DE LA RECHERCHE À L'INDUSTRIE

FORMAL VERIFICATION OF A SOFTWARE COUNTERMEASURE AGAINST INSTRUCTION SKIP ATTACKS

Karine Heydemann[1], **Nicolas Moro**[1,2],
Emmanuelle Encrenaz[1], Bruno Robisson[2]

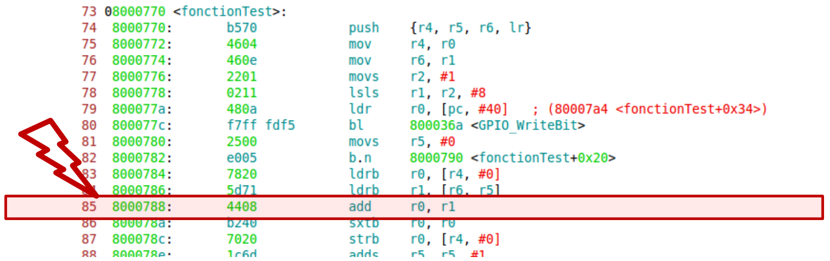
[1] **LIP6 - UPMC**
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie

[2] **CEA**
Commissariat à l'Energie Atomique et aux Energies Alternatives

PROOFS 2013 – AUGUST 24, SANTA BARBARA, USA

CONTEXT AND MOTIVATIONS

- Target: **Fault attacks on embedded programs**
- Fault model:
  assembly instruction skip



```
08000770 <fonctionTest>:
 8000770:       b570            push    {r4, r5, r6, lr}
 8000772:       4604            mov     r4, r0
 8000774:       460e            mov     r6, r1
 8000776:       2201            movs    r2, #1
 8000778:       0211            lsls    r1, r2, #8
 800077a:       480a            ldr     r0, [pc, #40]   ; (80007a4 <fonctionTest+0x34>)
 800077c:       f7ff fdf5       bl      800036a <GPIO_WriteBit>
 8000780:       2500            movs    r5, #0
 8000782:       e005            b.n     8000790 <fonctionTest+0x20>
 8000784:       7820            ldrb    r0, [r4, #0]
 8000786:       5d71            ldrb    r1, [r6, r5]
 8000788:       4408            add     r0, r1
 800078a:       b240            sxtb    r0, r0
 800078c:       7020            strb    r0, [r4, #0]
 800078e:       1c6d            adds    r5, r5, #1
```

- A large set of harmful attacks may be possible with such a fault model

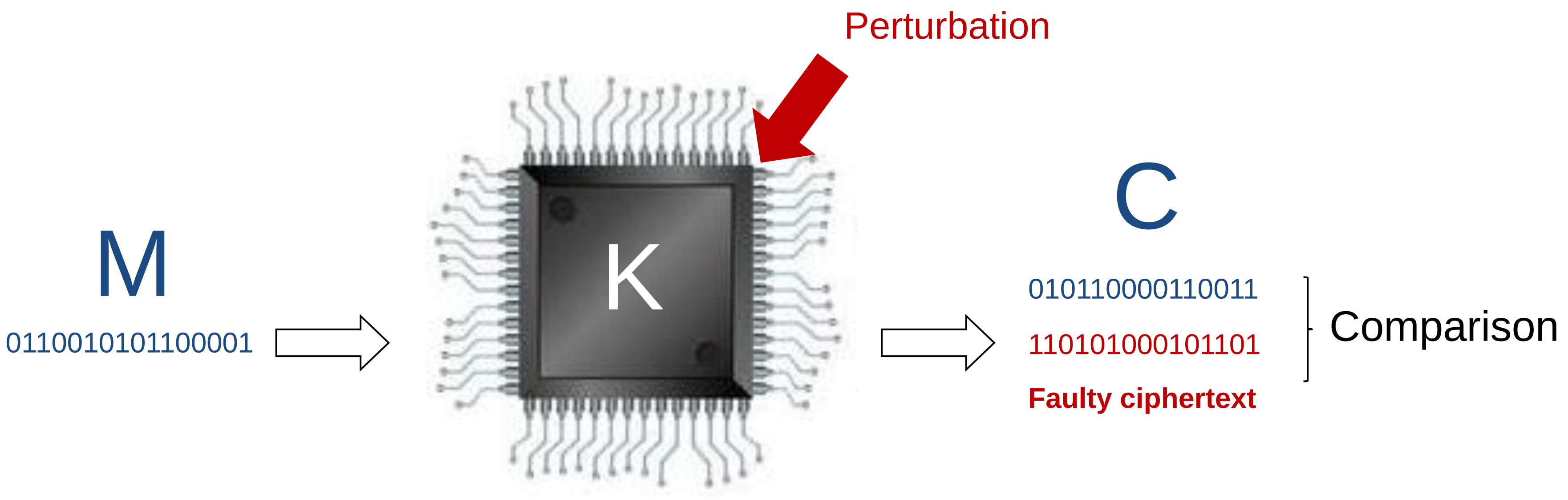
How can we **ensure a correct execution** of the embedded program with a **possible instruction skip fault**?

