Automated combination of tolerance and control flow integrity countermeasures against multiple fault attacks on embedded systems

Thierno **Barry**

PhD Candidate in security of Embedded Systems
at CEA (*the French Atomic Energy Commission*)
`thierno.barry@cea.fr`

Damien Couroussé (CEA) Karine Heydemann (LIP6) Bruno Robisson (CEA)

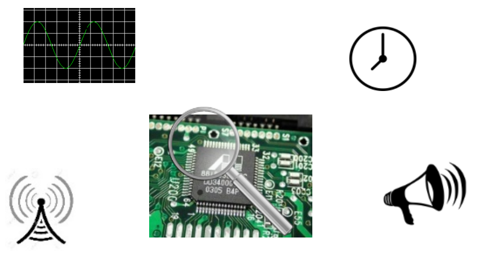
2017 European LLVM Developers' Meeting
March 28, 2017, Saarbrücken, Germany

CONTEXT

- Embedded systems have increasingly become critical part of our daily life

- One of the major threats against these systems are **physical attacks**

Side Channel Attacks



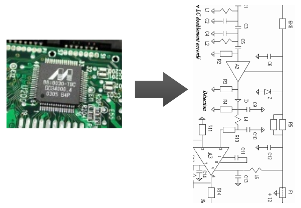
Fault Injection Attacks

- These attacks essentially aim to:

Obtain sensitive data

Bypass protections

Reverse engineering



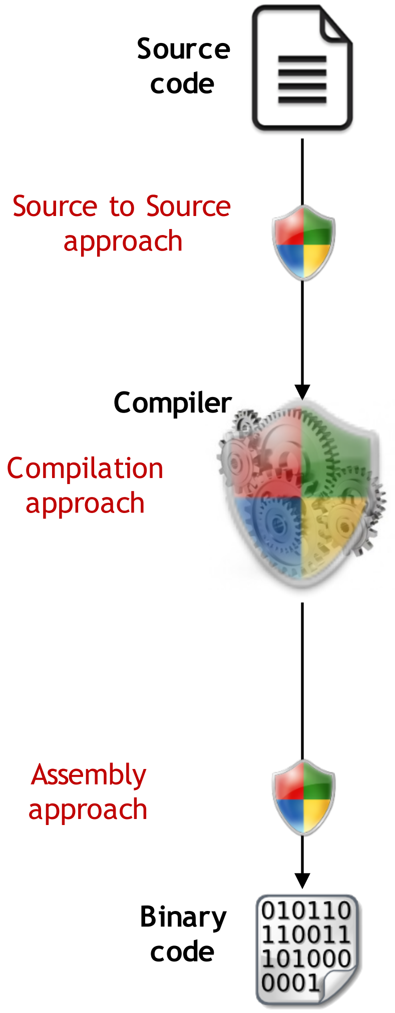
- The security of these systems reveals itself as major concern for both industrials and state organizations

Our work consists in generating codes that are protected against these attacks

STATE OF THE ART

Motivations

- A number of software-based countermeasures against fault attacks already exist



- Security properties cannot be guaranteed after code compilation [Balakrishan et al. 2008]
- Except if the compiler code optimizers are disabled as suggested in [Eldib et al. 2014]
  ➔ leads to a very high overheads +400% in [Lalande et al. 2014]

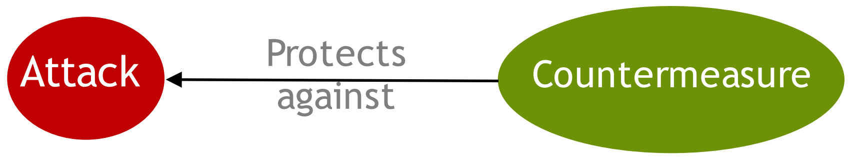
- Unlike the source to source approach we have control over code optimizers
- Unlike assembly approach we have the benefit of code transformation opportunities provided by the compiler

  ➔ Allows to reduce the security overhead

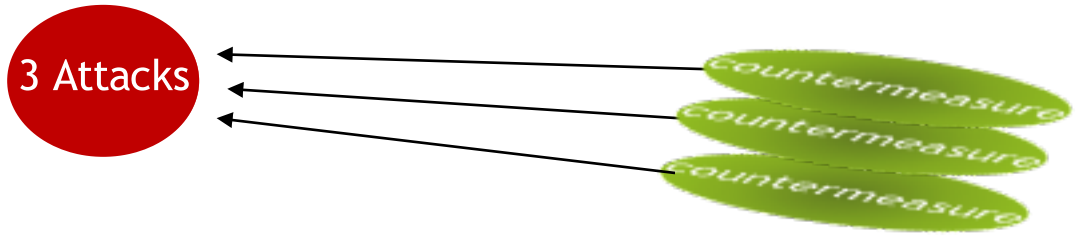
- Lack of semantic information
- Several transformations need to be performed
  ➔ leads to significant overheads [Moro et al. 2014]

