On-line Instruction Flow Obfuscation: Formal Proof and Proposal for Implementation on CVA6

Common meeting CT SED and GT AFSEC

30/01/2025

Théo Serru
theo.serru@univ-nantes.fr

Jean-Luc Béchennec
Loïg Jezequel

Mikaël Briday
Sébastien Faucou

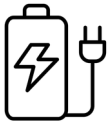
ANR-21-CE-39-0017

Context : hardware mitigation of side-channel attacks

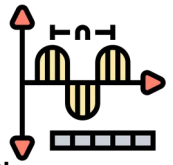
Side-channel attacks **exploit**

Cache

Timing



Power



Electromagnetic waves

Countermeasures

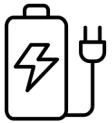
Software	Hardware
• Detection and response • Obfuscation • Randomization • Masking • Splitting	• Use of custom gates • Minimization of metal artifacts • Redundancy • Detection and response

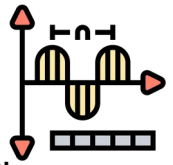
Context : hardware mitigation of side-channel attacks

Side-channel attacks **exploit**


Cache


Timing


Power

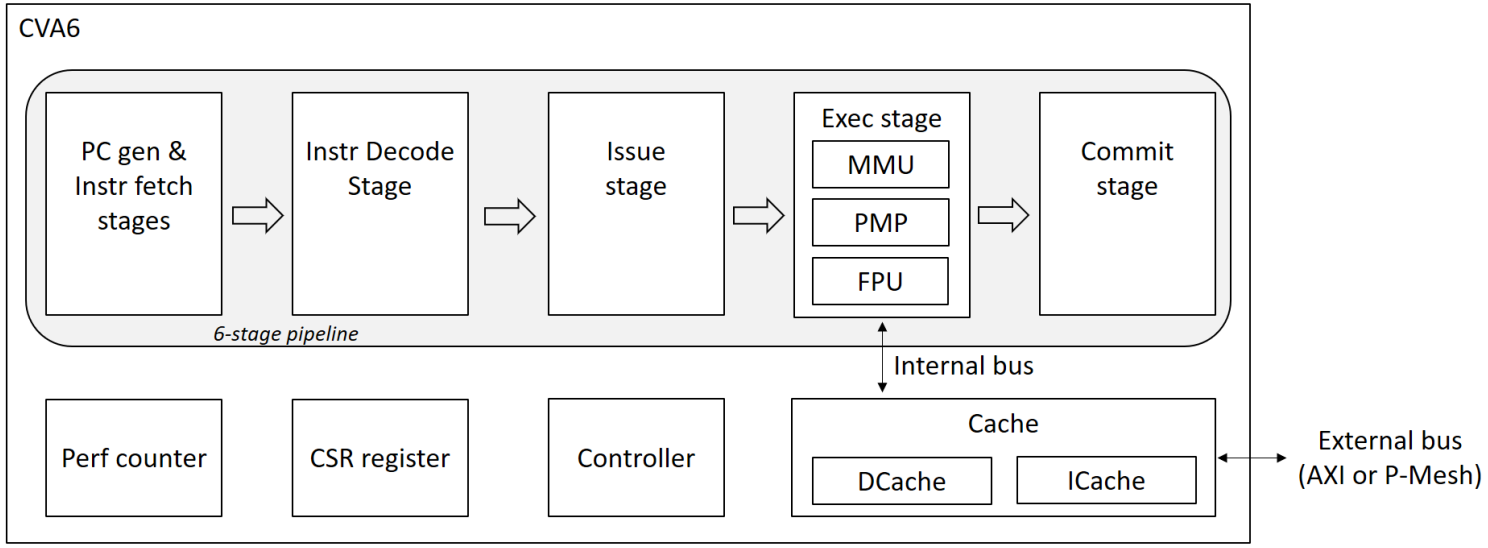

Electromagnetic waves

Countermeasures

Software	Hardware
• Provably secure • Flexible by nature • Widespread • Incur a higher overhead	• Less studied in the literature • Lack of formal proof • Lesser overhead **Needed for secure-by-design processors**

Context : hardware mitigation of side-channel attacks

Implement on-line instruction reordering by modifying the pipeline of a CVA6[1] core running with RISC-V instruction set architecture



Main subsystems of the CVA6 core

[1] https://github.com/openhwgroup/cva6

Context : hardware mitigation of side-channel attacks

Implement on-line instruction reordering by modifying the pipeline of a CVA6[1] core running with RISC-V instruction set architecture

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Not
add	ADD	R	0110011	0x0		d = rs1 + rs2	
sub	SUB	R	0110011	0x0		d = rs1 - rs2	
xor	XOR	R	0110011	0x4		d = rs1 ^ rs2	
or	OR	R	0110011	0x6		d = rs1 | rs2	
and	AND	R	0110011	0x7		d = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero

add rd, rs1, rs2
sub rd, rs1, rs2
xor rd, rs1, rs2

Extract of the RISC-V ISA

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	—
x3	gp	Global pointer	—
x4	tp	Thread pointer	Callee
x5	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

RISC-V registers

+ Program counter (pc) that points to the instruction to be executed next

[1] https://github.com/openhwgroup/cva6

Overview

Focus → new hardware, formally proven countermeasure against SC attacks, based on code obfuscation (i.e. instruction reordering)