1 – Provide a **fault-tolerant replacement sequence for each instruction**

2 – Provide a **formal proof for this fault tolerance**

OUTLINE

FAULT INJECTION ATTACKS



- Modify the circuit's environment to **change its computation**
- Many physical ways to inject such faults into a circuit
- Every fault injection means has a specific **fault model**

INSTRUCTION SKIP FAULT MODEL

- We assume an attacker can **skip an assembly instruction**

- Observed for different architectures and injection means

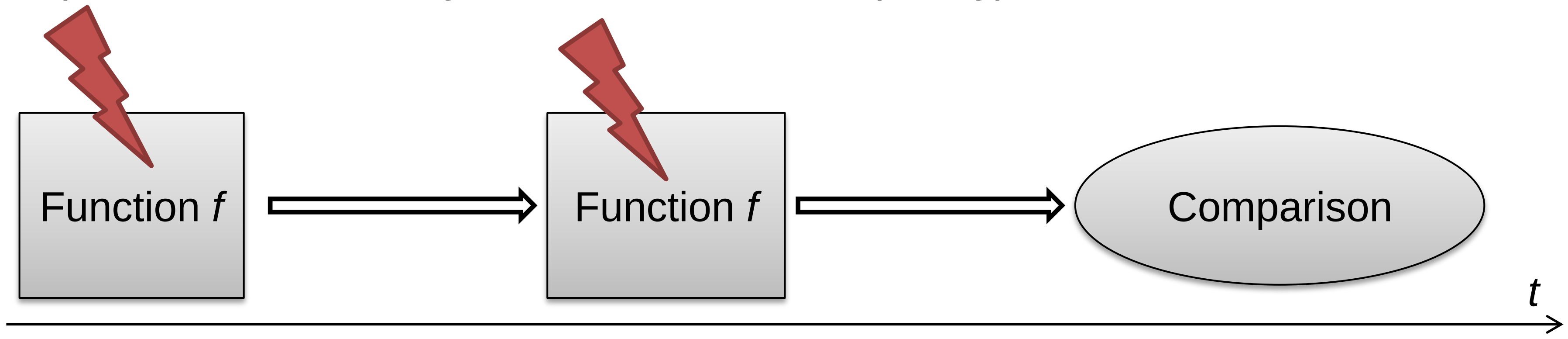
Fault injection means	Reference
Clock glitches	Balasch *et al.* (FDTC 2011)
Voltage underfeeding	Barenghi *et al.* (J. Syst. Soft. 2013)
Electromagnetic pulses	Dehbaoui *et al.* (FDTC 2012)
Laser	Trichina *et al.* (FDTC 2010)

- **In our own experiments**, instruction skips are specific cases of instruction replacements (FDTC 2013)

HOW TO DEAL WITH DOUBLE FAULTS ?