Motivations

- Each countermeasure is designed to protect against one single attack



- When it comes to protect against several attacks:



➔ Countermeasures are manually superposed

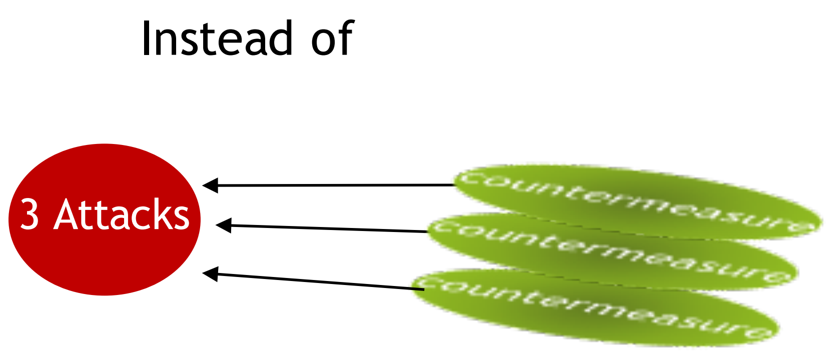
➔ Interactions between countermeasures are not considered

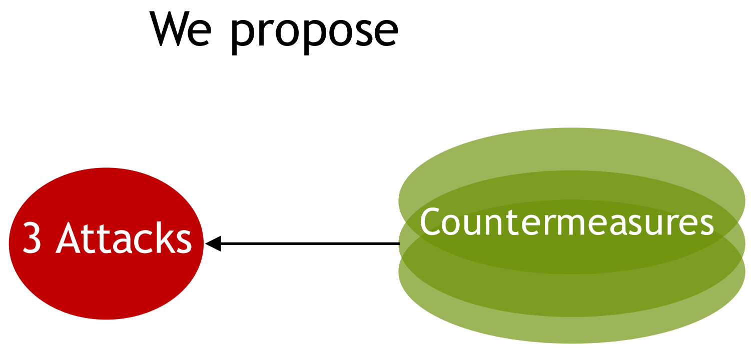
And yet

[Regazzoni et al. 2008] and [Luo et al. 2014] have demonstrated that a code protected against fault attacks may become more vulnerable to side channel attacks