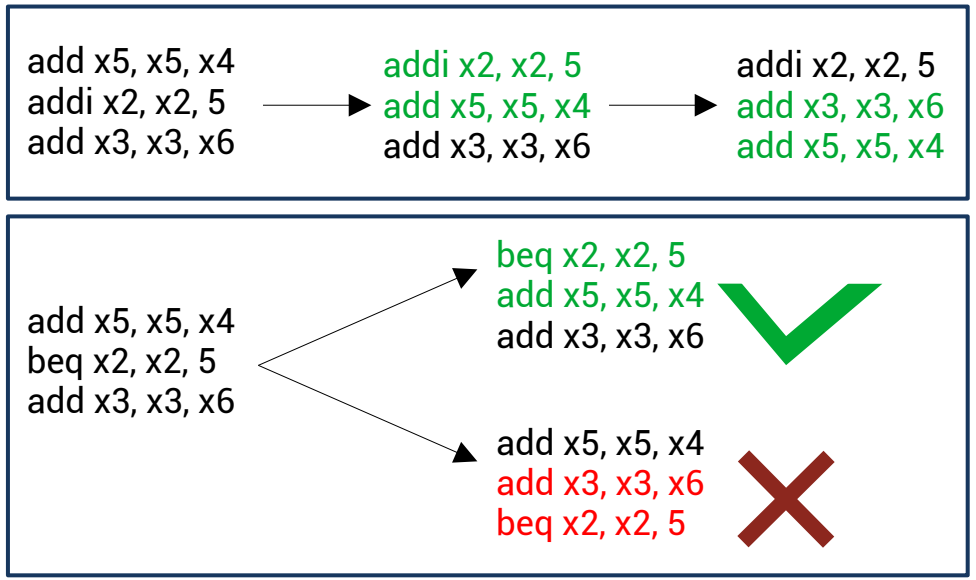
Contributions

1. Rules for reordering independent instructions
2. Formal proof of correctness and implementation guidelines
3. Achitecture modifications that enables reordering
4. Evaluation of diversity

1.
Reordering independent instructions

Reordering instructions

Based on previous works [1, 2], we choose to reorder independent instructions



We can swap two consecutive instructions *i* and *i'* if :

- *i* and *i'* are **data** and **control independent**
- *i'* is **not a load or store** (for peripheral management)
- *i* is **not a return**

We will prove that this reordering is sound

[1]: Couroussé, Damien, *et al.* “Runtime Code Polymorphism as a Protection Against Side Channel Attacks.” In Information Security Theory and Practice, 2016.
[2]: Belleville, Nicolas. “Compilation Pour l’application de Contre-Mesures Contre Les Attaques Par Canal Auxiliaire.” These de doctorat, 2019.

Incoming proof

Reordering independent instructions does not change the program behavior, i.e. executing a program P and $P_{reordered}$ leads to the same program state.

Definitions

The **state s** of a program **P** is a cuple (p, σ) where $p \in \{1, \ldots, |P|\}$ and σ is a valuation of all registers.

In *P* a **step** allows to pass from a state *s* to a state *s'* by executing an instruction (e.g. *i*), it is written $s \xrightarrow{i}_p s'$

A sequence of steps is called an **execution**: $s_0 \xrightarrow{i_0}_P s_1 \xrightarrow{i_1}_P s_2 \xrightarrow{i_2}_P \ldots$

Instructions *i=(o, in, out)* are classified into three distinct sets:

- Affectation$(\mathcal{A})$: $pc \notin in$ and $pc \notin out$ → Arithmetic operations
- Branch$(\mathcal{B})$: $pc \notin in$ and $out \in \{pc\}$ → Branching instructions
- Call $(\mathcal{C})$: $pc \in in$ and $pc \in out$ → Jump instructions

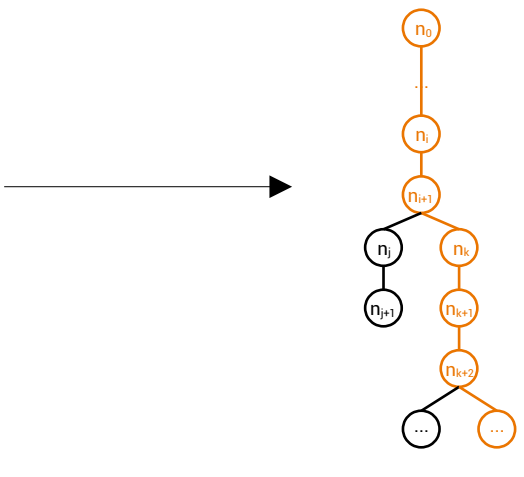
Executing an instruction changes the value of *out* as such:

- $if\ v \in out\ then\ \sigma'(v) = f(\{\sigma(v^*) | v^* \in in\}),\ \text{with}\ v \in Reg \cup pc$

Definitions

We define the set of all executions of a program as a ***Control-Flow Tree***, i.e. an unfolding of the control-flow graph.

```
000120e8 <cleanup_stdio>:
  lw   a1,4(a0)
  addi   sp,sp,-16
  sw   s0,8(sp)
  sw   ra,12(sp)
  addi   a5,gp,168
  mv   s0,a0
  beq   a1,a5,12108
  jal   12ff0
  lw   a1,8(s0)
  addi   a5,gp,272
  beq   a1,a5,1211c
 mv   a0,s0
  jal   12ff0
  lw   a1,12(s0)
  addi   a5,gp,376
  beq   a1,a5,1213c
  mv   a0,s0
  lw   s0,8(sp)
  lw   ra,12(sp)
  addi   sp,sp,16
  j   12ff0
  lw   ra,12(sp)
  lw   s0,8(sp)
  addi   sp,sp,16
  ret
```



Example of a Function from AES Code, and its Associated Control-Flow Tree

$$T = (n_0, N, A, \ell)$$

(n_0, N, A) is a directed tree and

$\ell : n_o \to Inst$ is a labelling of the nodes

An execution is the labeling of a path:

$$\pi = (n_0, n_1)(n_1, n_2)\dots(n_{k-1}, n_k)$$

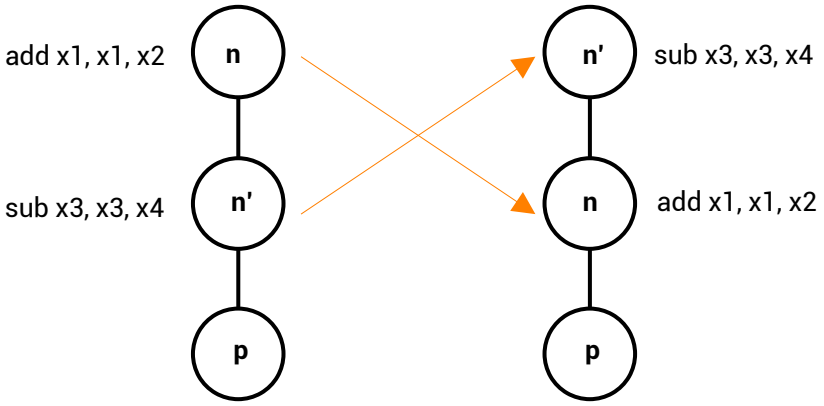
that is, the sequence:

