Compilation for the Composition of Software Protections for Embedded Systems

Thierno Barry, Damien Couroussé, Bruno Robisson

► **To cite this version:**

Thierno Barry, Damien Couroussé, Bruno Robisson. Compilation for the Composition of Software Protections for Embedded Systems. 5ème édition de la rencontre Crypto'Puce, May 2015, Île de Porquerolles, France. cea-01273410

HAL Id: cea-01273410

https://cea.hal.science/cea-01273410v1

Submitted on 13 Feb 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FROM RESEARCH TO INDUSTRY

cea tech

Compilation for the composition of software protections for embedded systems

Thierno BARRY[1]

Damien COUROUSSÉ[1]

Bruno ROBISSON[2]

[1]CEA – LIST / DACLE

[2]CEA / DPACA

Firstname.LASTNAME@cea.fr

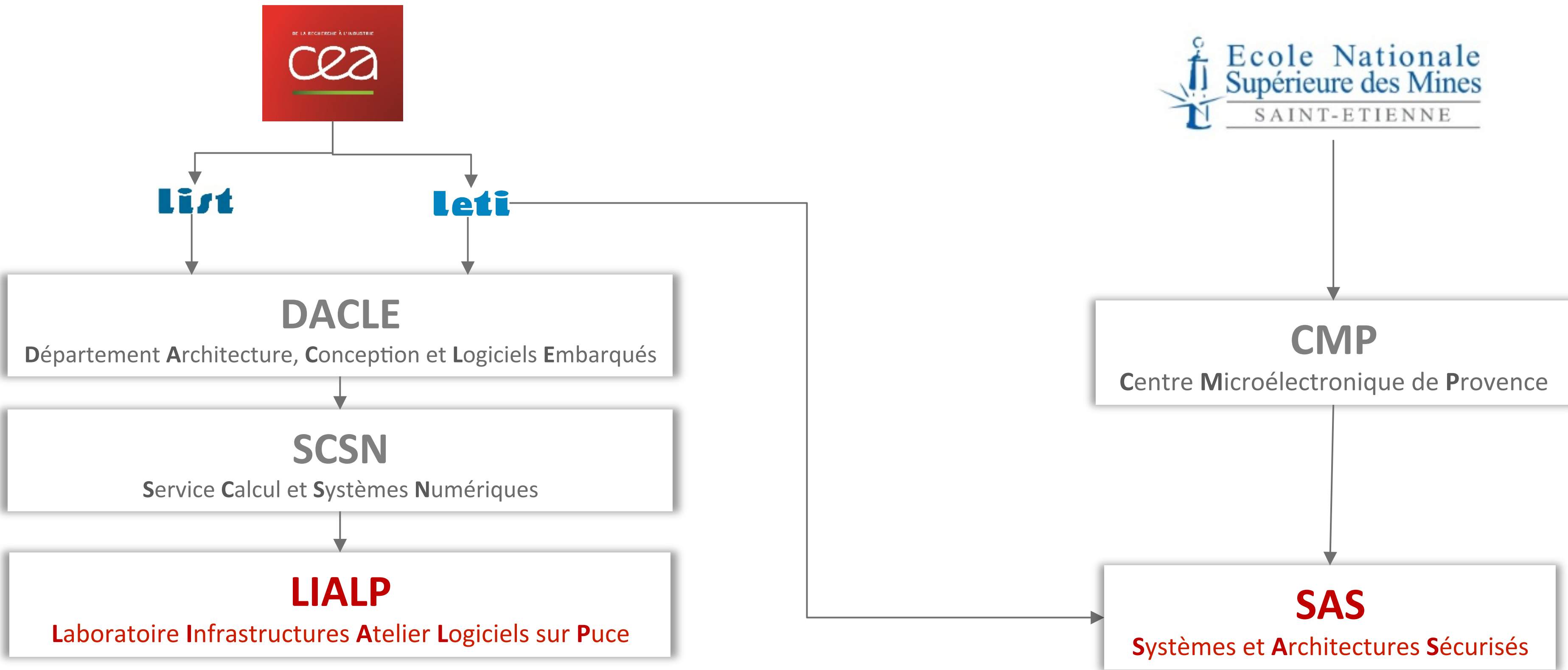
Crypto'Puce 2015

Porquerolles Tuesday, May 5, 2015

www.cea.fr

leti & list

AFFILIATION



- Automatic control (centralized and distributed)
- Middleware and communication
- Compilation and code generation
- Methods and tools: design flow for HW/SW integration

- Hardware security

© CEA. All rights reserved

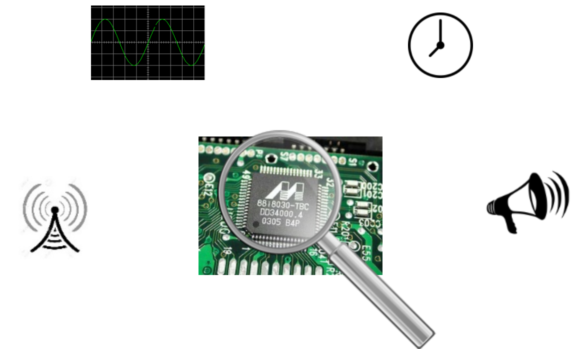
CONTEXT

- Nowadays, embedded systems have increasingly become critical part of our daily life
- One of the major threats against these systems are **physical attacks**

There are two main categories

1 Side channel attacks

Observing physical quantities of the device during operation



2 Fault attacks

Injecting a fault in order to disrupt the normal functioning of the device

© CEA. All rights reserved

