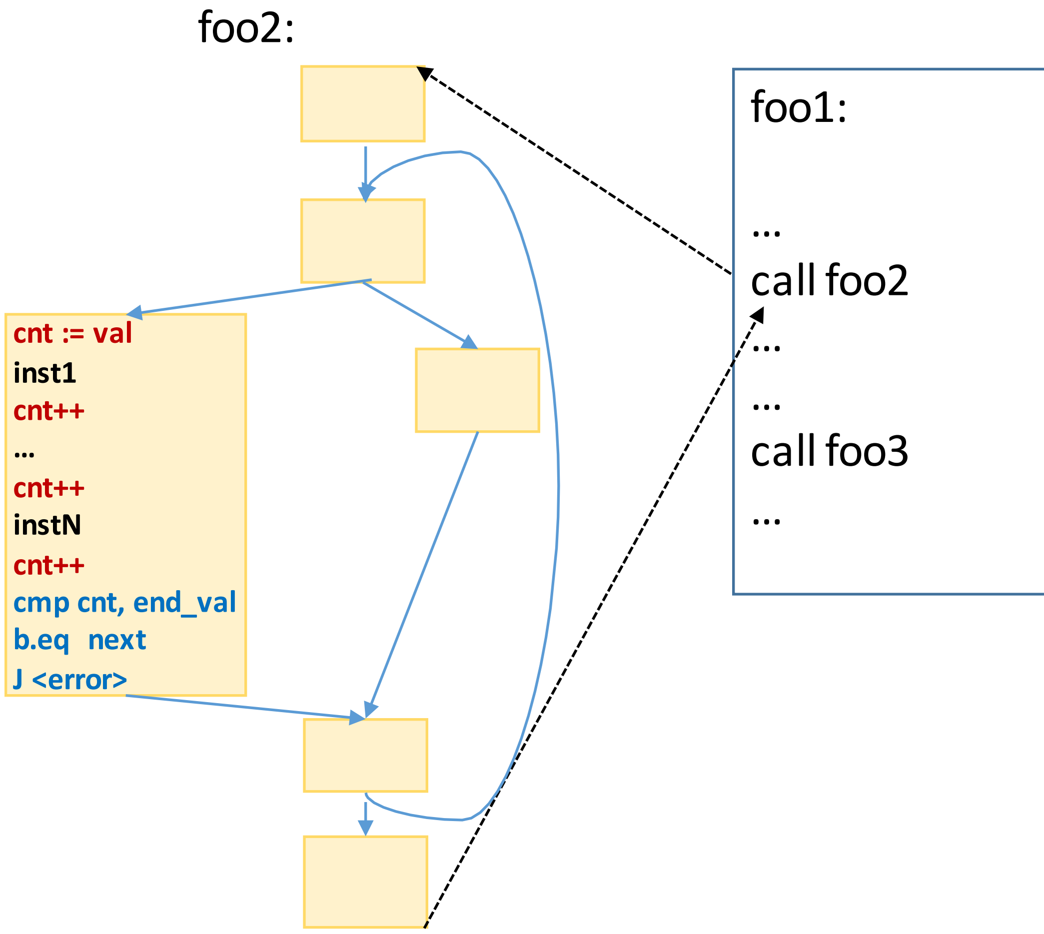
- Intra basic block
  integrity of straight-line code
- Intra procedural
  integrity of control flow transfers inside a function (control flow graph)
- Inter procedural
  integrity of function calls and returns