$$\ell(\pi) = \ell(n_0)\ell(n_1)\dots\ell(n_k)$$

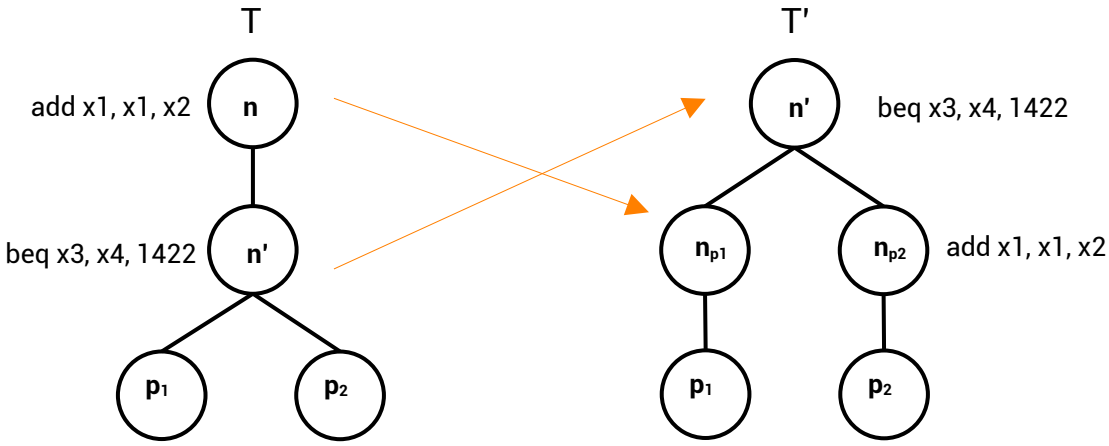
e.g.

```
lw   a1,4(a0)
addi   sp,sp,-16
sw   s0,8(sp)
sw   ra,12(sp)
addi   a5,gp,168
mv   s0,a0
beq   a1,a5,12108
jal   12ff0
...
```

Reordering Instructions



Reordering two $(\mathcal{A})$ instructions



Reordering two instructions with $\ell(n') \in (\mathcal{B})$

A one to one correspondence can be expressed between executions in *T* and *T'*, i.e.

if $\pi = \rho_{pre}(n, n')(n', p_1)\rho_{post}$ and $\pi' = \rho_{pre}(n', n_{p_1})(n_{p_1}, p_1)\rho_{post}$, then $\ell(\pi)$ and $\ell(\pi')$ are matching executions.

2.

Proof of Correctness

Reordering $(\mathcal{A})$ and $(\mathcal{C})$ independent instructions

Data independence Two instructions $i_A = (o_A, in_A, out_A)$ and $i_B = (o_B, in_B, out_B)$ are data-independent if:

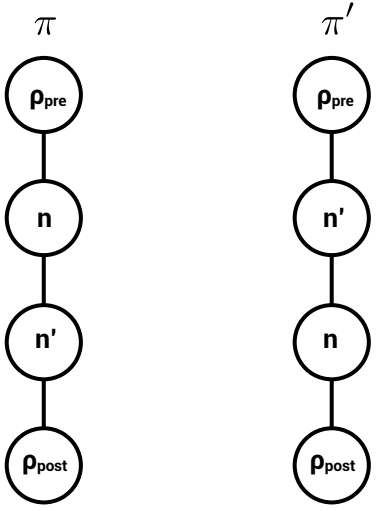
$$[in_A \cap out_B] \cup [out_A \cap in_B] \cup [out_A \cap out_B] = \varnothing$$

```
add x1, x2, x2
sub x3, x4, x4
```

If

- T is a control-flow tree
- T' is the matching tree with respect to (n,n')
- $\ell(n), \ell(n') \in \mathcal{A} \cup \mathcal{C}$ are data-independent

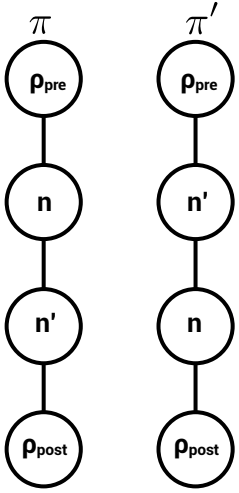
Theorem 1. $\ell'(\pi')$ the matching execution of $\ell(\pi)$ is possible from s_0, and reaches $s' = (p', \sigma')$ from s_0, with $\sigma' = \sigma$.



Reordering $(\mathcal{A})$ and $(\mathcal{C})$ independent instructions

$\ell(n) =$ add x1, x2, x2

$\ell(n') =$ sub x3, x4, x4



1. Execute $\ell(n)$ from s_{pre}
2. Execute $\ell(n')$ from s_n

Reg	Val
x0	x0
x1	x2+x2
x2	x2
x3	x4-x4
x4	x4
x5	x5
x6	x6
x7	x7
x8	x8
x9	x9

$$\sigma_{nn'} \equiv \sigma_{n'n}$$

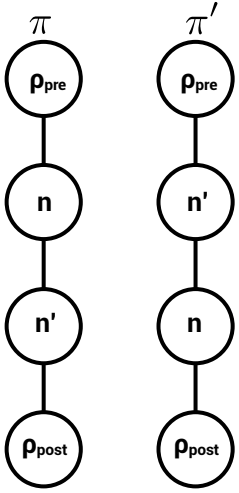
1. Execute $\ell(n')$ from s_{pre}
2. Execute $\ell(n)$ from $s_{n'}$

Reg	Val
x0	x0
x1	x2+x2
x2	x2
x3	x4-x4
x4	x4
x5	x5
x6	x6
x7	x7
x8	x8
x9	x9

Reordering $(\mathcal{A})$ and $(\mathcal{C})$ dependent instructions

$\ell(n) =$ add x3, x2, x2

$\ell(n') =$ sub x3, x4, x4



1. Execute $\ell(n)$ from s_{pre}
2. Execute $\ell(n')$ from s_n

Reg	Val
x0	x0
x1	x1
x2	x2
x3	x4-x4
x4	x4
x5	x5
x6	x6
x7	x7
x8	x8
x9	x9

$$\sigma_{nn'} \neq \sigma_{n'n}$$

1. Execute $\ell(n')$ from s_{pre}
2. Execute $\ell(n)$ from $s_{n'}$

Reg	Val
x0	x0
x1	x1
x2	x2
x3	x2+x2
x4	x4
x5	x5
x6	x6
x7	x7
x8	x8
x9	x9

Reordering $(\mathcal{A})$ and $(\mathcal{C})$ independent instructions

Proof Let π be a maximal path of T and let s_0 be a state such that $\ell(\pi)$ is possible from s_0. Let π' be the matching path of π in T' with respect to (n,n').