OBJECTIVE

- Proposing a **tool** for composing several software protections against physical attacks

- Through a compilation toolchain

- Our work involves two disciplines:

  1. Physical security
  2. Compilation

also called: Compilation for security

© CEA. All rights reserved

OUTLINE

- Existing countermeasures against physical attacks
  - Concluding remarks
- Our approach
- A safari inside a compiler
  - why compilation + security is not obvious ?
- Why operating inside a compiler?
- First results
- Outlook

© CEA. All rights reserved

EXISTING COUNTERMEASURES

Side Channel Attacks

- Work because there is a correlation between the operations being processed and some observable physical quantities
- The objective of countermeasures is:

- Two concepts:

1 Masking

Concealing each intermediate value ***v*** by a random value ***m*** such as: $\boldsymbol{v_m = v \; op \; m}$

$\mathbf{v_m = v \oplus m}$ → Boolean

$\mathbf{v_m = v + m}$ → Modular addition

$\mathbf{v_m = v \times m}$ → Modular multiplication

2 Hiding

- Software
  - Insertion of dummy instructions
  - Instructions shuffling
- Hardware
  - Randomize the power consumption
  - Equalize the power consumption

© CEA. All rights reserved

EXISTING COUNTERMEASURES

Fault Attacks

- Based on fault models where an attacker can:
  - Skip an instruction
  - Replace an instruction with another one
  - Corrupt data being transferred from/to memory
- Proposed countermeasures are:
  - Instructions redundancy
  - Control flow hardening
  - CRC / Parity Check / …