# Intra basic block control flow integrity

**Counter-based protections** [Akkar et al., 2003]

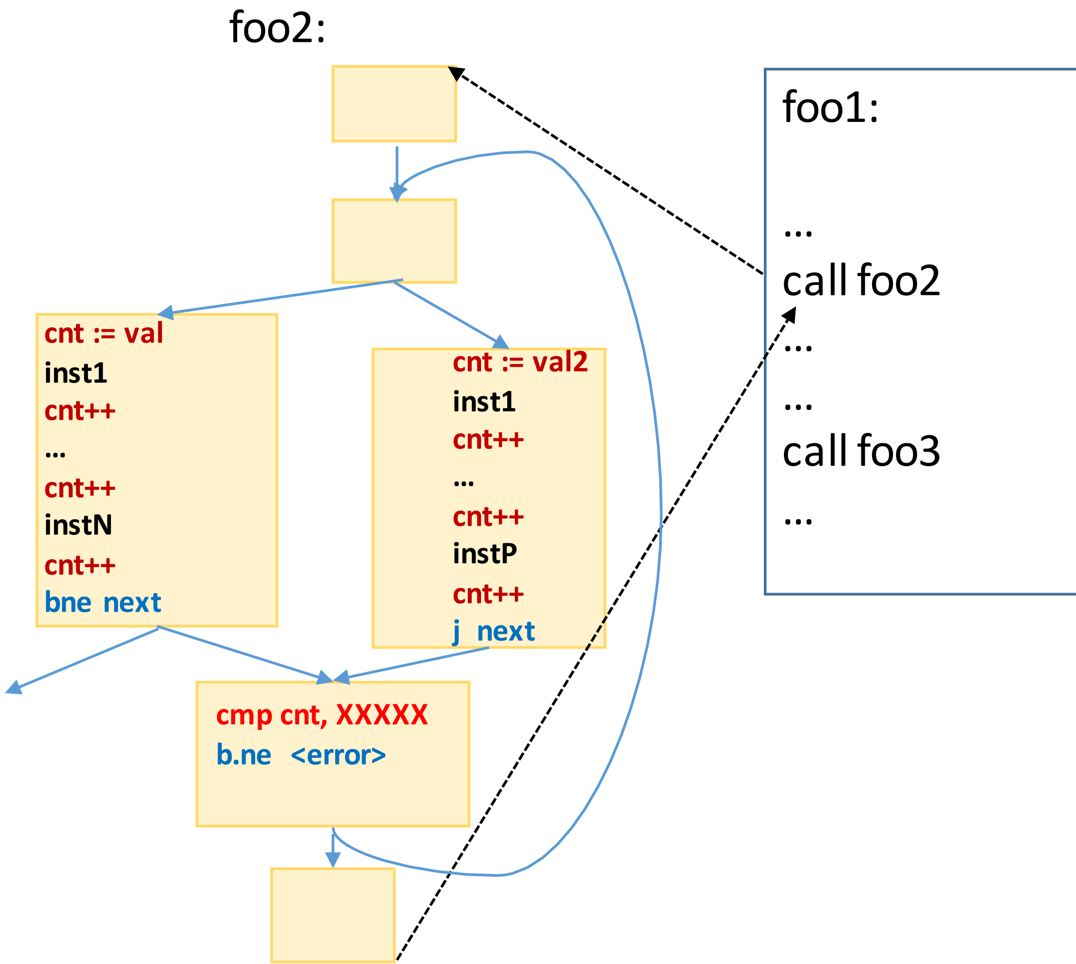
- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps



# Control flow integrity

**Counter-based protections**

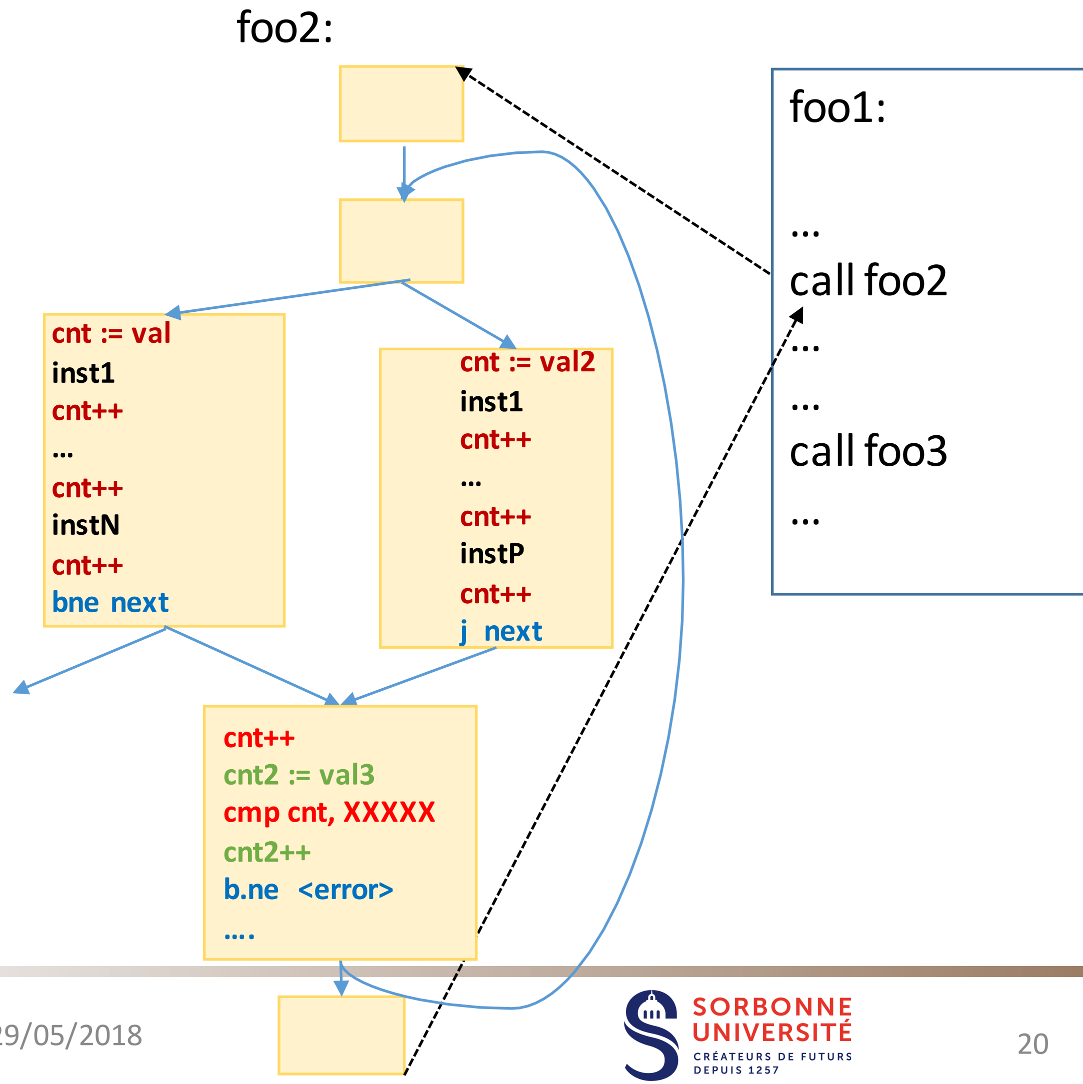
- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks
    - Need for extra code



