GT Méthodes Formelles pour la Sécurité
January 30th 2020

A Cross-Layer Security Approach: Combining Accurate Modelling of Hardware Faults with Static Software Analysis

Johan Laurent[1], Christophe Deleuze[1], Vincent Beroulle[1], Florian Pebay-Peyroula[2]

[1] Univ. Grenoble Alpes, Grenoble INP, LCIS
26000 Valence, France
firstname.lastname@lcis.grenoble-inp.fr

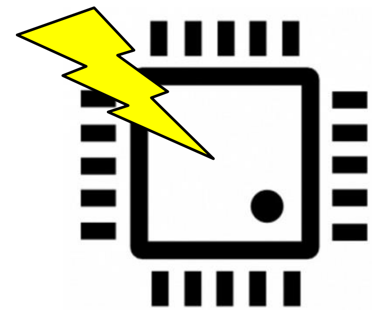
[2] Univ. Grenoble Alpes, CEA, LETI
38000 Grenoble, France
firstname.lastname@cea.fr

This work was funded thanks to the French national program 'programme d'Investissements d'Avenir, IRT Nanoelec' ANR-10-AIRT-05

Hardware Fault Attacks

- **Fault injection attacks: perturbing a circuit**
  - Power/clock glitches, heating, EM injection, **laser…**
- **Attacker's goals :**
  - Bypass security measures

    (authentication with a wrong PIN code)
  - Extract secret information from fault effects.



- **How to protect ?**
  - Hardware countermeasures (CM): duplication, error correcting codes, watchdog…
  - Software CM: duplication, control flow integrity…

UNIVERSITÉ Grenoble Alpes

Grenoble INP

Summary

- **I. Introduction**
- **II. Our approach**
  - Overview
  - Software fault injection
- **III. Case study**
- **IV. Discussion**
  - Invariant properties
  - Performances
  - False positives
- **V. Conclusion & perspectives**

I. Introduction

- **Software analyses are based on software fault models (defined by the Joint Interpretation Library for example [1])**
  - Instruction skip [2]
  - Control-flow corruption (test inversion, …) [3][4]
  - Register/memory corruptions [5][6]

- **Problem: there are hardware fault effects that are not modelled in typical software fault models [7]**