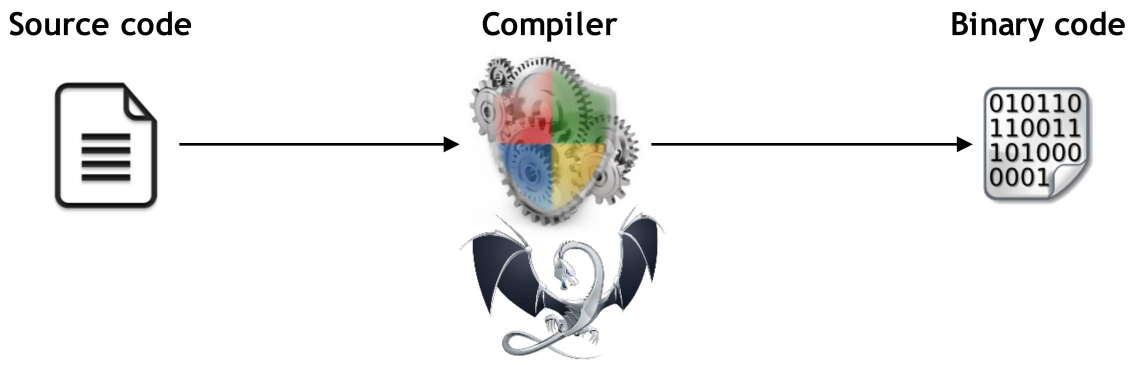
OUR APPROACH

1 Composition approach



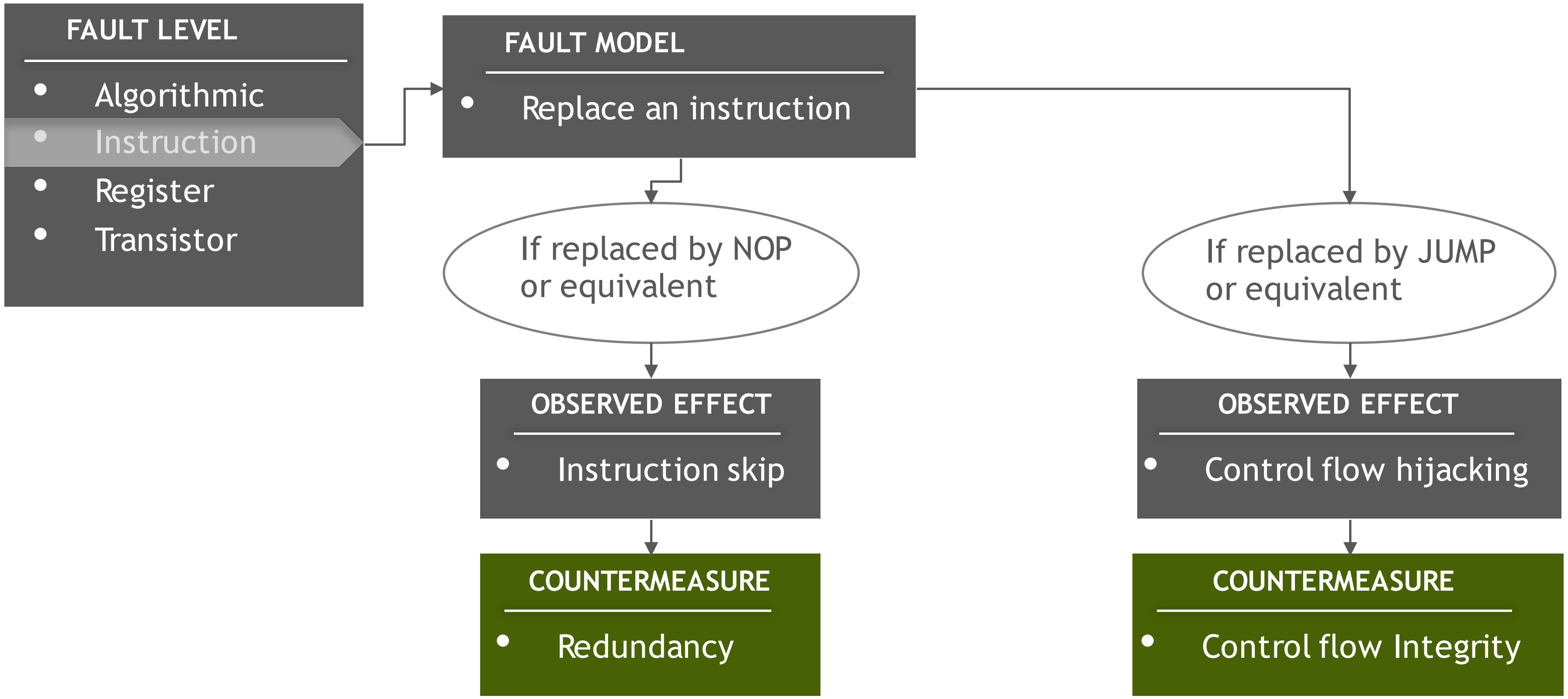


2 Compilation approach



Fault Injection Attacks

- A fault may occurs at different levels

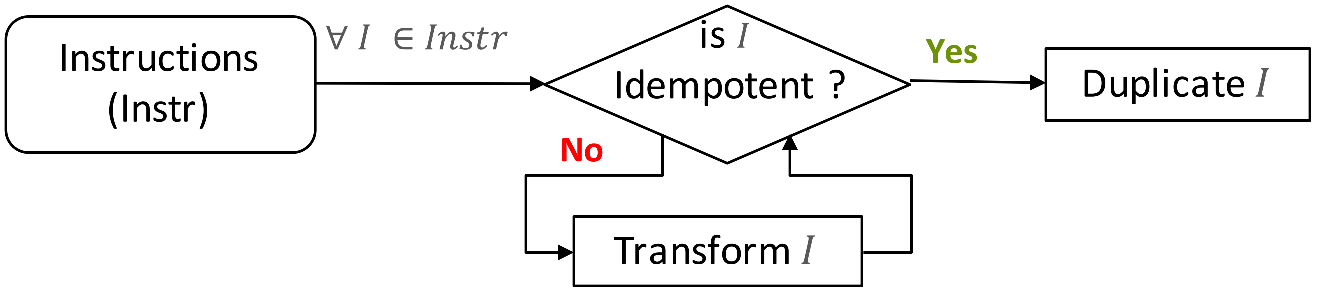


- Our implemented countermeasure resists against:
  - Multi-fault that lead to skip N instructions
  - Fault that leads to skip W bytes
  - Control flow hijacking