1. (n, n') is not part of π. In this case $\pi' = \pi$, and $\ell'(\pi') = \ell(\pi)$.
2. (n, n') is part of π. In this case, $\ell(\pi) = \rho_{pre}\ell(n)\ell(n')\rho_{post}$ exists and $\ell'(\pi') = \rho_{pre}\ell(n')\ell(n)\rho_{post}$.
   Then, given $s_{pre} \overset{\ell(n)}{\rightarrow}_P s_n \overset{\ell(n')}{\rightarrow}_P s_{nn'}$ and $s_{pre} \overset{\ell(n')}{\rightarrow}_P s'_n \overset{\ell(n)}{\rightarrow}_P s_{n'n}$, we must prove that $\sigma_{nn'} = \sigma_{n'n}$.
   (a) $\ell(n')\ell(n)$ is possible from s_{pre}
   (b) $\sigma_{nn'} = \sigma_{n'n}$ (previous slides)

From points (1) and (2) we get that Theorem 1 is correct.

By induction, Corollary 1 is a consequence of Theorem 1.
→ Theorem 1 is valid when reordering *k* independent instructions

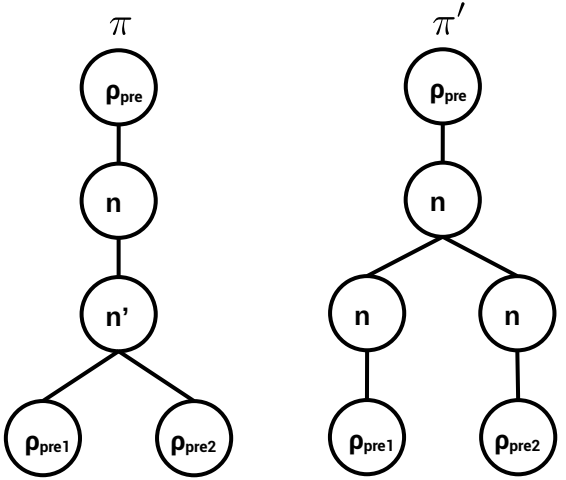
Reordering $(\mathcal{B})$ Independent Instructions

Control independence Two instructions i_A and i_B are control-independent if $s \xrightarrow{i_A}_P s'$ (with $s' = (p', \sigma')$) and for every initial states s_0:

$$P(p') = i_B \oplus P(p') \neq i_B$$

If

- T is a control-flow tree
- T' is the matching tree with respect to (n,n')
- $\ell(n) \in \mathcal{A} \cup \mathcal{C}$, $\ell(n') \in \mathcal{B}$ are data and control-independent



Theorem 2. $\ell'(\pi')$ the matching execution of $\ell(\pi)$ is possible from s_0, and reaches $s' = (p', \sigma')$ from s_0, with $\sigma' = \sigma$.

Reordering $(\mathcal{B})$ Independent Instructions

The only difference in the proof is we must show that branching evaluation will be the same, i.e. that instruction *n* doesn't change $in_{n'}$

$$\ell(n) = \text{add x1, x2, x2}$$
$$\ell(n') = \text{beq x3, x4, 80050}$$

As $\ell(n)$ and $\ell(n')$ are data-independent, $in_{n'}$ is disjoint from out_n, meaning that $\forall v \in in_{n'}, \sigma_{pre}(v) = \sigma_n(v)$.

Then, Theorem 2 is valid and Corollary 2 allows to reorder multiple independent $\mathcal{A}$ and $\mathcal{C}$ instructions over a branching.

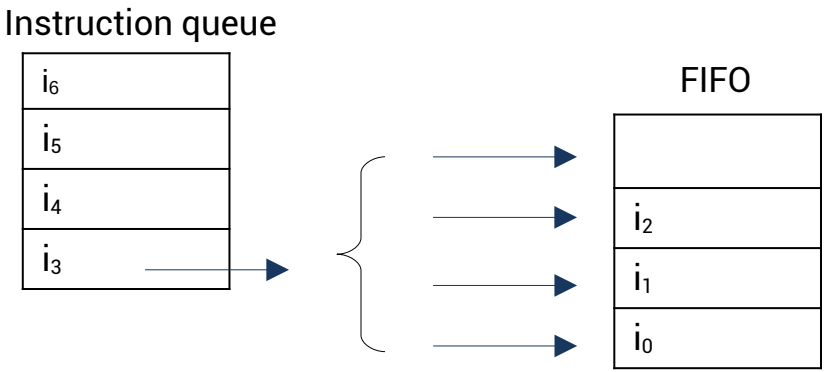
3.

Architecture model for on-line obfuscation

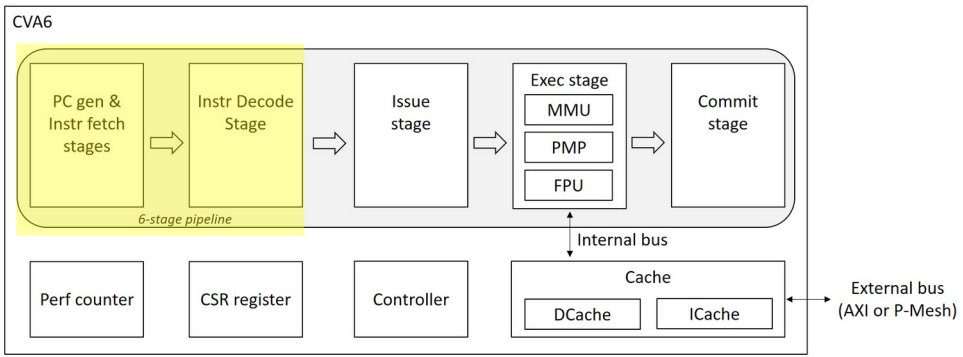
→ **Simple case**

Architecture model for on-line obfuscation

Inspired by the work of *Couroussé et al.* [1], we employ a *random insertion FIFO* to reorder instructions.