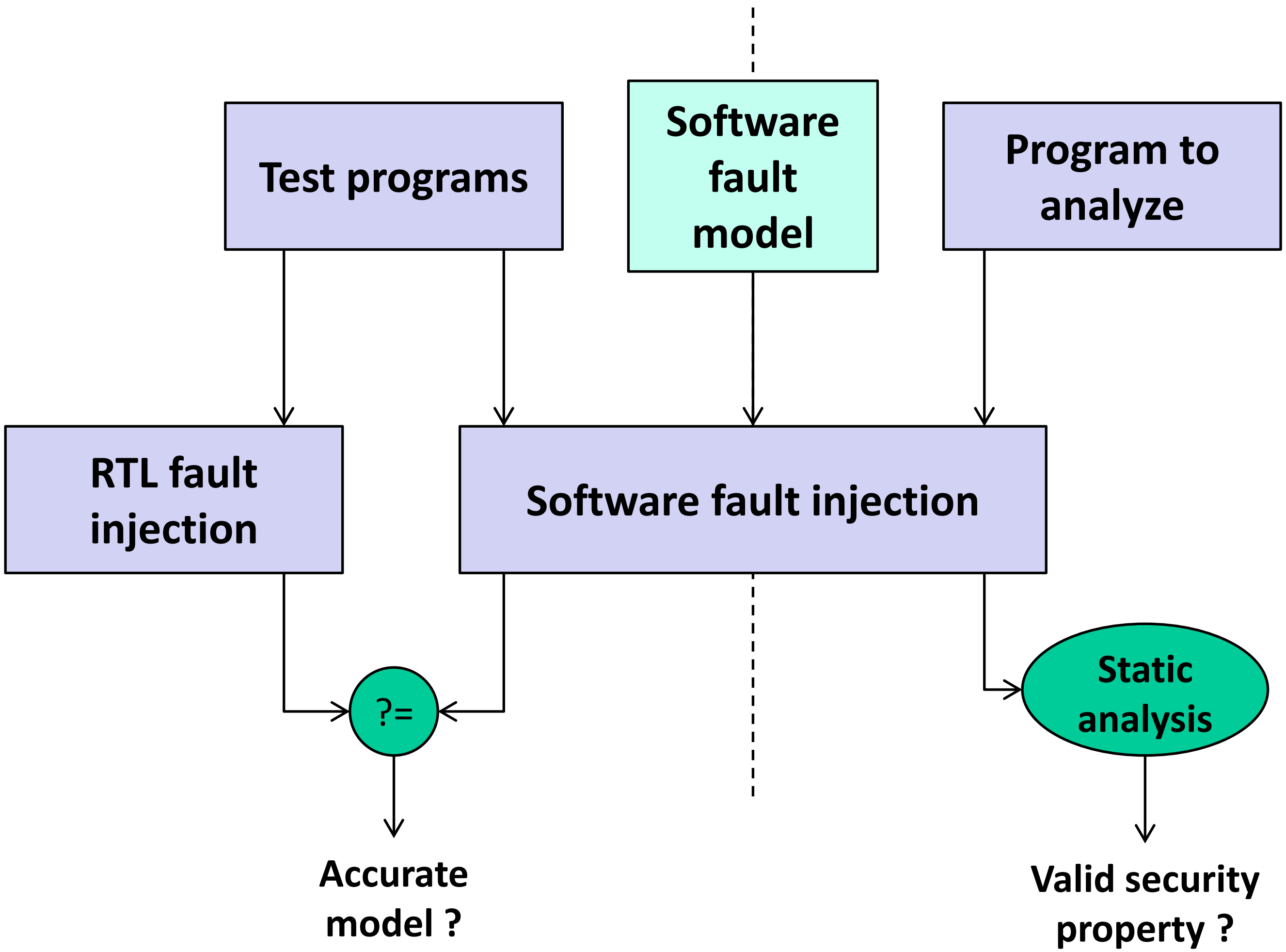
- **Effects obtained in simulation in a LowRISC v0.2 processor [8]:**
  - Replace an argument by the last computed value
  - Make an instruction “transient”
  - Set an architectural register to 0 or 1 during a branching instruction
  - Commit a speculated instruction
  - …

I. Introduction

- **How to model these effects ?**
- **How to perform efficient security analyses with these complex software fault models?**