N and **W** are arguments of our compiler

Instruction **redundancy**

TOLERANCE SCHEME



"An instruction is idempotent when it can be **re-executed** several times with always the same result"

EXAMPLE

is idempotent

```
add R0, R1, R2
```

Duplication →

```
add R0, R1, R2
add R0, R1, R2
```

Is not idempotent

```
add R1, R1, R2
```

✗ →

```
add R1, R1, R2
add R1, R1, R2
```

Transformation [Moro et al. 2014]

```
mv RX, R1
add R1, RX, R2
```

Duplication →

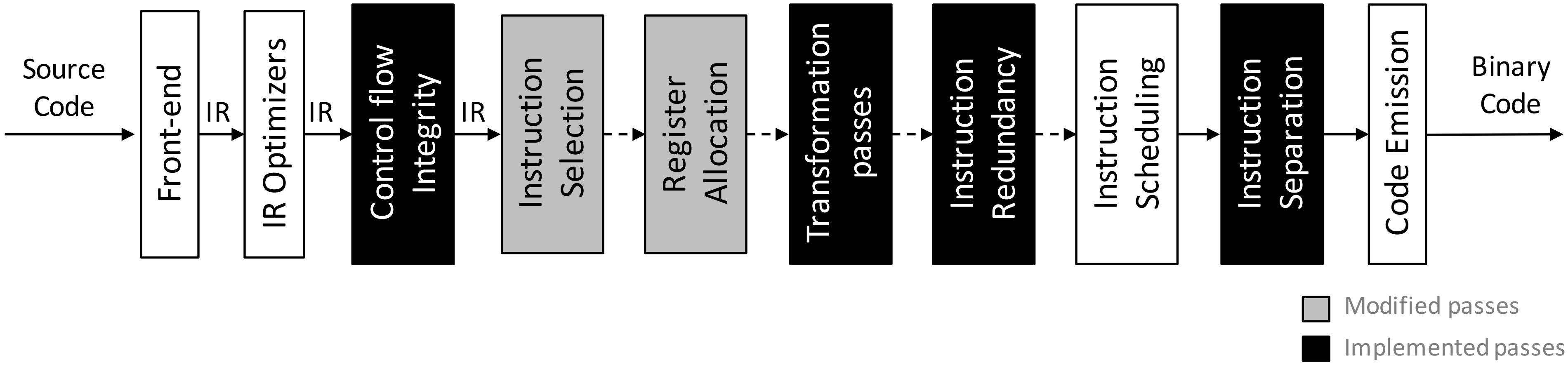
```
mv RX, R1
mv RX, R1
add R1, RX, R2
add R1, RX, R2
```

LIMITATIONS

- **How to find free registers at this level**
  - For [Barenghi et al. 2010] The number of free registers are known for their implemented AES
  - For [Moro et al. 2014] Using the ARM scratch register `r12`
- **Overhead**
  - At least $\times 4$ for each non-idempotent instruction
  - [Moro et al. 2014] reported $\times 14$ for `umlal`

COMPILATION APPROACH

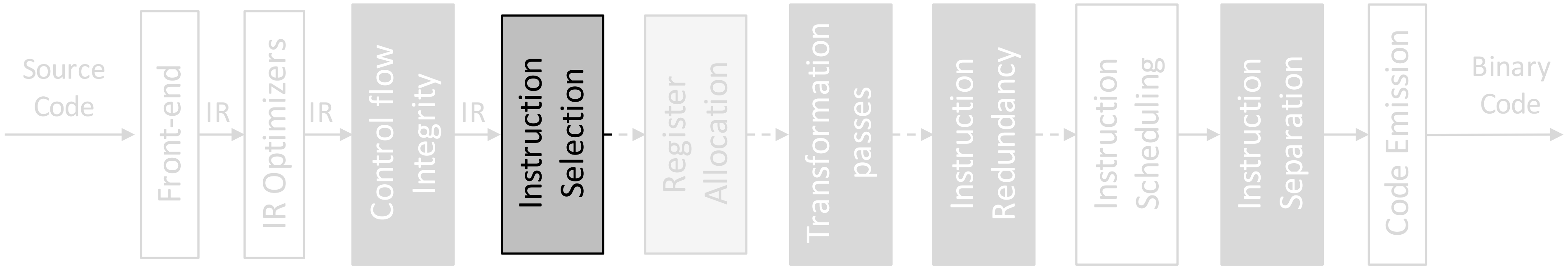
- The internal structure of our compiler is