1) Use our rules to identify insertion slots
2) Insert randomly among possible slots



Architecture model for on-line obfuscation

If we reorder only $(\mathcal{A})$ instructions, the behavior of the processor follows Theorem 1.

Theorem 3 Given a control flow tree T of a program P, we denote π a path of T such that $\ell(\pi)$ exists from s_0.
Executing P from s_0 with our processor, and reordering only $\mathcal{A}$ instructions leads to a sequence $\ell'(\pi')$ that exists in T' and matches $\ell(\pi)$.

The processor follows Theorem 1

Proof If an instruction from $\mathcal{A}$ is fetched and reordered in the FIFO the processor may reorder an instruction in π with m independent and consecutive instructions.

- If $m = 1$, we reorder two instructions $\ell(n)$ and $\ell(n')$. According to Theorem 1, the matching execution $\ell'(\pi')$ reaches the same state.
- If $m > 1$, Corollary 1 states that the execution $\ell_k(\pi_k)$ resulting from several reordering matches $\ell(\pi)$.

In both cases, the semantic is preserved.

3.

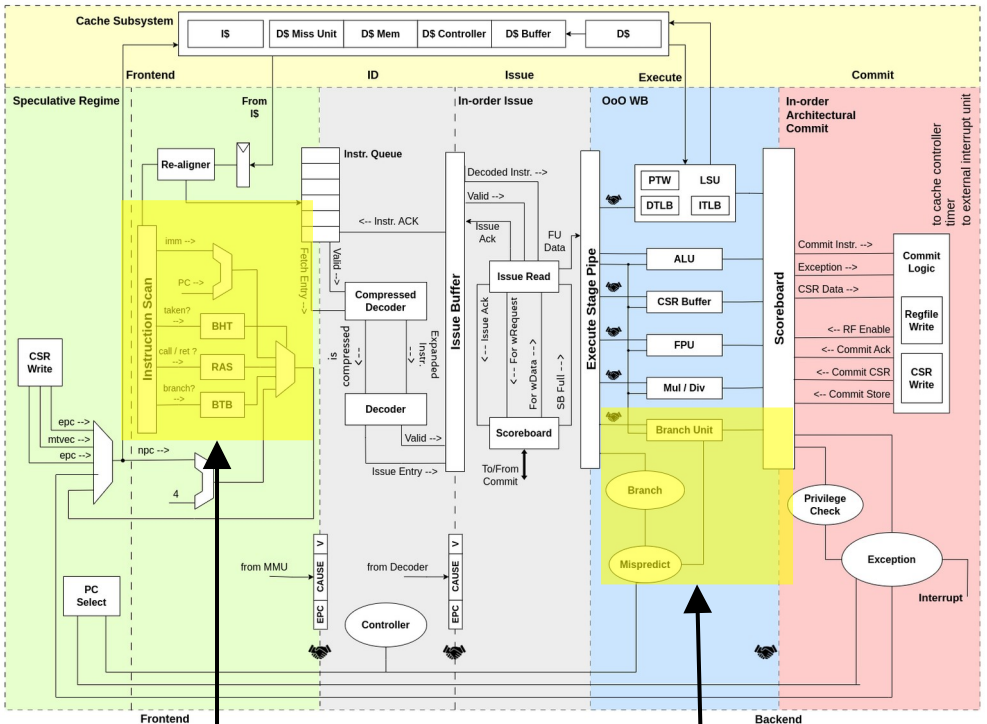
Architecture model for on-line obfuscation

→ **Simple case**

→ **Less simple case**

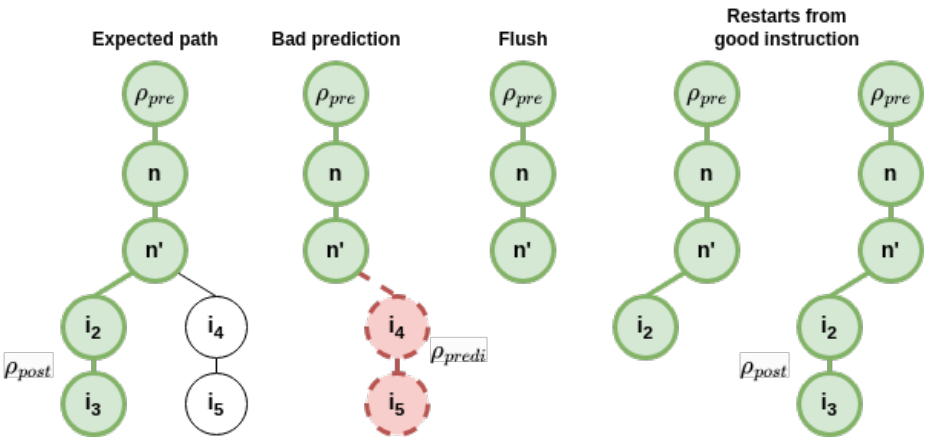
Architecture model for on-line obfuscation