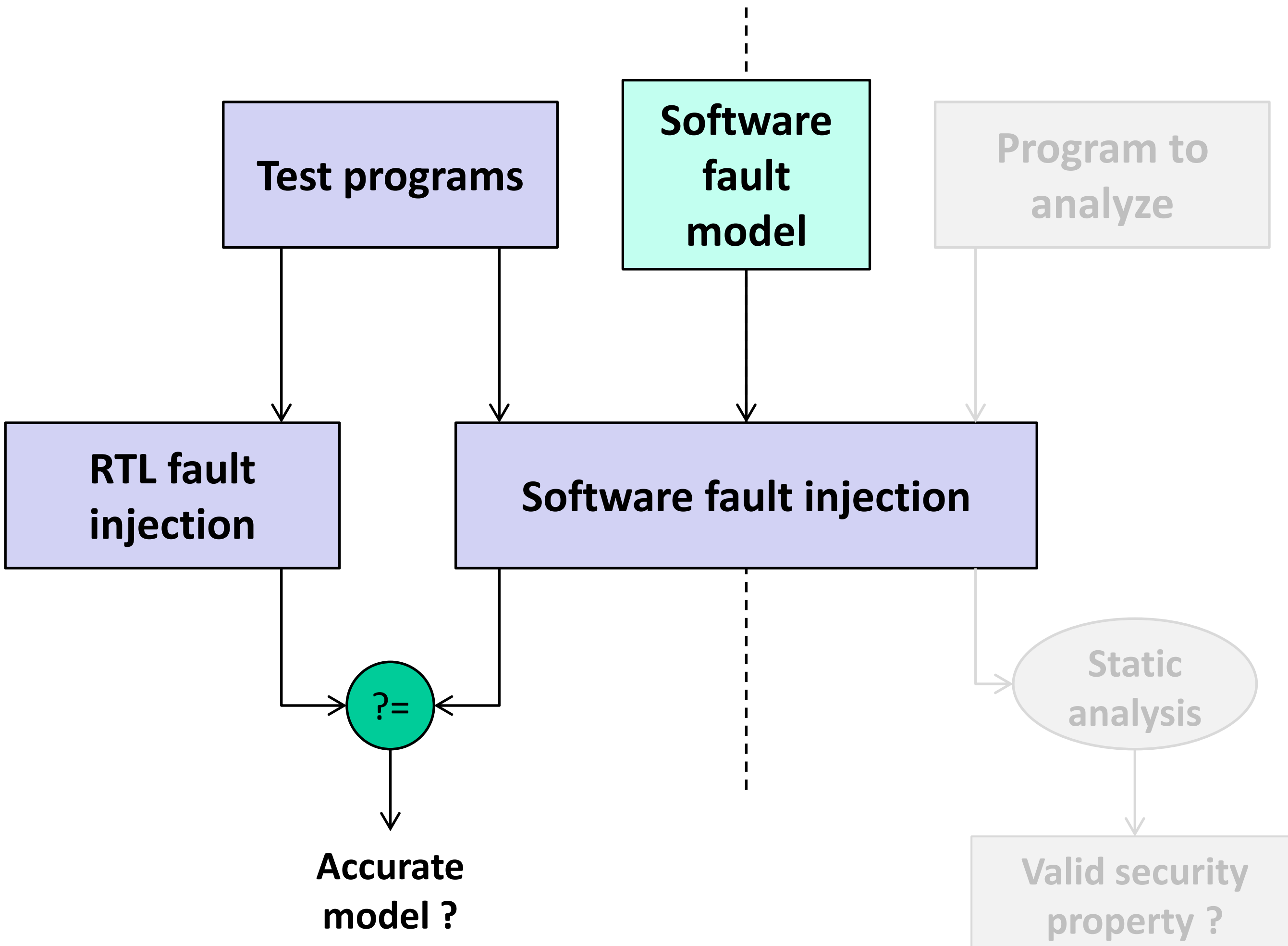
II. Approach

a. Overview



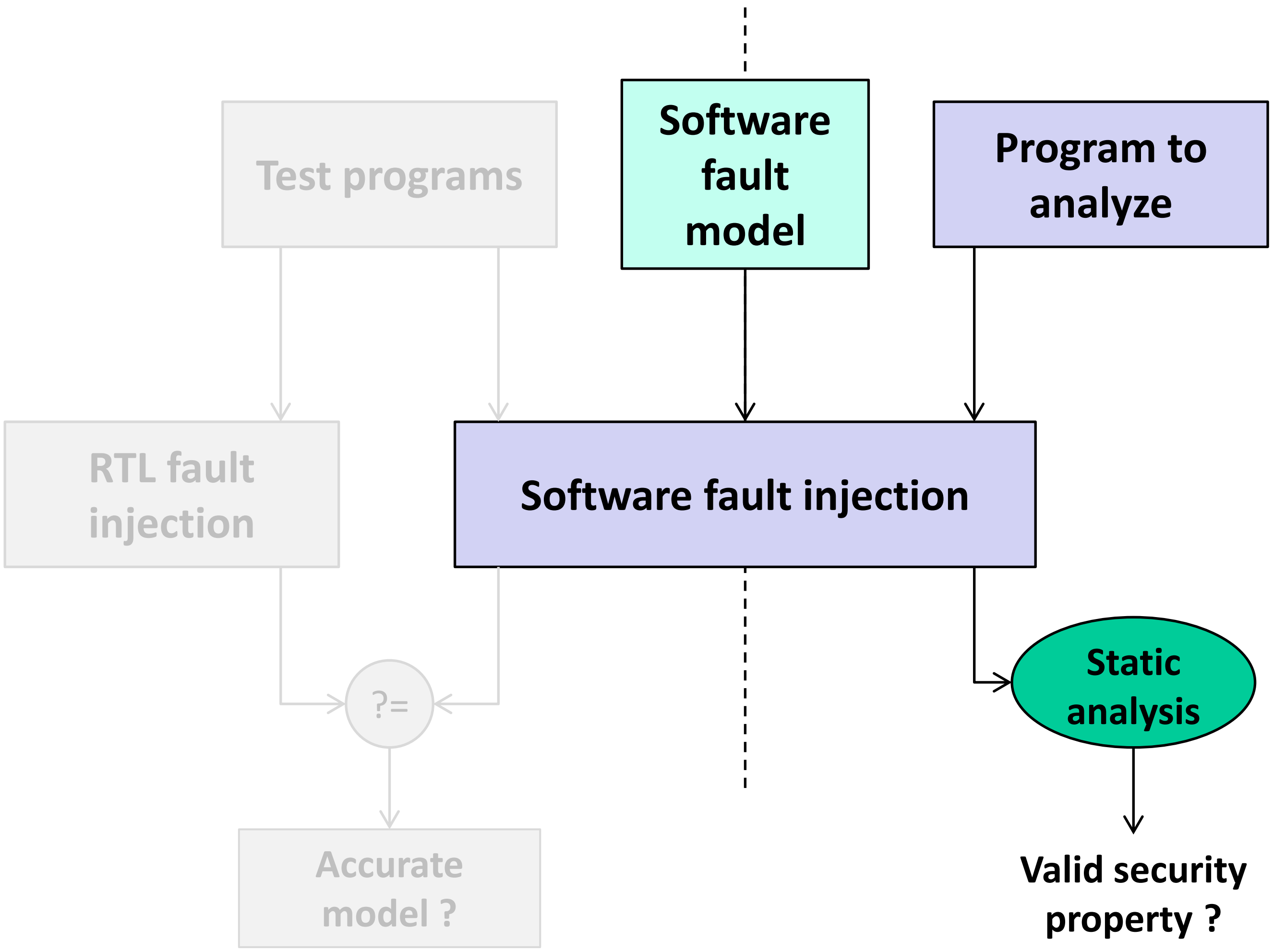
II. Approach