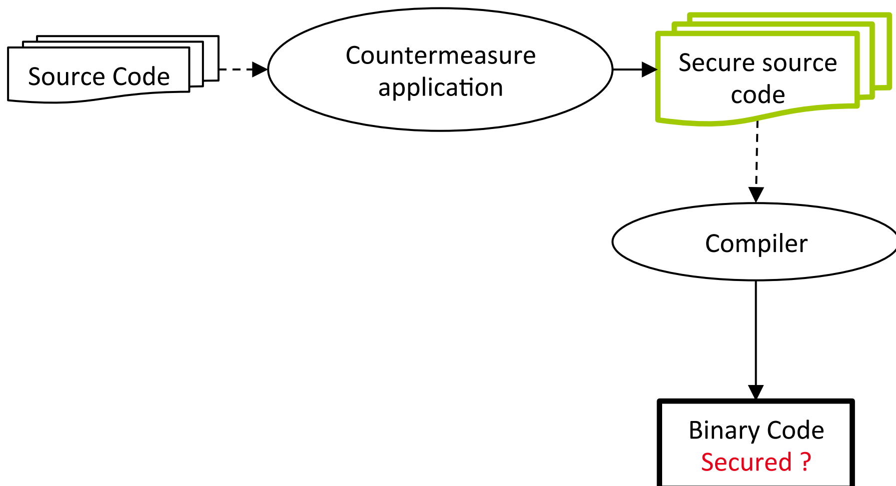
© CEA. All rights reserved

EXISTING COUNTERMEASURES

Concluding

- We notice two approaches for applying countermeasures

1 At the source code level



source to source approach

Problems:

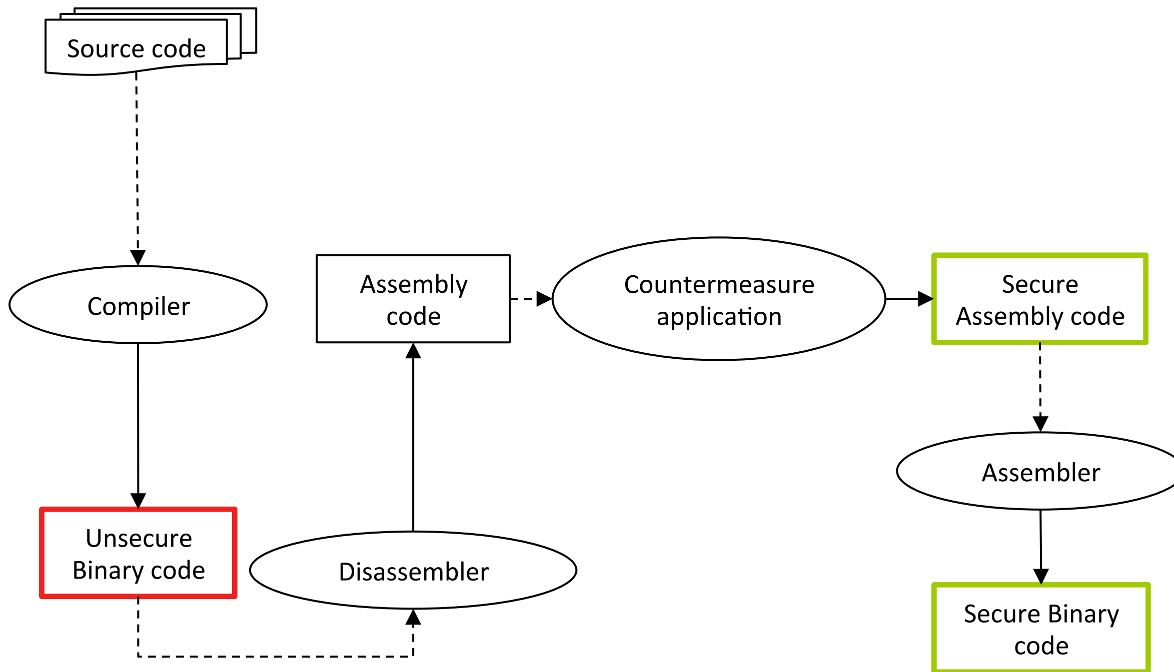
- None of security properties applied to the source code are guaranteed after the compilation
- Except if all the compiler code optimizers are disabled as suggested in [Eldib et al. 2014]
- Leads to very high execution overheads: ➔ + 400% in [Lalande et al. 2014]

© CEA. All rights reserved

EXISTING COUNTERMEASURES

Concluding

2 At Assembly level



Assembly approach

Problems:

- Lack of visibility program context
  - ➔ Overheads ++
- Often ad-hoc [Barenghi et al., 2010]

© CEA. All rights reserved

EXISTING COUNTERMEASURES

Concluding

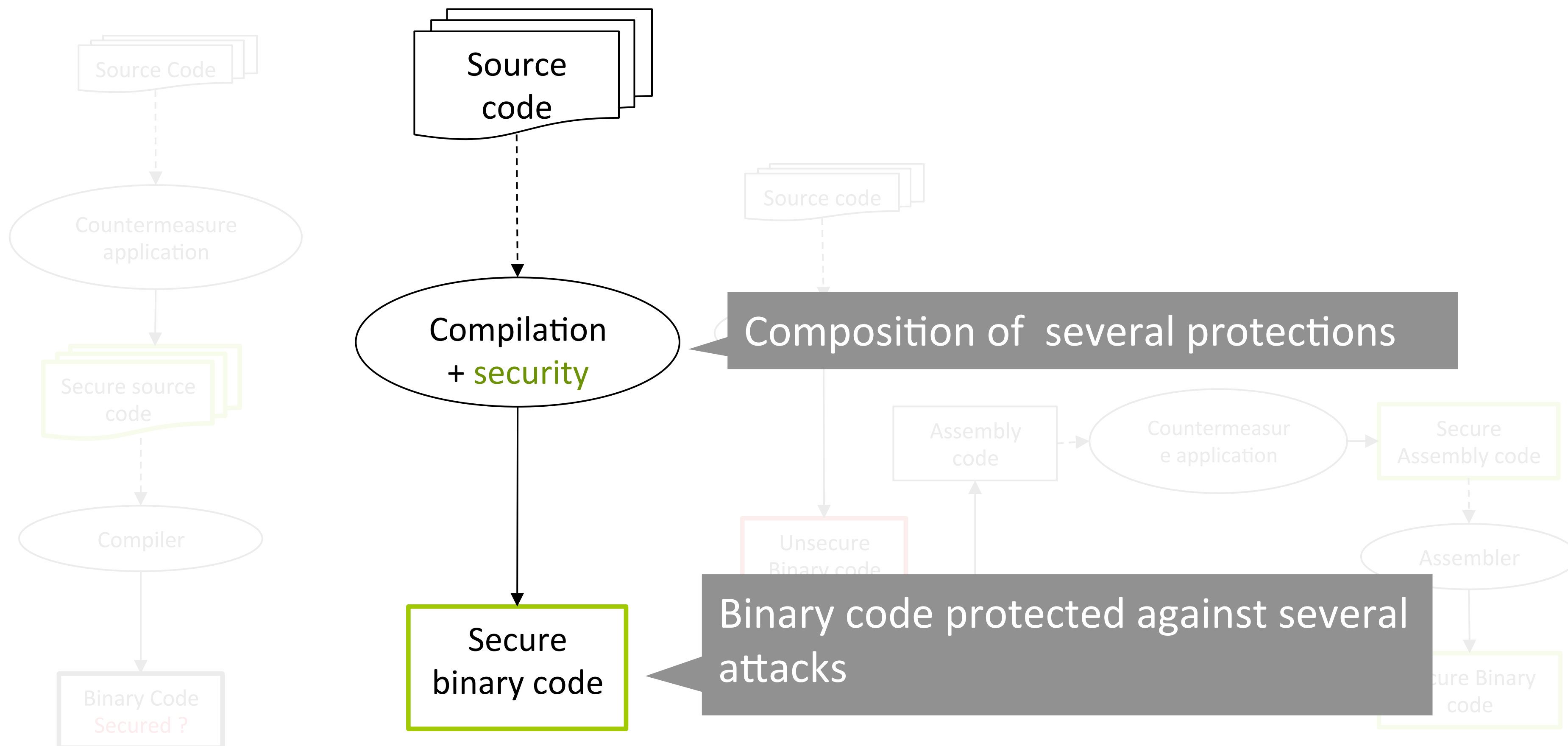
- A countermeasure is designed to protect against one single attack

- [Regazzoni et al. 2008] and [Luo et al. 2014] have shown that a code protected against Fault attacks may increase the power leakage
  ➔ and then become more vulnerable to power analysis attacks

How to take into account several threats inside a countermeasure ?