# Control flow integrity

**Counter-based protections** [Lalande et al., 2014]

- Dedicated counters incremented between instructions
- Check of their values at some specific points
  - At the end of each BB: only detects some intra BB jumps
  - At the beginning of target blocks
    - Need for extra code
    - Overlap of counters initialization and check



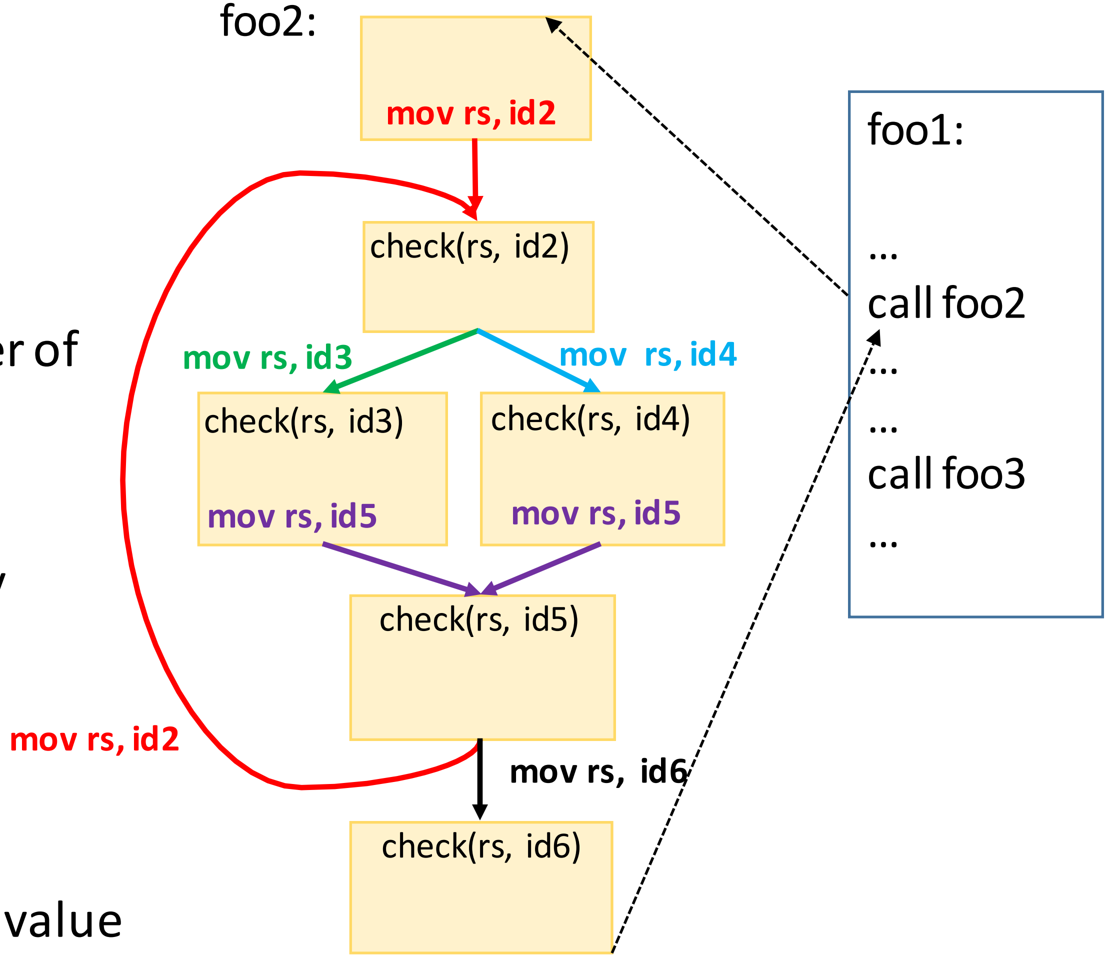
# Countermeasures for control flow integrity

**Signature-based protections** [Oh et al. 2002] [Goloubeva et al., 2005]

- Unique identifier / signature assigned to every basic block (and function)
- Use to check every single control flow transfer
- Global signature computation limits the number of checks
- Ensure the CFG integrity
- Need branch condition integrity / data integrity

**Combination** [SIED, 2003]

- Step counters inside basic blocks
- Signature for control flow transfers
- Signature computed with the branch condition value