- Double faults are **practical** under some specific constraints

(a few tens of clock cycles, at a few KHz frequency)



- We assume performing a **double fault on two consecutive clock cycles** is significantly harder to do in practice

➔ Usual redundancy countermeasures need to be adapted

OUTLINE

COUNTERMEASURE SCHEME : APPROACH

- Propose a **fault-tolerant replacement scheme** for each assembly instruction of the whole instruction set

- Each encoding has its own **fault-tolerant replacement sequence**

ARM Thumb2 instruction set

-16/32 bit **RISC instruction set**

- **151 instructions**

- Each instruction has **up to 4 encodings**

(depends on the operands, the registers used, conditional execution, …)

THREE CLASSES OF INSTRUCTIONS

Class	Examples

Idempotent instructions	`mov r1,r8`
	`add r3,r1,r2`

Separable instructions	`add r1,r1,#1`
	`push {r4,r5,r6}`

Specific instructions	`bl <function>` IT blocks

Standard code

```
ADD     R1, R1, #1
CMP     R1, #9
B       <label>
```

Code with replacement sequences

```
ADD     R12, R1, #1
ADD     R12, R1, #1
MOV     R1, R12
MOV     R1, R12
CMP     R1, #9
CMP     R1, #9
B       <label>
B       <label>
```

IDEMPOTENT INSTRUCTIONS

- Instructions that have the same effect if executed once or twice

➔ **Simple instruction duplication**

Instruction	Replacement sequence

`mov r1,r8`
 (copies r8 into r1)
	`mov r1,r8`
	 `mov r1,r8`

`ldr r1, [r8, r2]`
 (loads the value at the address r8+r2 into r1)
	`ldr r1, [r8, r2]`
	 `ldr r1, [r8, r2]`

`str r3, [r2, #10]`
 (stores r3 at the address r2+10)
	`str r3, [r2, #10]`
	 `str r3, [r2, #10]`

`add r3,r1,r2`
 (puts r1+r2 into r3)
	`add r3,r1,r2`
	 `add r3,r1,r2`

- Doubles the **code size**, doubles the **execution time**

SEPARABLE INSTRUCTIONS

- **Not idempotent, with a source register also destination**

… but can be replaced by a sequence of idempotent instructions

ADD R1, R1, #1

```
ADD     r12, r1, #1
ADD     r12, r1, #1
MOV     r1, r12
MOV     r1, r12
```

PUSH {r1, r2, r3, lr}
(equivalent to `STMDB sp!, {r1,r2,r3,lr}`)

```
STMDB   sp, {r1, r2, r3, lr}
STMDB   sp, {r1, r2, r3, lr}
SUB     r12, sp, #16
SUB     r12, sp, #16
MOV     sp, r12
MOV     sp, r12
```

- Variable **overhead cost in code size and performance**
- Need for an **available free register**

SPECIFIC INSTRUCTIONS

- Some instructions cannot be easily replaced by such a sequence
- Branch instructions can be duplicated, but not **subroutine calls**
  (otherwise those subroutines would be executed twice)

bl <function>

```
ADR     r12, <return_label>
ADR     r12, <return_label>
ADD     lr, r12, 1
ADD     lr, r12, 1
B       <function>
B       <function>
```

return_label:
        …

Puts the return address into r12

Updates the return pointer (LR)

Branches to the subroutine code

OTHER VERY SPECIFIC SITUATIONS

Instructions that read and write the flags

- A replacement sequence can be found

  if the flags are not alive
- Otherwise:
  - forbid the use of those instructions
  - use a **fault detection** sequence

```
mrs     r12 , APSR
mrs     r12 , APSR
adcs    r1 , r2 , r3
msr     APSR, r12
msr     APSR, r12
adcs    r1 , r2 , r3
```

IT blocks for conditional execution

- Convert IT blocks into **branch-based if/then/else structures**
- Then apply the individual countermeasure scheme
- A more tricky replacement is possible by keeping the IT structure

  but it can quickly become very costly

OVERVIEW OF THE COUNTERMEASURE

- A **fault-tolerant replacement sequence** for all the instructions
  (except for the ones that read and write the flags)

- Can be **directly applied as a transformation** to an assembly code

➔Can we prove the replacement sequences are **fault-tolerant** ?

➔Can we prove they are **equivalent to the initial instructions** ?