© CEA. All rights reserved

OUR APPROACH



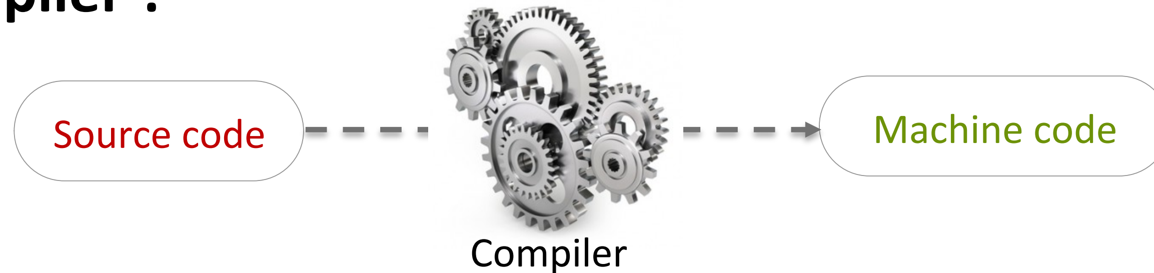
© CEA. All rights reserved

Outline

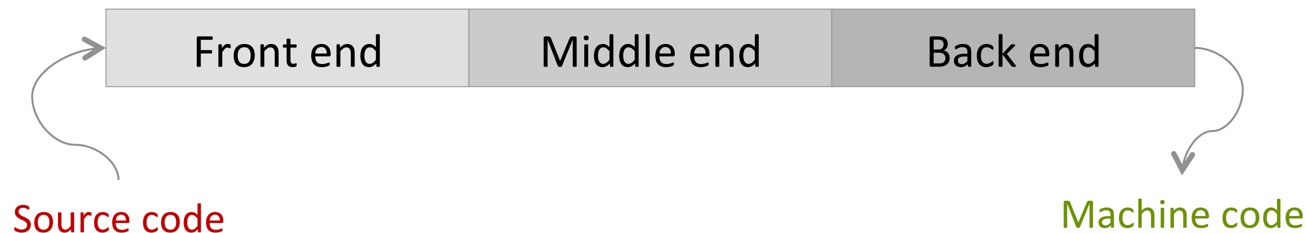
- Existing countermeasures against physical attacks
  - Concluding remarks
- Our approach
- A safari inside a compiler
  - why compilation + security is not obvious ?
- Why operating inside a compiler?
- First results
- Outlook

© CEA. All rights reserved

What is a compiler ?



- The source code passes through several transformations and representation before the Machine code
- Each one is suitable for some kind of tasks of the compiler
- Modern compilers are structured in 3 phases:



© CEA. All rights reserved

A SAFARI INSIDE A COMPILER

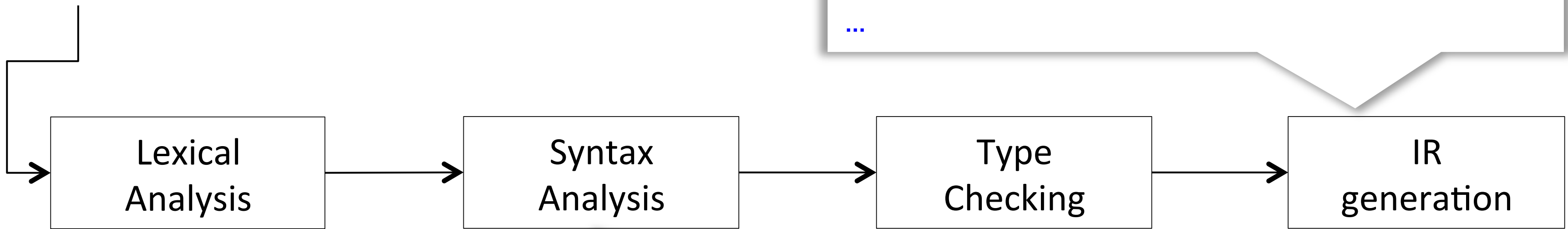
Front end

```
If (x > 0){
    return a+b;
}
```

Source code

Simplified LLVM-IR

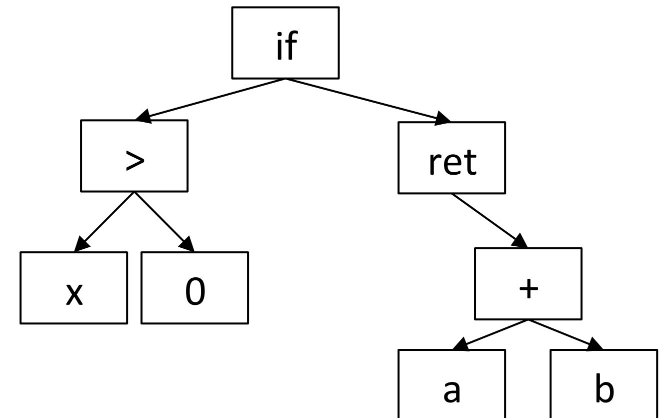
```
…
%cmp = icmp sgt i32 %x, 0
br i1 %cmp, label %if.then, …

%if.then:
    %add = add nsw i32 %a, %b
    store i32 %add, i32* %retval
    br label %return
…
```



Reads the source code and splits it into a list of tokens e.g.:

if	(	x	>	0	)	{
Return		a	+	b	;	}

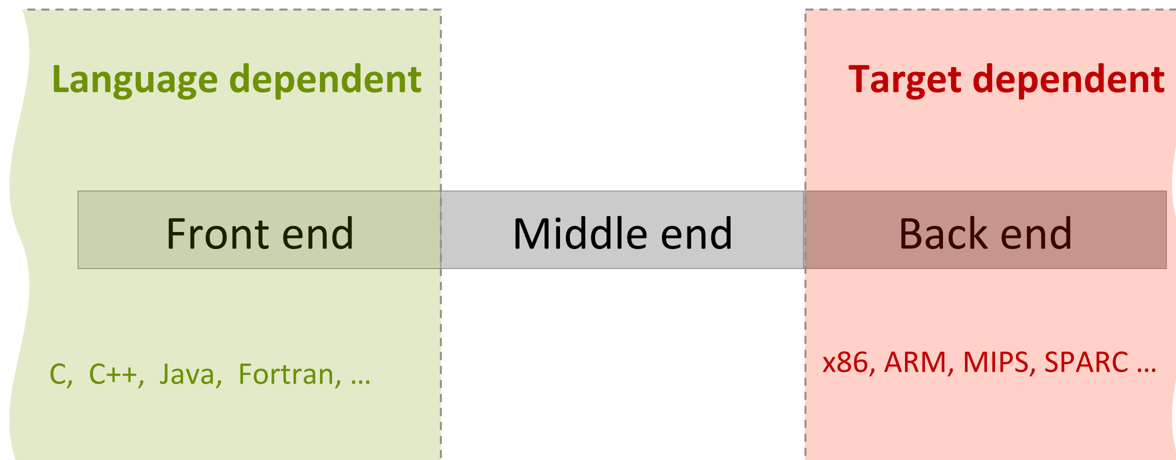
Take the list of tokens, built the AST ➔ check the validity of the syntax