COMPILATION APPROACH

- The internal structure of our compiler is

▨	Modified passes
■	Implemented passes



This pass is modified in such a way that idempotent instructions are the ones privileged during the selection

EXAMPLE

For the operation: $a * b + c$

`mul` and `add` are selected instead of `mla`

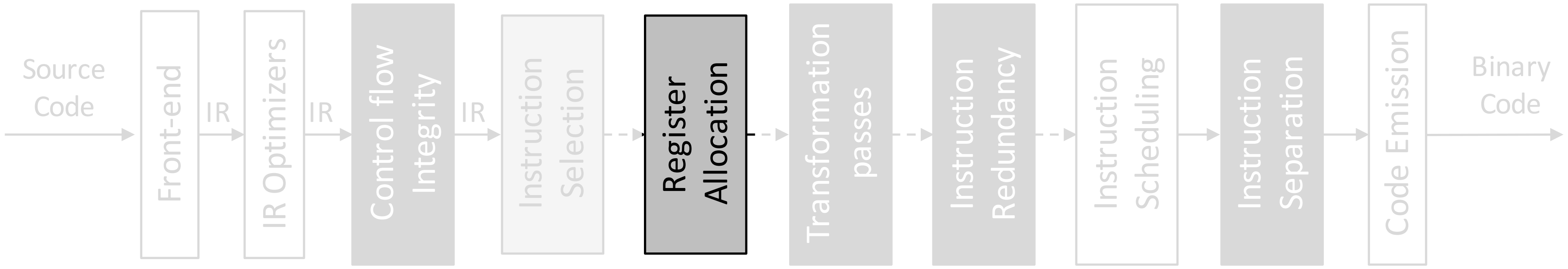
`mla` is not idempotent

But `mul` and `add` can be idempotent if the source and destination registers are different

COMPILATION APPROACH

- The internal structure of our compiler is

▨ Modified passes
■ Implemented passes



This pass is modified to introduce a constraint so that: destinations registers are always different to sources ones

EXAMPLE

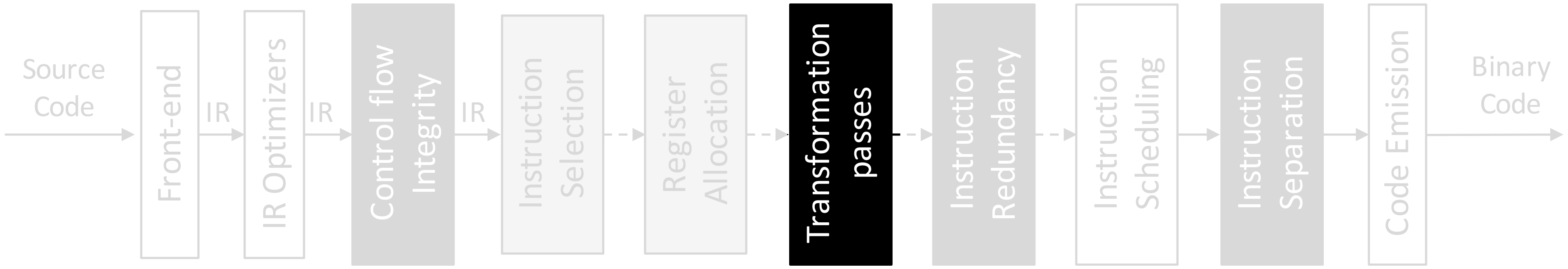
For the operation: $a = b + c$

instead of having:
```
add R0, R0, R1
```
we have something like:
```
add R0, R1, R2
```
$\xrightarrow{\text{Duplication}}$
```
add R0, R1, R2
add R0, R1, R2
```