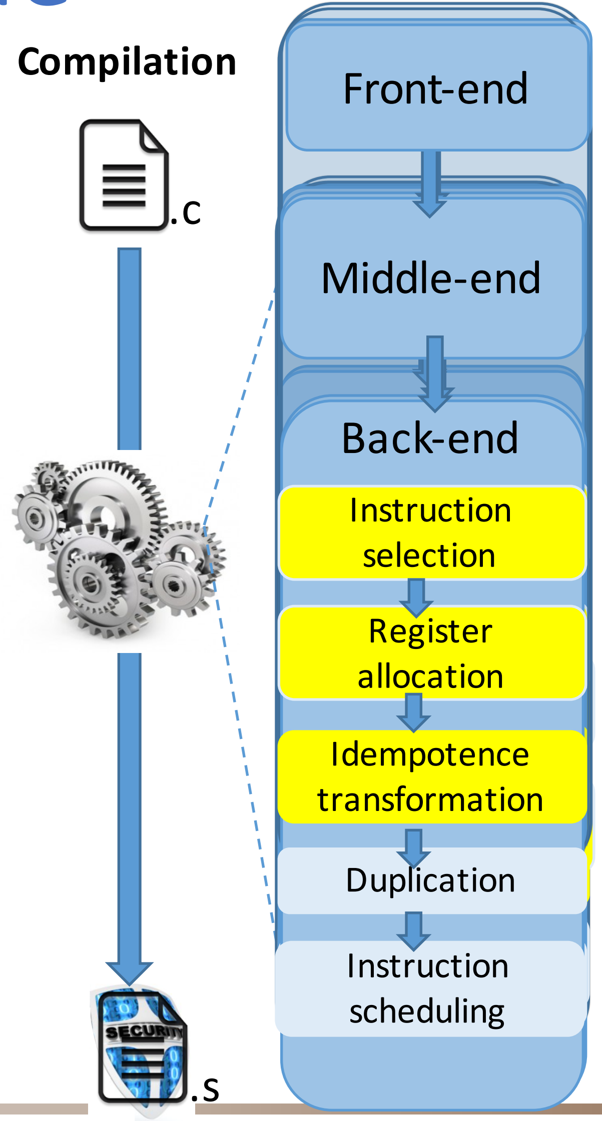


# Outline

- **Principle of software countermeasures**
  - Data integrity
  - Code integrity
  - Control-flow integrity
- **Compiler-assisted code hardening**
  - Protection against instruction skip
  - Loop hardening scheme

# Protection at compilation-time

- **Protection scheme against instruction skip** [Moro et al. 2014]
- Main principle: **duplication of idempotent instructions**
- **Take advantage of compilation flow to**
  - Force the generation of idempotent instructions
    - Avoidance of some instructions at the selection step
    - Modification of the register allocation
    - Additional transformation for remaining non-idempotent instructions (e.g. push and pop instruction that use and modify the stack pointer)
  - Add an instruction duplication pass
  - Let the scheduler optimize the resulting protected code
- Results in automatically protected code with better code size and performance




T. Barry et al. *Compilation of a Countermeasure Against Instruction-Skip Fault Attacks.* CS2 2016.

# Compile-time loop hardening

## Motivation

- Several attacks exploit a **corruption of loop iteration count** (early or deferred exit)
  - Buffer overflows [Nashimoto et al. 2017]
  - Cryptanalysis by round reduction [Dehbaoui et al. 2013, Espitau et al. 2016]
  - Authentication process [Dureuil et al., FISSC, 2016]
- Full duplication schemes are too expensive
- How to automatically protect a loop ?

```
void aes_addRoundKey_cpy(
     uint8_t *buf, uint8_t *
     key, uint8_t *cpk)
{
  register uint8_t i = 16 ;

  while (i--)
  {
    buf[i] ^= key[i] ;
    cpk[i] = key[i] ;
    cpk[16+i] = key[16+i] ;
  }

}
```