 © CEA. All rights reserved

Middle end

- Takes as input the Intermediate representation
- The IR is supposed to be language and target independent



- A countermeasure applied at the *middle end* remain valid for all languages and targets supported by the compiler

© CEA. All rights reserved

Middle end

- The majority of code optimizer are applied at *middle end*
- Among them we have:
  - Global Value Numbering (GVN) — Remove all redundant instructions
  - Dead Code Elimination (DCE) — Remove all unreachable instructions
  - Dead Store Elimination (DSE) — Remove memory writings that are never read

```
int x = 0;
int y = f(x);
for(int i=1; i<= 100; i++)
    if(i > 0)
        x = x + 1;
    else
        x = x - 1;
y = f(x)
```

```
int x = 0;
int y = f(x);
for(int i=1; i<= 100; i++)
    x = x + 1;
y = f(x)
```

```
int x = 0;
int y = f(x);
x = 100;
y = f(x)
```

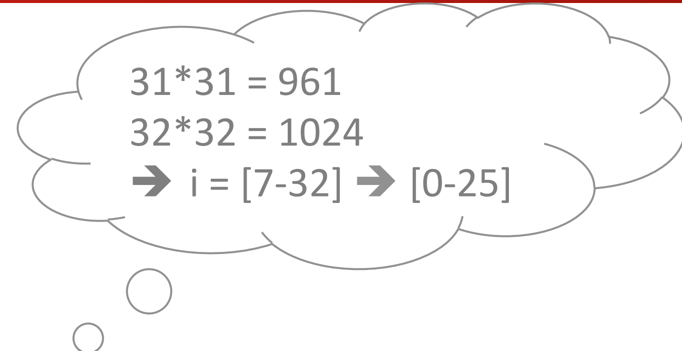
```
int y = f(0);
int x = 100;
y = f(x)
```

```
int y = f(100);
```

 © CEA. All rights reserved

Middle end

- Loop Invariant Code Motion (LICM)
- LOOP-UNROLLING / LOOP UNSWITCH




```
bool flag;
for(int i=7; i*i< 1000; i++){
    flag = verdict(1);
    if(flag == true)
        foo();
    else
        bar();
}
```

```
bool flag;
for(int i=0; i<25; i++){
    flag = verdict(1);
    if(flag == true)
        foo();
    else
        bar();
}
```

```
bool flag = verdict(1);
for(int i=0; i<25; i++){
    if(flag == true)
        foo();
    else
        bar();
}
```

```
bool flag = verdict(1);
if(flag == true)
  for(int i=0; i<25; i++)
      foo();
else
  for(int i=0; i<25; i++)
      bar();
```


© CEA. All rights reserved

Back end

Takes the IR as input

- Instructions selections
  - Convert the IR to a representation close to the target architecture
- Register allocation
  - Find the best way to assign physical registers to variables in order to reduce register pressure and avoid memory spills
- Instruction scheduling
  - Rearrange instructions to obtain the best execution order in order to avoid *stalls* inside the pipeline
- Machine code emission
  - Emit executable code that is target-specific

© CEA. All rights reserved

COMPILATION VS. SECURITY

	COMPILATION	SECURITY
GOAL	■ Generation of executable code for a target architecture ■ Making the execution as fast as possible	■ Safety ■ Resistance against attacks
HOW?	■ Removing any instruction redundancy ■ Dead code/store elimination ■ Smart scheduling ■ Simplifying and combining operation	■ Adding instruction redundancy ■ Insertion of dummy instructions ■ Random scheduling (*shuffling*) ■ Masking intermediate values

© CEA. All rights reserved

WHY OPERATING INSIDE A COMPILER?

1. We have a complete view on the program being compiled
   - Possibility to reduce the cost of the security
2. We have control over code optimizers
   - We can decide where and when to apply security
   - We can ensure that the security won't be removed by the compiler
   - We can take advantage of code optimization
3. We can scale the security level relative to optimization level

© CEA. All rights reserved

Instruction duplication (ID) inside the compiler

1 With a very optimal overhead thanks to our hacked register allocator

WHY?