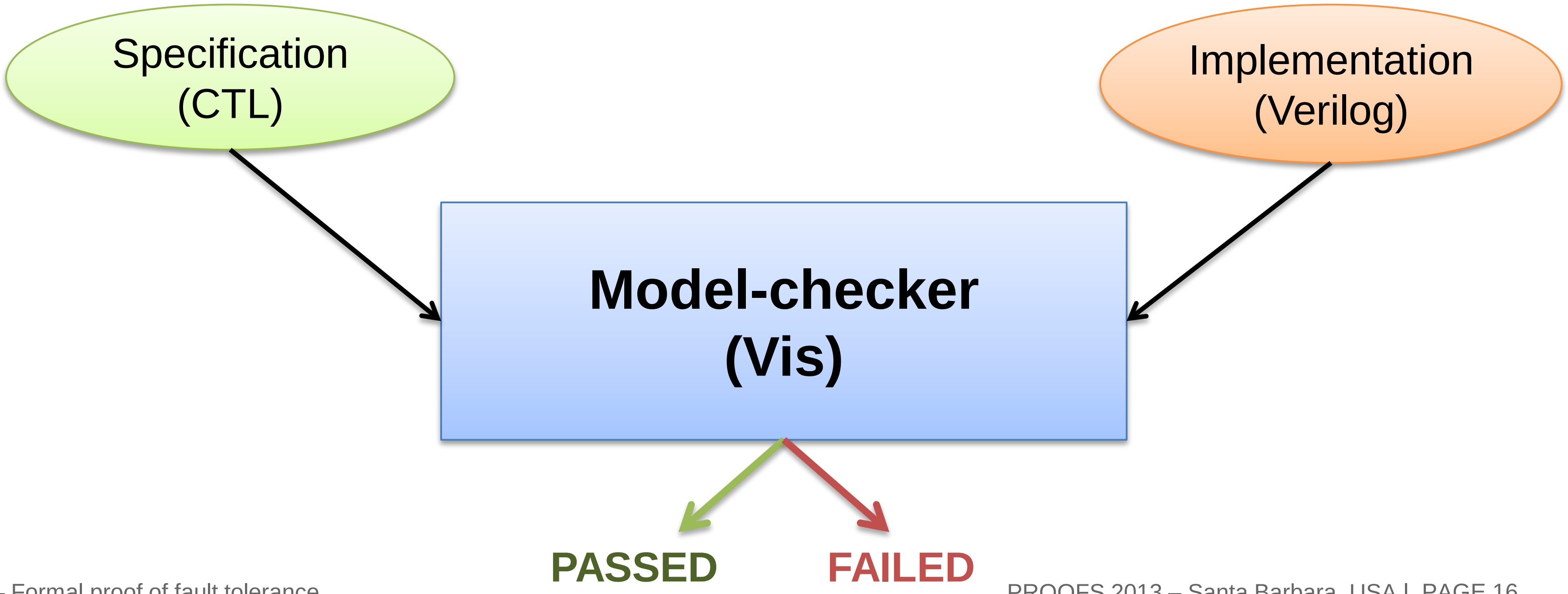
We use a **model-checking approach** for such a proof