Modern processors use **branch prediction**



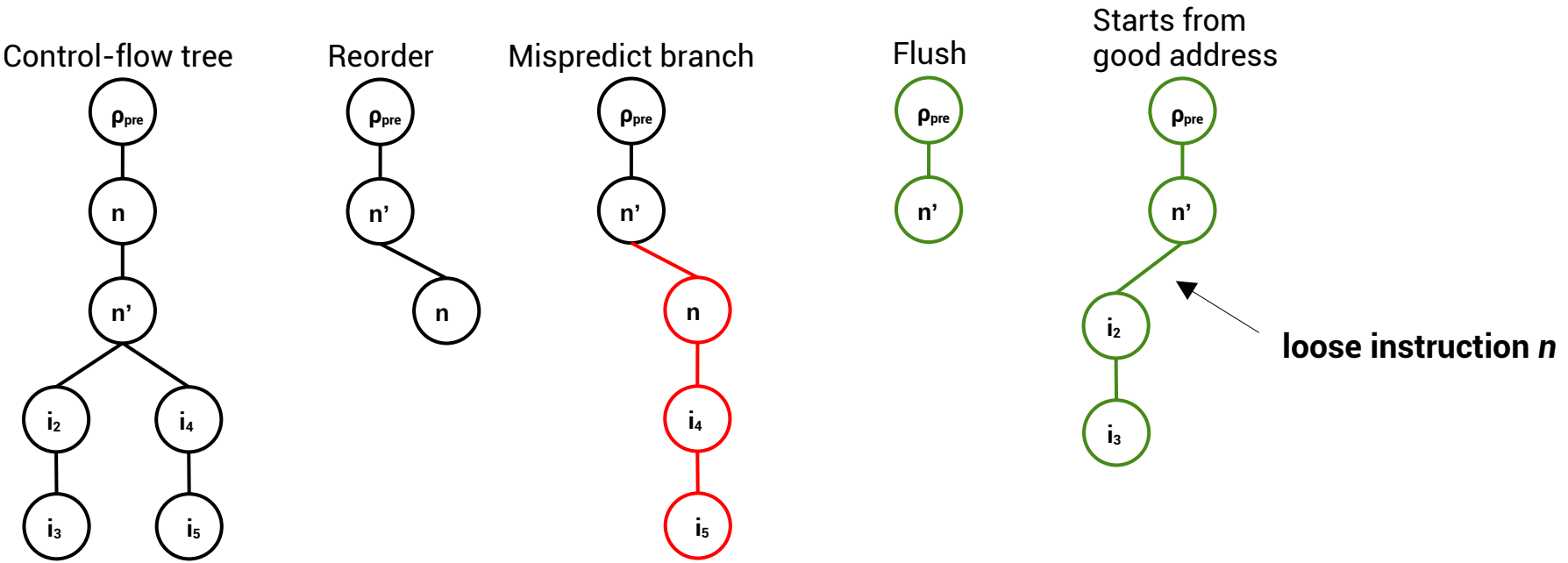
Branch prediction done here

And resolved here



Architecture model for on-line obfuscation

Branch prediction is a threat to reordering correctness.



We need to save reordered instructions in case of flush

Architecture model for on-line obfuscation

New rule: save and label instructions that are reordered after a branch/jump

Pipeline					ρ_0
Label					none

Pipeline				ρ_0	b_1
Label				1	1

Insert/reorder b_1 and actualize imbrication counter

Pipeline			ρ_0	ρ_2	b_1
Label			1	none	1

Insert/reorder ρ_2

Pipeline				ρ_0	ρ_2
Label				0	none

Commit b_1 → Decrement counters

Pipeline					ρ_0
Label					0

Bad prediction = flush if label ≠ 0

Architecture model for on-line obfuscation

New rule: save and label instructions that are reordered after a branch/jump

Pipeline					ρ_0
Label					none

Pipeline				ρ_0	b_1
Label				1	1

Insert/reorder b_1 and actualize imbrication counter

Pipeline			ρ_0	ρ_2	b_1
Label			1	none	1

Insert/reorder ρ_2

Pipeline	ρ_4	ρ_0	ρ_2	b_3	b_1
Label	none	1	2	2	1

Insert/reorder b_3 and actualize imbrication counter
Insert ρ_4

Pipeline		ρ_4	ρ_0	ρ_2	b_3
Label		none	0	1	1

Commit b_1
Decrement counters

Pipeline					ρ_0
Label					0

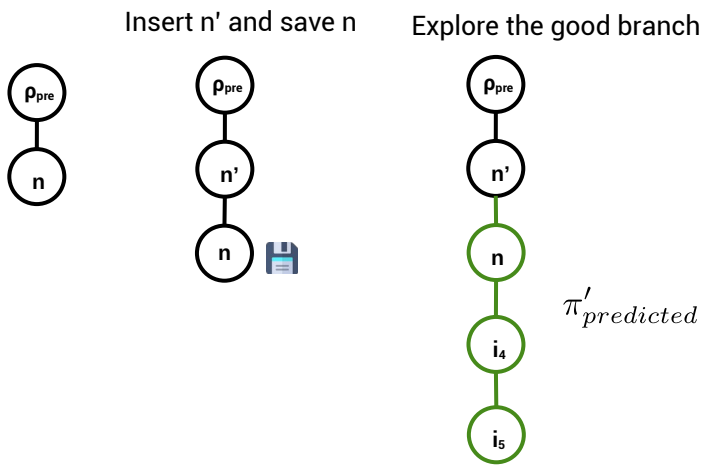
Bad prediction = flush if label ≠ 0

Architecture model for on-line obfuscation

Theorem 4 Executing P with our processor and reordering $\ell(n)$ and $\ell(n')$ leads to a sequence $\ell'(\pi')$ that exists in T' and matches $\ell(\pi)$.