COMPILATION APPROACH

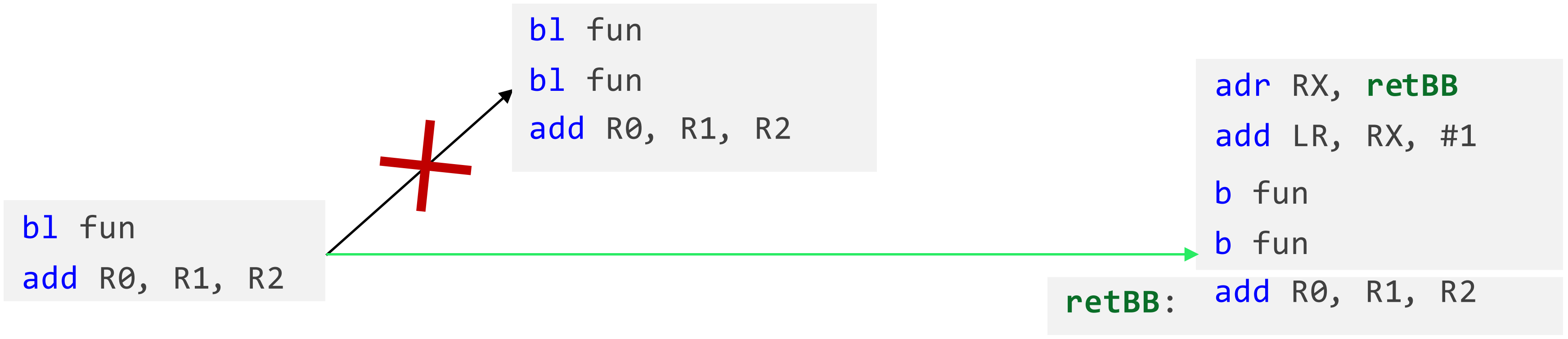
- The internal structure of our compiler is

▨ Modified passes
■ Implemented passes



BL Elimination Pass

The role of these passes is to handle instructions that need special treatments



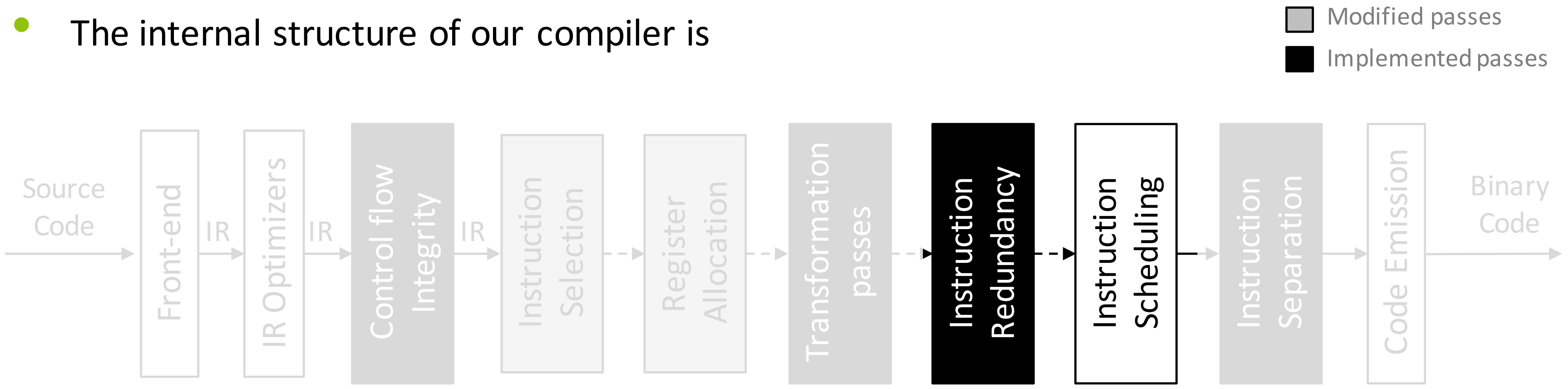
```
bl fun
add R0, R1, R2
```

```
bl fun
bl fun
add R0, R1, R2
```

```
        adr RX, retBB
        add LR, RX, #1
        b fun
        b fun
retBB:  add R0, R1, R2
```