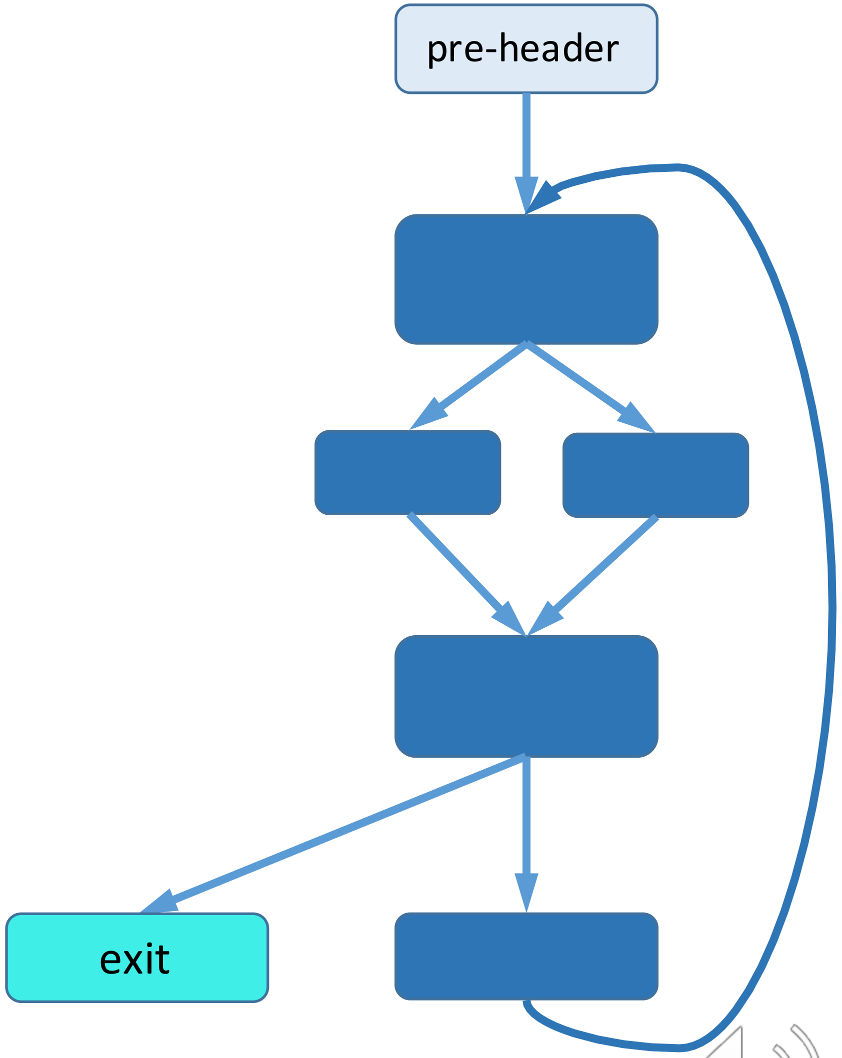
# Loop hardening scheme

**Fault model**

- One instruction skip
- One general purpose register corruption
- During loop execution

**Security objective**

- The loop performs the right iteration count
- The loop exits from the right exit
- Otherwise an attack is detected