OUTLINE

DE LA RECHERCHE À L'INDUSTRIE

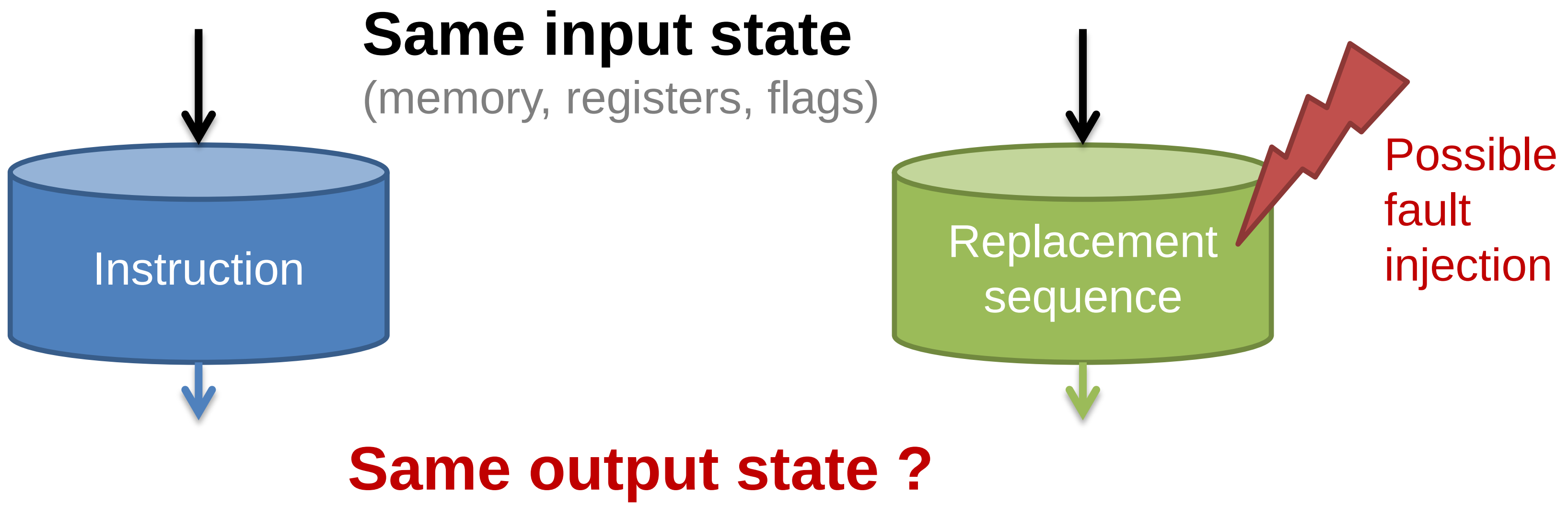
MODEL-CHECKING APPROACH

- Model-checking aims at **ensuring an implementation satisfies a specification**
- Specifications can be expressed with **temporal logic formulas**



MODEL-CHECKING APPROACH

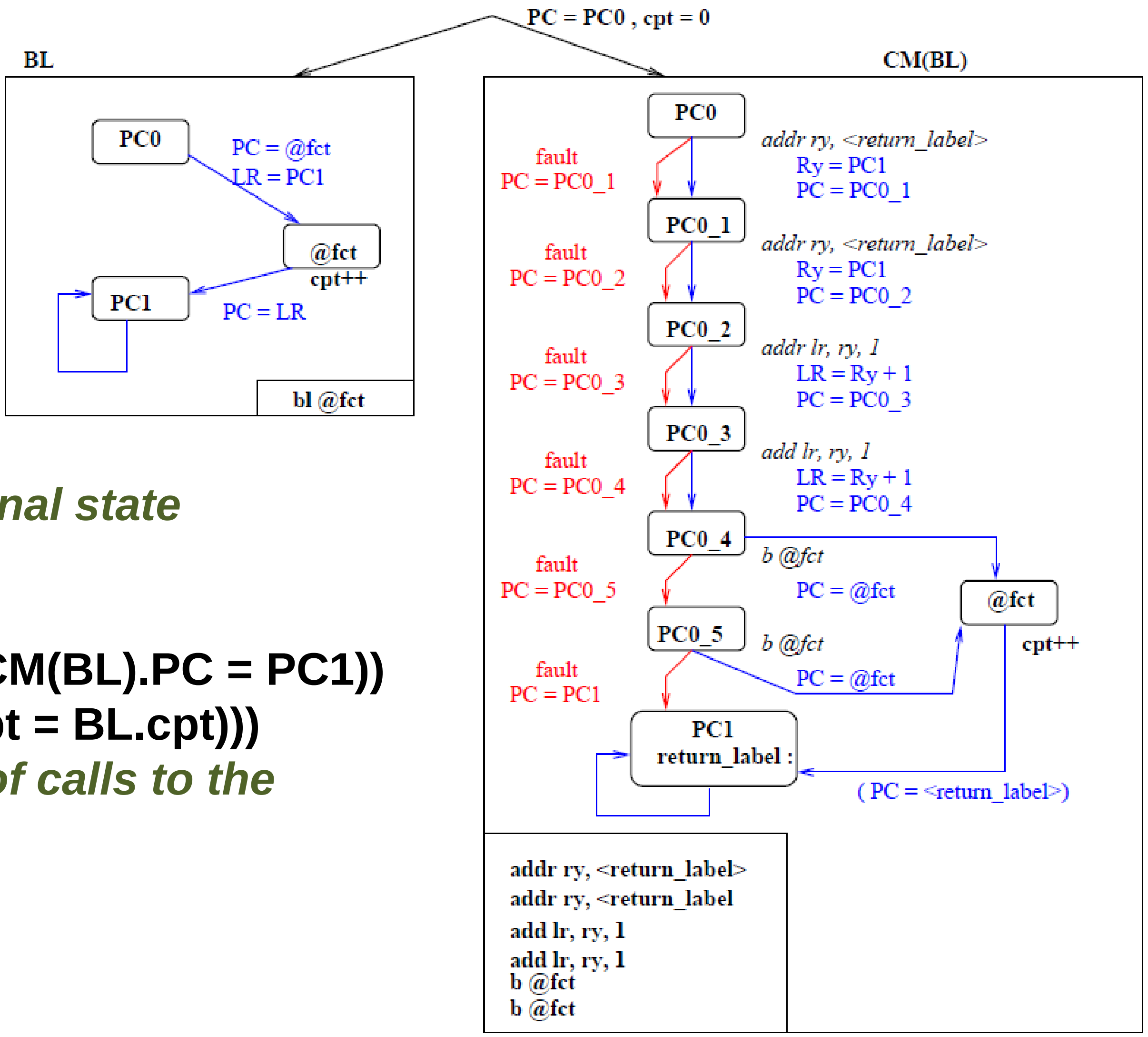
- Model-checking approach **at an instruction scale**
- Specific **construction of a state machine** with an instruction and its replacement sequence
- We need to prove that the **output state** of a replacement sequence is **equivalent to the output state of the initial instruction**