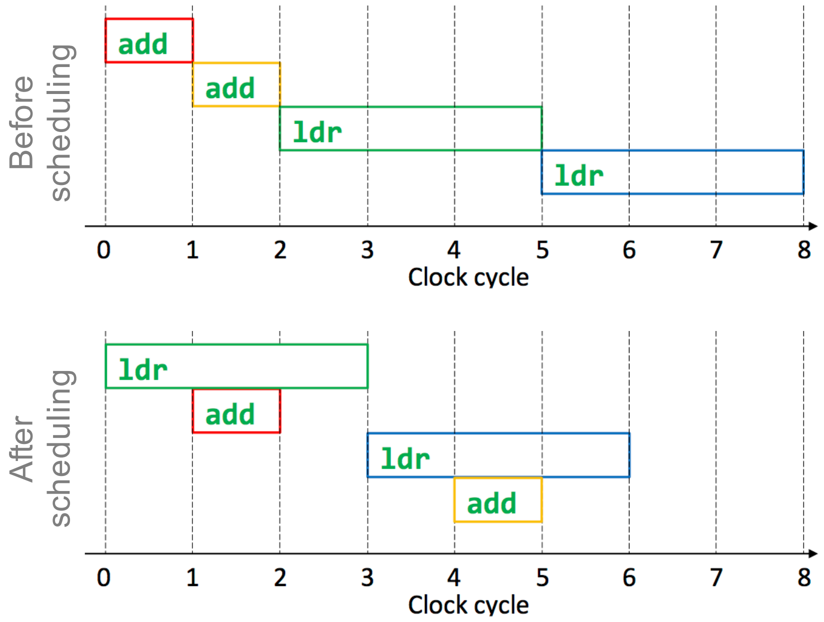
COMPILATION APPROACH

- The internal structure of our compiler is



Example:

```
add R0, R1, R2
add R0, R1, R2
ldr R3, [R1, #4]
ldr R3, [R1, #4]
```



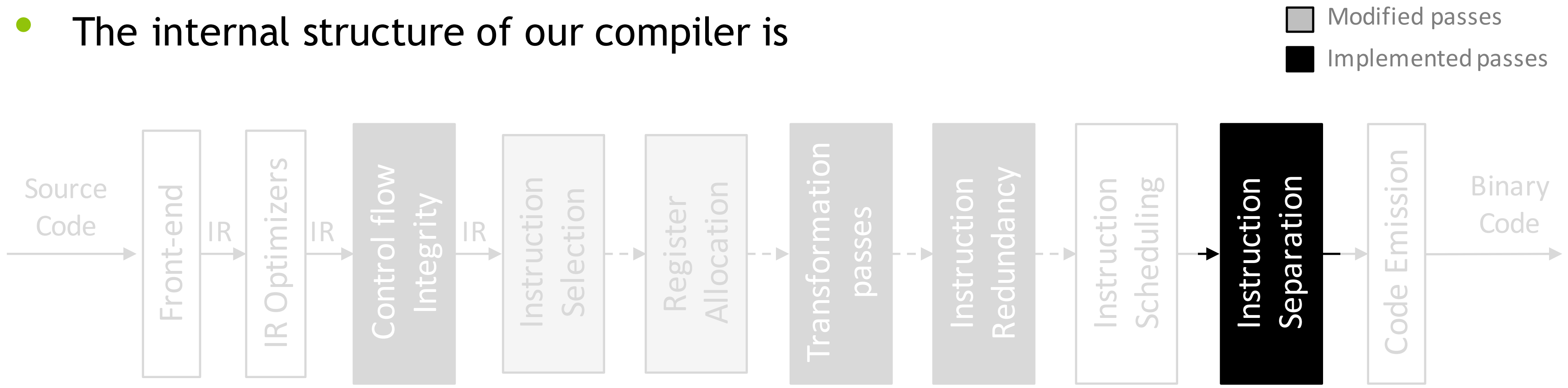
Advantages:

1. Performance
2. Security

→ to prevent faulting the original and duplicated instruction simultaneously

COMPILATION APPROACH

- The internal structure of our compiler is



The role of this pass is to leave the required distance between redundant instructions to protect against fault models for which the with > size of an instruction

EXAMPLE

- [Moro et al. 2014]: protects against fault that are >= 32-bit of width on an ARM Cortex-M3
  → 16-bit instructions are disabled ➔ ++ code size

- [Rivière et al. 2015]: successfully injected faults that are = 64-bit of width
  → Moro et al's solution doesn't work

Our scheme resists against both of these attack models

- Without disabling 16-bit instructions encoding
- By simply providing the right parameters to our compiler

EXPERIMENATL EVALUATION

- Comparison with *Moro et al.*'s result, using the same benchmarks and same architecture
- **Target architecture**: ARM Cortex-M3 **Benchmark : AES (MiBench) Size**: bytes