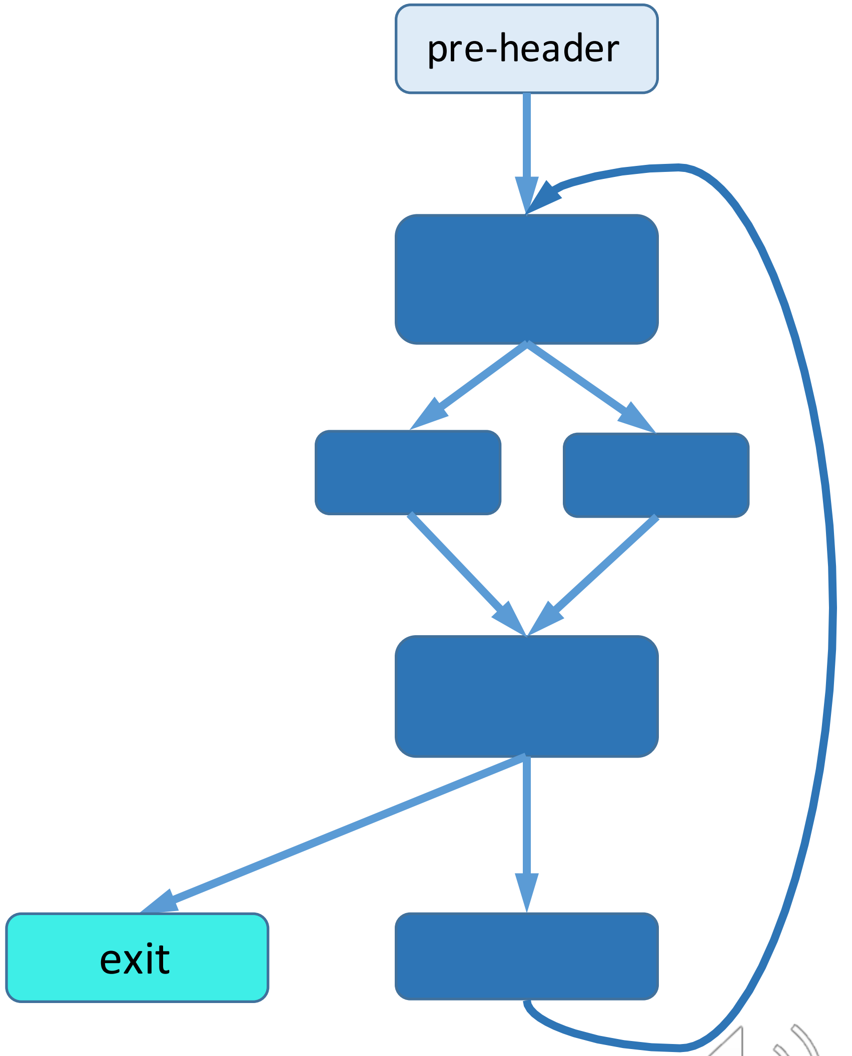
# Loop hardening scheme

## Protection principle

- For each loop exit, check its outcome

## Realisation

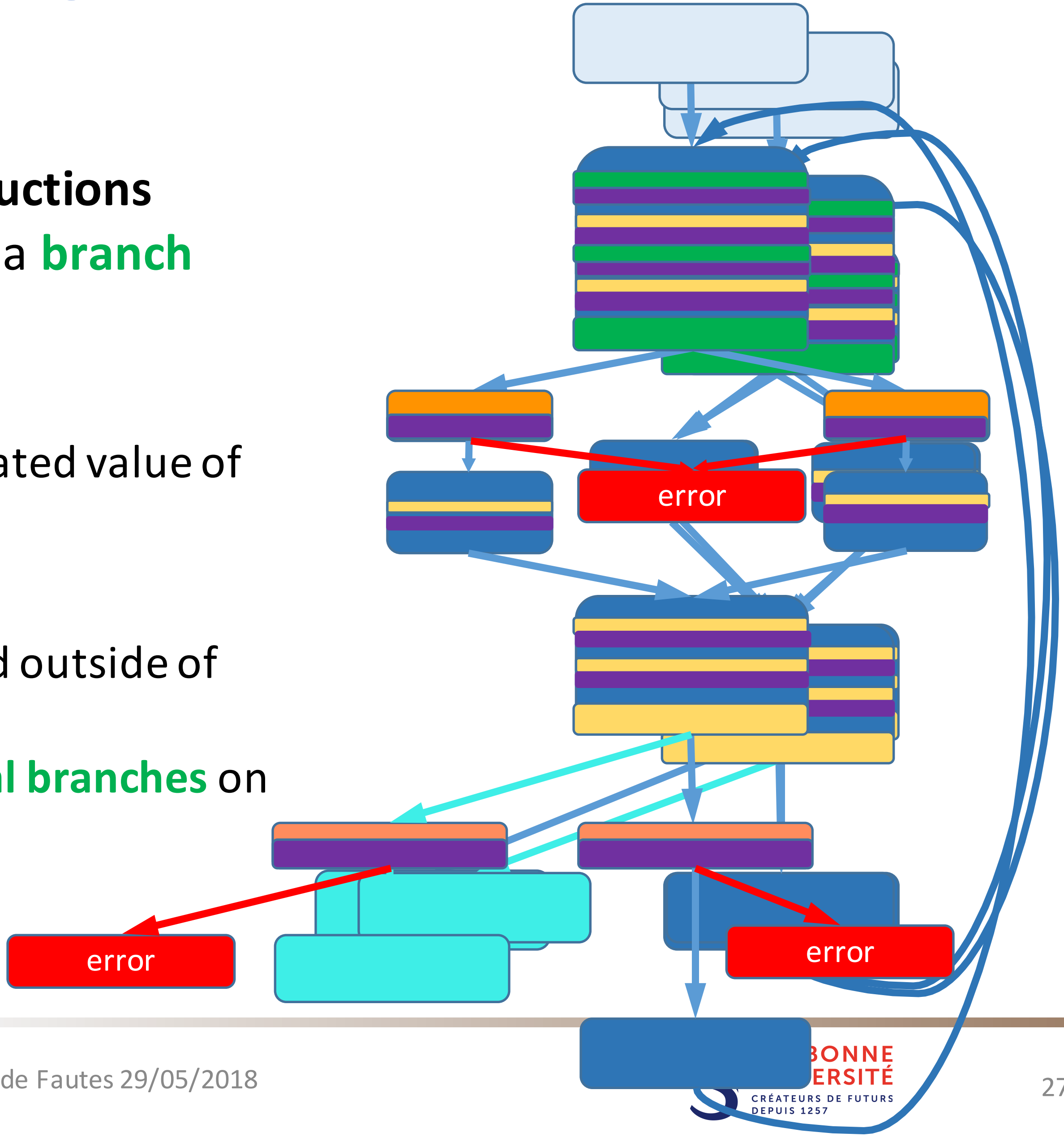
- **Duplication** of all the instructions involved in the computation of an exit condition
- Addition of **verification basic blocks** on all the paths following from an exiting block
- Protection of the **internal control flow** that may impact an exit condition



# Loop hardening scheme

**For each exit of a loop**

- Determination by a backward analysis of the **instructions involved** in **an exit condition** or in an condition of a **branch that may influence an exit condition**
- **Instruction duplication**
  - Creation of a **second data flow** leading to a duplicated value of the condition, independant from the original one
- **Addition of verification blocks**
  - Checks of **the duplicated exit condition** inside and outside of the loop to verify the exiting branch
  - Checks of **the duplicated conditions of the internal branches** on all possible following paths
  - **Call to a fault detection handler**



# Loop hardening pass and a compilation flow

**Automation and insertion in a compilation flow**

- Implemented in a compiler (LLVM 3.9+) at the intermediate level
- Insertion after optimization passes that may alter the protection

**Experimental results**

- 99% of harmful simulated fault are detected
- Low overhead in performance and code size