MODEL-CHECKING APPROACH

- Each instruction is a **function that maps a registers and memory configuration** to a new registers and memory configuration
- A **sequence of instructions** is modeled as a **transition system**
- States are **registers and memory configurations**
- Transitions mimic the **state transformations applied by the instructions**
- Each instructions has **specific properties that need to be proved**

PROOF SYSTEM FOR THE BL INSTRUCTION



P1 : AF(BL.PC = PC1
+ CM(BL).PC = PC1)

Any path finally goes to a final state

P2 : AG(((BL.PC = PC1) *(CM(BL).PC = PC1))
=> ((BL.cpt = 1) * CM(BL).cpt = BL.cpt)))

In a final state the number of calls to the function is equal to one

ADCS INSTRUCTION

```
mrs     r12 , APSR       ←  Saves the flags
mrs     r12 , APSR
adcs    r1 , r2 , r3
msr     APSR, r12        ←  Restores the flags
msr     APSR, r12
adcs    r1 , r2 , r3
```

- Flags are not equal if a fault targets the last ADCS instruction
- `LIGHT_RESULT` releases this constraint

```
# MC: formula passed --- AG(AF(adcs.pc=PC1))
# MC: formula passed --- AG(AF(cm(adcs).pc=PC1))
# MC: formula passed --- AG(((adcs.pc=PC1 * cm(adcs).pc=PC1) -> LIGHT_RESULT=1))
# MC: formula failed --- AG(((adcs.pc=PC1 * cm(adcs).pc=PC1) -> RESULT=1))
```

OUTLINE

APPLICATION TO AN AES IMPLEMENTATION

- Round keys calculated before each `AddRoundKey` operation
- Possible optimization: last two rounds with countermeasure

Implementation	Clock cycles	Code size
AES - without countermeasure	9595	490 bytes
AES - whole code with CM	20503 (+113.7%)	1480 bytes (+202%)
AES – last two rounds with CM	11374 (+18.6 %)	1874 bytes (+282.5%)

➔ **High overhead cost**,

but **comparable** to the cost brought by usual redundancy approaches

➔ Enables a fault tolerance at a cheaper cost compared to triplication

OUTLINE

CONCLUSION AND PERSPECTIVES

- **Fault-tolerant** countermeasure scheme
- Tolerant to multiple fault that do not target two consecutive instructions
- **Proof** of fault tolerance and equivalence to the initial instructions
- Adds a **reasonably good security level**, without any hardware countermeasure
- Cost comparable to usual **algorithm-level redundancy schemes**

Perspectives

- **Extend the fault model** to include faults on the data loads

- **Make a practical evaluation** of such a countermeasure scheme

THANK YOU FOR YOUR ATTENTION

Any questions ?