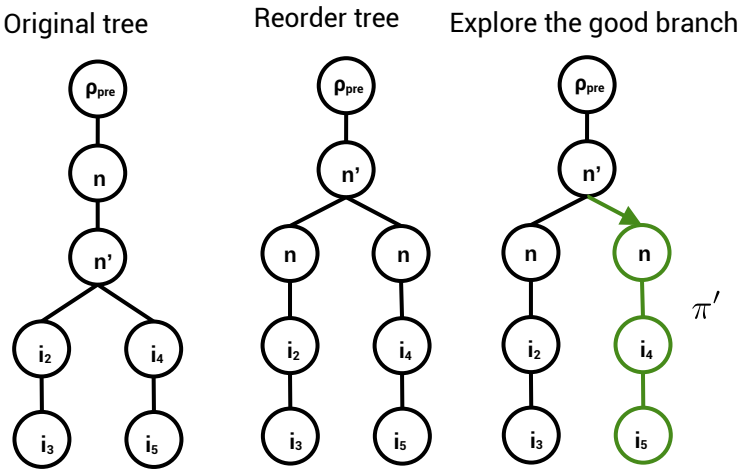
1. Case for good prediction

FIFO



$$\ell'(\pi'_{predicted}) = \rho_{pre}\ell(n')\ell(n)\ell(i_2)\ell(i_3).$$

Control-flow tree



$$\ell(\pi') = \rho_{pre}\ell(n')\ell(n)\ell(i_2)\ell(i_3).$$

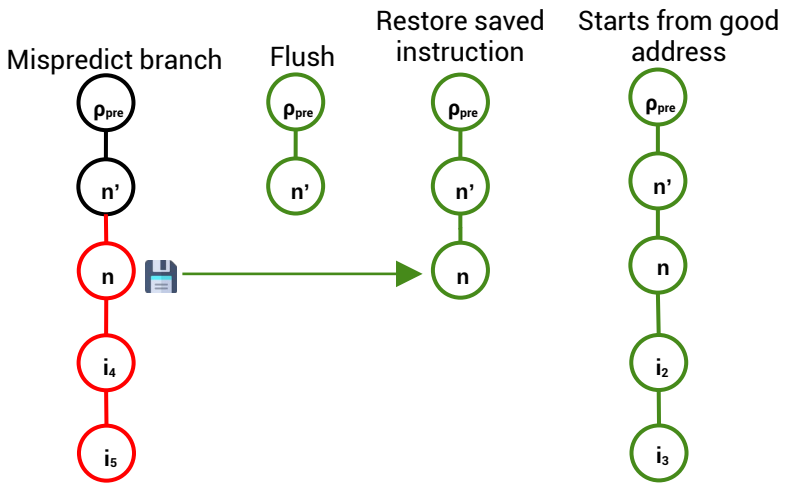
According to Theorem 2, $\ell(\pi') \equiv \ell'(\pi'_{predicted}) \equiv \ell(\pi)$

Architecture model for on-line obfuscation

Theorem 4 Executing P with our processor and reordering $\ell(n)$ and $\ell(n')$ leads to a sequence $\ell'(\pi')$ that exists in T' and matches $\ell(\pi)$.

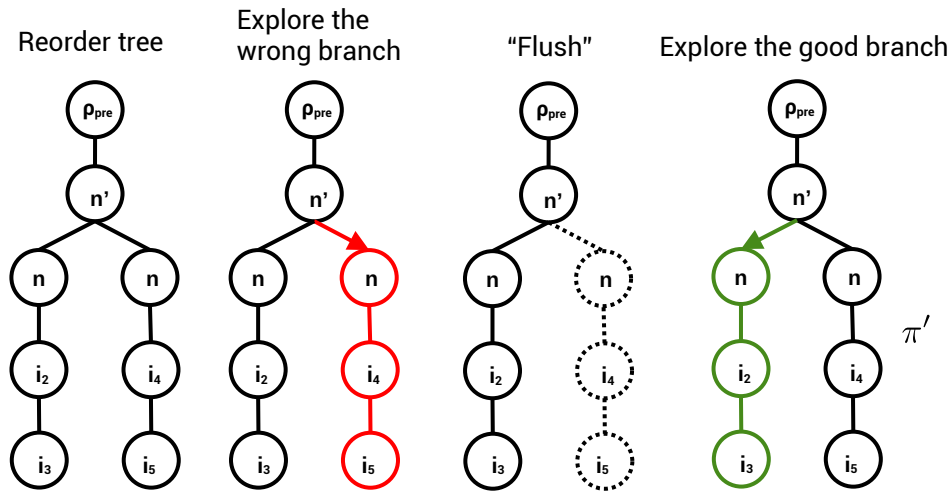
2. Case for misprediction

FIFO



$$\ell(\pi_{flush\_and\_restore}) = \rho_{pre}\ell(n')\ell(n)\ell(i_2)\ell(i_3).$$

Control-flow tree



$$\ell(\pi') = \rho_{pre}\ell(n')\ell(n)\ell(i_2)\ell(i_3).$$

According to Theorem 2, $\ell(\pi') \equiv \ell(\pi_{flush\_and\_restore}) \equiv \ell(\pi)$

3.

Architecture model for on-line obfuscation

The FIFO allows to reorder independent instructions and to follow Theorem 1 and 2

Note on exceptions and interruptions:
1) They save the context
2) Execute trap handling procedure
3) Restore the context
→ Control flow is not modified, the proof holds

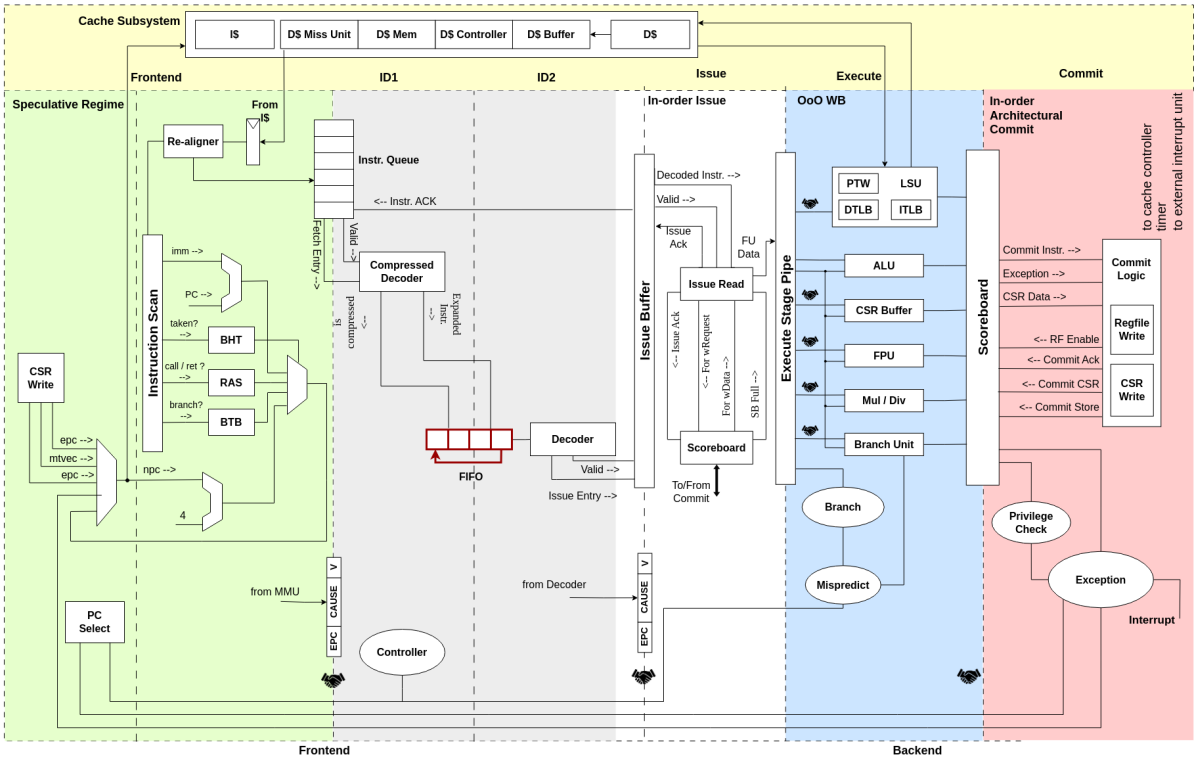
4.