Performance Evaluation

Opt. flags	Overhead	
	Execution time	size
O0	× 1.66	× 2.28
O3	× 1.98	× 2.16

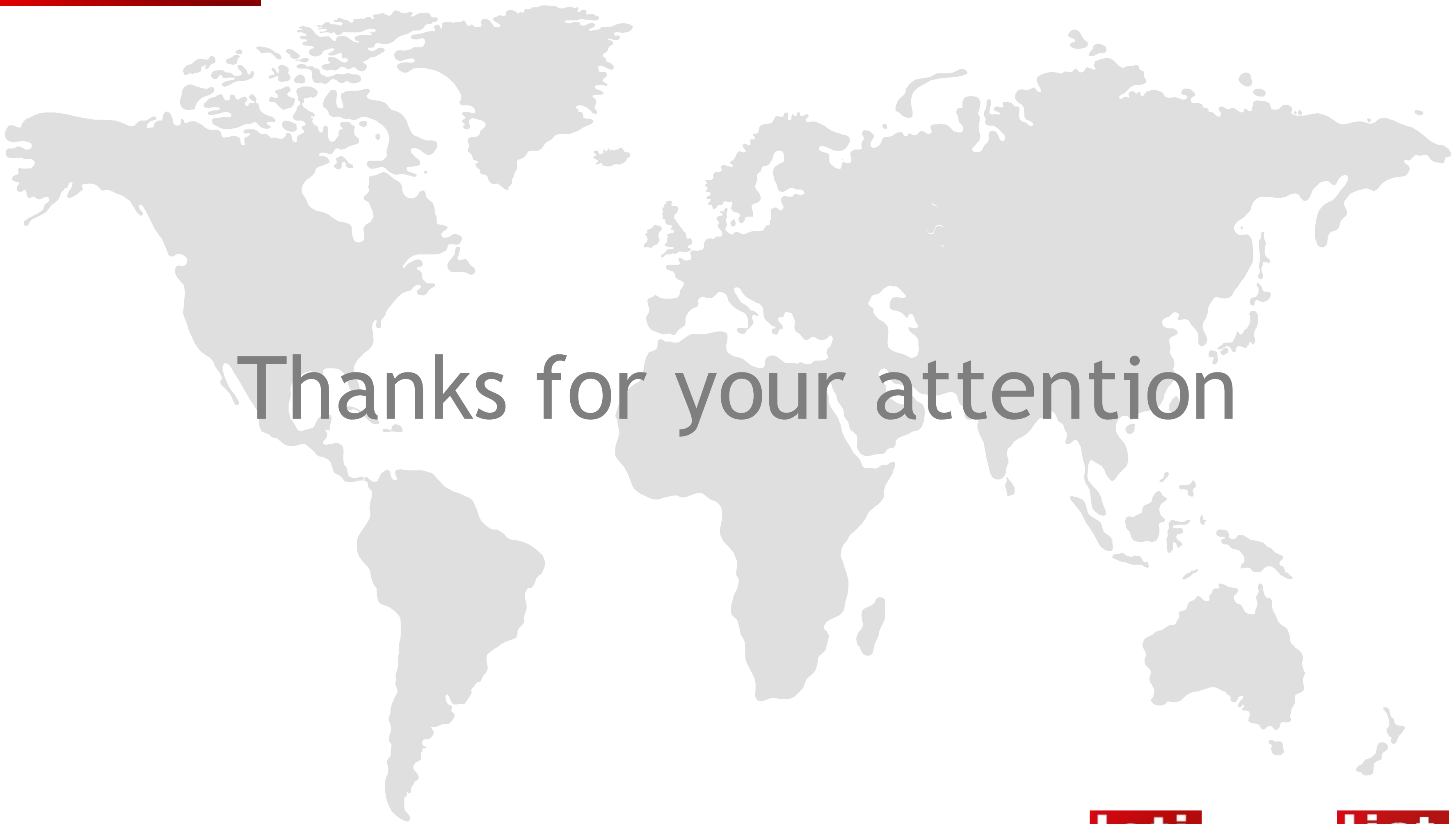
Moro et al 2014	
Execution time	Size
× 2.14	× 3.02

COMPARED TO *Moro et al.*

Best case: we are 22% better in execution speed and 25% in code size
Worst case: 6% better in execution speed and 26% better in code size

Security Evaluation

- We successfully resisted against the following models of fault injections
  - ✓ Single fault that skips one instruction
  - ✓ Single fault that skips one W-instruction
  - ✓ N simultaneous faults where each fault skips one instructions
  - ✓ N simultaneous faults where each fault skips W-instructions
  - ✓ Control flow hijacking



Thanks for your attention

Thierno **Barry**

PhD Candidate in Security of Embedded Systems at CEA

thierno.barry@cea.fr

http://thiernobarry.fr

Centre de Grenoble
17 rue des Martyrs
38054 Grenoble Cedex

Centre de Saclay
Nano-Innov PC 172
91191 Gif sur Yvette Cedex