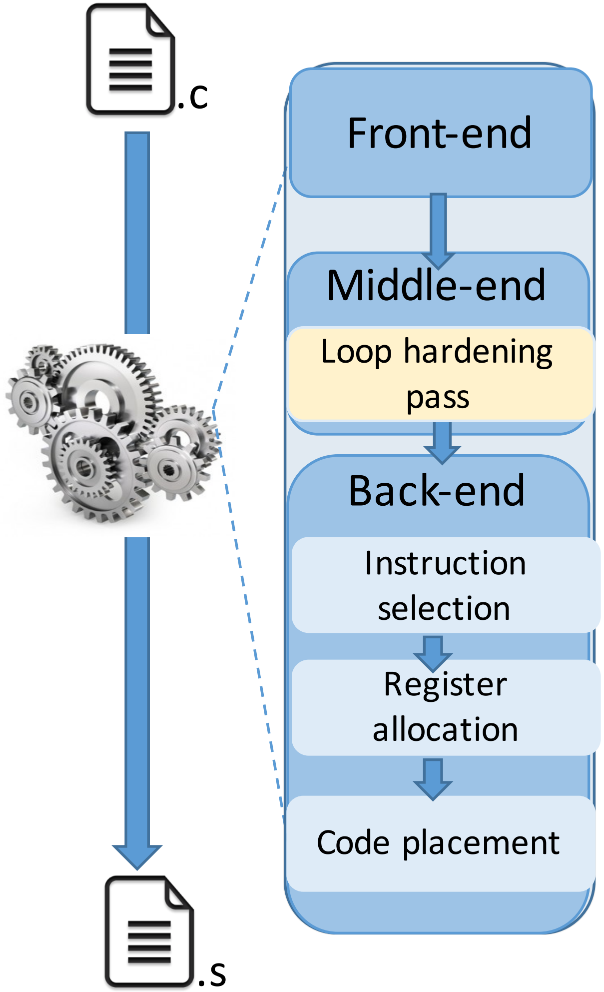
**Harmful post-securing transformations and optimizations**

- All kind of redundancy elimination
- Instruction selection, register allocation, code placement optimization

➜ Compiler is not compliant with protection / security properties

➜ **Need to analyze the generated code**

➜ **Need to deactivate, adapt, or add some passes to enforce the security property**




J. Proy at al. *Compiler-Assisted Loop Hardening Against Fault Attacks.* ACM Transaction on Architecture and Code Optimization. December 2017

# Summary and conclusion

- **Various types of protection**
  - Large set of fault models / attacker capabilities
- **Need of automatic code hardening** and against a large set of (faults) attacks
  - Compiler-assisted code hardening
  - Framework enabling the analysis and the **preservation of security properties**
    - In the compilation flow
    - For a post-compilation robustness analysis
- **Combination of protections**
  - Interaction between protections? Stacking or smarter combination?

# References

[AAPS CC 2013] Common Criteria. (2013). Application of Attack Potential to Smartcards, Version 2.9.

[Balash et al. 2011] Balasch, J., Gierlichs, B., and Verbauwhede, I. (2011). An In-depth and Black-box Characterization of the Effects of Clock Glitches on 8-bit MCUs. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2011), pages 105–114. IEEE.

[Berthomé et al, 2010] Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., and Lalande, J.-F. (2010). Attack model for verification of interval security properties for smart card c codes. 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'10)

[Berthomé et al, 2012] Berthomé, P., Heydemann, K., Kauffmann-Tourkestansky, X., and Lalande, J.-F. (2012). High level model of control flow attacks for smart card functional security. In Proceedings of the 7th International Conference on Availability, Reliability and Security (ARES'12),.

[Dehbaoui et al. 2013] Amine Dehbaoui, A., Mirbaha, A-P., Moro, N., Dutertre, J-M., Tria, A.: Electromagnetic Glitch on the AES Round Counter. COSADE 2013: 17-31

[Dureuil et al. 2015] Dureuil, L., Potet, M., de Choudens, P., Dumas, C., and Clédière, J. (2015). From code review to fault injection attacks: Filling the gap using fault model inference. In Proceedings of the 14th International Conference Smart Card Research and Advanced Applications (CARDIS), volume 9514 of LNCS, Springer.

[Dureuil et al, FISSC, 2016] Dureuil, L., Petiot, G., Potet, M.-L., Crohen, A., and Choudens, P. D. (2016). FISSC: a Fault Injection and Simulation Secure Collection. In Proceedings of International Conference on Computer Safety, Reliability and Security, volume 9922 of (SAFECOMP).

# References

[El Bar et al., 2006] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., and Whelan, C. (2006). The sorcerer's apprentice guide to fault attacks. Proceedings of the IEEE, 94(2):370–382.

[Espitau et al. 2016] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, Mehdi Tibouchi: Loop-Abort Faults on Lattice-Based Fiat-Shamir and Hash-and-Sign Signatures. SAC 2016: 140-158

[Kelly et al., 2017] M.S.Kelly, K.Mayes, J.F.Walker. Characterising a CPU fault attack model via run-time data analysis. In 2017 IEEE International Symposium on Hard- 1268 ware Oriented Security and Trust (HOST). 79–84.

[Kumar et al., 2017] S. V. Dilip Kumar; Sikhar Patranabis; Jakub Breier; Debdeep Mukhopadhyay; Shivam Bhasin; Anupam Chattopadhyay; Anubhab Baksi. A Practical Fault Attack on ARX-Like Ciphers with a Case Study on ChaCha20. 2017 Workshop Fault Diagnosis and Tolerance in Cryptography Workshop (FDTC), 2017. IEEE.