A real architecture for on-line obfuscation

(On-going work)

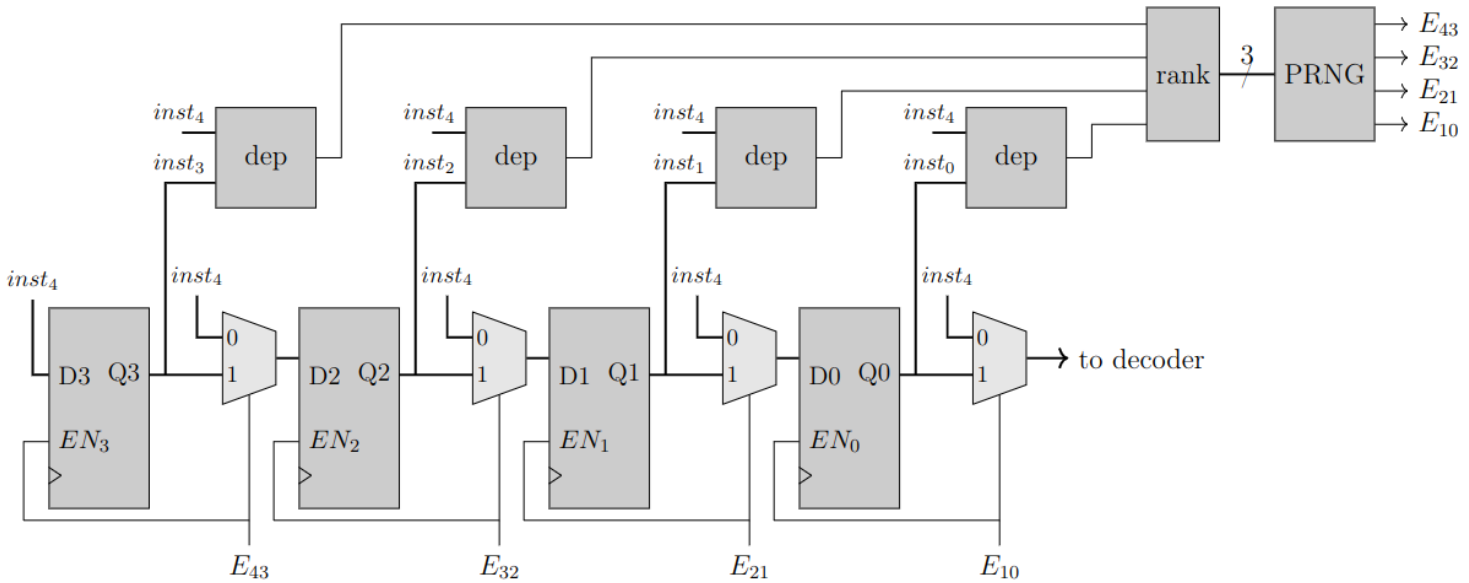
Implementation proposal

One idea is to add a stage in the CVA6 pipeline



Implementation proposal

One idea is to add a stage in the CVA6 pipeline



Hardware implementation of the FIFO with dependency management

Implementation proposal

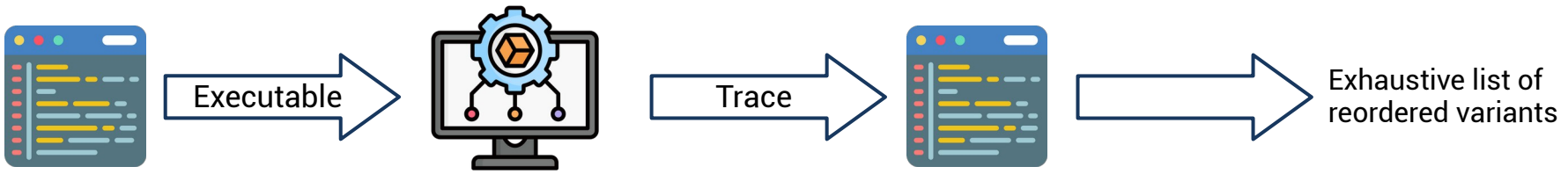
Branch prediction with global counter is yet to come.

5.

Evaluation of diversity

Evaluation of diversity

Before the real implementation, we can write a script to have insight on the diversity obtained.



Outputs :
- Number of variants
- Average distance with original

Lignes (code C)	Lignes (asm)	Buffer	# Traces	Temps (s)
126	207-226	2	35	5.80E-04
		3	90	1.70E-03
		4	90	1.80E-03
		5	90	1.80E-03
		10	90	2.00E-03
127	227-259	2	196	3.20E-03
		3	900	1.30E-02
		4	1764	3.00E-02
		5	3136	6.20E-02
		10	3136	8.70E-02
126+127	207-259	2	13720	3.30E-01
		3	162000	3.60E+01
		4	317520	1.70E+02
		5	564480	6.20E+02
		10	564480	9.60E+02

Conclusion

Conclusion

We proposed on-line code obfuscation to protect against side-channel attacks with:
- Formal proof of correctness
- Architecture model compliant with the proof
- Clues about a real implementation (*on-going*)
- Diversity measurements (*on-going*)

In the future:
- Implement architecture on a simulator
- Tests against side-channels (overhead vs security)
- Implement other obfuscation methods (noise injection, register reallocation, etc.)
- Tests against other attacks

Thanks for listening

SEC-V

Common meeting CT SED and GT AFSEC

30/01/2025

Théo Serru
theo.serru@univ-nantes.fr

Jean-Luc Béchennec
Loïg Jezequel

Mikaël Briday
Sébastien Faucou

LS2N Laboratoire des Sciences du Numérique de Nantes

IETR

THALES

anr

ANR-21-CE-39-0017