With an **Assembly** approach, when comes to duplicate an instruction like:

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

Just doing

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

is invalid because R0 is both source and destination

An extra available register is needed to save R0:

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

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

How to find an extra available register?

1. you are designing an ad-hoc countermeasure and you know how many registers are available [Barenghi et al., 2010]
2. you parse your assembly code (not easy)
3. Save an restore

```
push R2
mov R2, R0
add R0, R2, R1
pop R2
```

```
push R2
mov R2, R0
mov R2, R0
add R0, R2, R1
add R0, R2, R1
pop R2
```


© CEA. All rights reserved

Instruction duplication (ID) inside the compiler

1 With a very optimal overhead thanks to our hacked register allocator (RA)

We modified our RA in such a way that the destination register is always different to source registers:

```
opcode Rdst, Rsrc1, Rsrc2
```

(Rdst != Rsrc1) and (Rdst != Rsrc2)

THAT'S WHY

Instead of generating:

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

We automatically generate:

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

and duplicating such an instruction is straightforward

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

X 2

with a reduced overhead compared to:

X 4

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

and X 6

```
push R2
mov R2, R0
mov R2, R0
add R0, R2, R1
add R0, R2, R1
pop R2
```


© CEA. All rights reserved

Instruction duplication (ID) inside the compiler

1. With a very optimal overhead thanks to our hacked register allocator (RA)

2. Our duplication process is done before the instruction scheduling
   - The compiler will rearrange the instructions in order to find the best execution order
   - As a consequence:
     - ➔ duplicated instructions may not necessary be glued

➔ improve the execution speed

© CEA. All rights reserved

OUTLOOK

Our objective is not to produce new unknown countermeasure

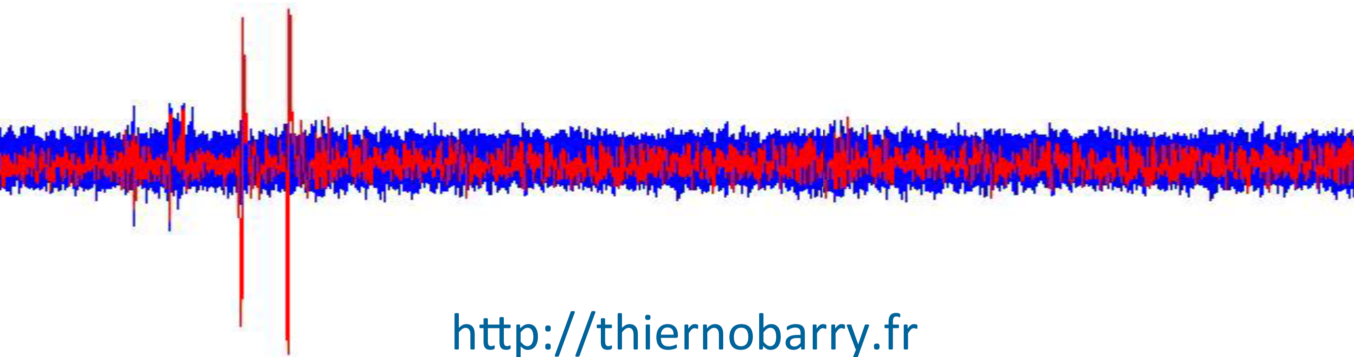
BUT

Finding a way to combine them in a single tool, without marginalizing the execution performance

- The next step is to implement power analysis countermeasures in our compiler
- And then implementing a unified countermeasure model
- Proving the validity of the model

© CEA. All rights reserved

Thank you for your attention



http://thiernobarry.fr

leti
Centre de Grenoble
17 rue des Martyrs
38054 Grenoble Cedex

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