[Karaklajic et al, 2013] Karaklajic, D., Schmidt, J-M., and Verbauwhede, I. Hardware Designer's Guide to Fault Attacks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2013.

[Nashimoto et al. 2017] Nashimoto, S. Homma, N., Hayashi, Y., Takahashi, J., Fuji, H., Taoki, T.: Buffer overflow attack with multiple fault injection and a proven countermeasure. J. Cryptographic Engineering 7(1): 35-46 (2017)

[Rivière et al. 2015] Rivière, L., Najm, Z., Rauzy, P., Danger, J.-L., Bringer, J., and Sauvage, L. (2015). High precision fault injections on the instruction cache of armv7-m architectures. IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2015)

[Timmers et al., 2016] Timmers, N., Spruyt, A., and Witteman, M. (2016). Controlling PC on ARM Using Fault Injection. 2016 Fault Diagnosis and Tolerance in Cryptography Workshop (FDTC 2016), pages 25–35, Santa Barbara, CA, USA. IEEE.

# References

[Timmers et al., 2017] Timmers, N., Mune, C. Escalating Privileges in Linux Using Voltage Fault Injection. Workshop Fault Diagnosis and Tolerance in Cryptography Workshop (FDTC), 2017. IEEE.

[Vasselle et al., 2017] Vasselle, A., Thiebeauld, H., Maouhoub, Q., Morisset, A. and Ermeneux, S. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot. 2017 Workshop Fault Diagnosis and Tolerance in Cryptography Workshop (FDTC), 2017. IEEE.

[Verbauwhede et al., 2011] Verbauwhede, I., Karaklajic, D., and Schmidt, J.-M. (2011). The Fault Attack Jungle - A Classification Model to Guide You. In Proceedings of the 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2011), IEEE.

[Yuce et al., 2017] Yuce, B., Ghalaty, N. F., Deshpande, D., Santapure, H., Conor, P., Nashandali, L., and Schaumont, P. (2017)
Analyzing the Fault Injection Sensitivity of Secure Embedded Software. ACM Trans. Embedded Comput. Syst. 16(4): 95:1-95:25

[Yuce et al., JHSS 2018] Yuce, B., Schaumont, P. and Witteman, M. Fault. Attacks on Secure Embedded Software: Threats, Design, and Evaluation. Journal of Hardware and System Security, May 2018.

[Zussa et al., 2014] Zussa, L., Dehbaoui,A. , Tobich,K. , Dutertre, J-M., Maurine, P., Guillaume-Sage, L., Clediere, J., and Tria, A. Efficiency of a glitch detector against electromagnetic fault injection. In Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014.

# References

[Akkar et al., 2003] Akkar, M.-L., Goubin, L., and Ly, O. (2003). Automatic Integration of Counter-Measures Against Fault Injection Attacks. In Proceedings of E-Smart'2003, pages 1–13, Nice.

[Barenghi et al. 2010] Barenghi, A., Breveglieri, L., Koren, I., Pelosi, G., and Regazzoni, F. (2010). Countermeasures against fault attacks on software implemented AES. In Proceedings of the 5th Workshop on Embedded Systems Security (WESS'10), pages 1–10, ACM Press.

[Barry et al., 2016] Barry, T., Couroussé, D., and Robisson, B. (2016). Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In Proceedings of the 3rd Workshop on Cryptography and Security in Computing Systems (CS2), pages 1–6, ACM Press.

[De Keulenaer et al., 2016] De Keulenaer, R., Maebe, J., De Bosschere, K., and De Sutter, B. (2016). Link-time smart card code hardening. International Journal of Information Security, 15(2):111–130.

[Goloubeva et al., 2005] Goloubeva, O., Rebaudengo, M., Sonza Reorda, M., and Violante, M. (2005). Improved software-based processor control-flow errors detection technique. In Annual Reliability and Maintainability Symposium, pages 583–589, IEEE.

[Lalande et al., 2014 ] J-F. Lalande et al. *Software countermeasures for control flow integrity of smart card C codes.* ESORICS 2014.

[Oh et al. ] [Oh et al., 2002a] Oh, N., Shirvani, P. P., and McCluskey, E. J. (2002a). Control-flow checking by software signatures. IEEE Transactions on Reliability, 51(1):111–122.

[Rauzy et al., 2015] Pablo Rauzy, Sylvain Guilley. A formal proof of countermeasures against fault injection attacks on CRT-RSA. J. Cryptographic Engineering 4(3): 173-185 (2014)

[Reis et al., 2005] Reis, G., Chang, J., Vachharajani, N., Rangan, R., and August, D. (2005). SWIFT: Software Implemented Fault Tolerance. In Proceedings of the 2005 International Symposium on Code Generation and Optimization (CGO'05), pages 243–254. IEEE.

[SIED, 2003] [Nicolescu et al., 2003] Nicolescu, B., Savaria, Y., and Velazco, R. (2003). SIED: Software Implemented Error Detection. In Proceedings of the 18th International Symposium on Defect and Fault Tolerance in VLSI Systems