a. Overview – Fault Modelling



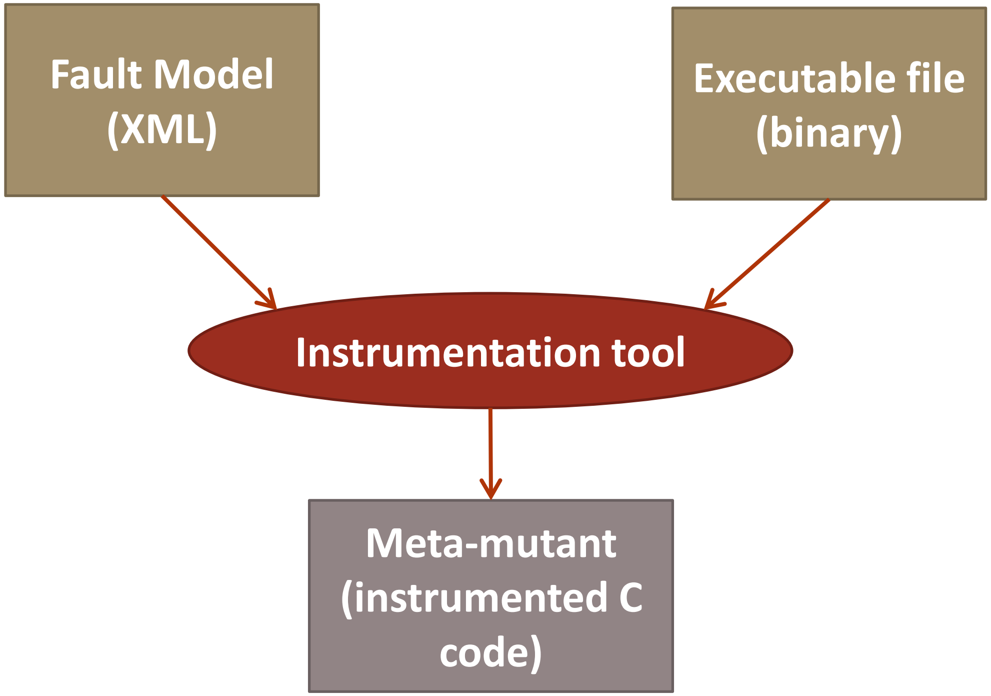
II. Approach

a. Overview – Security analysis



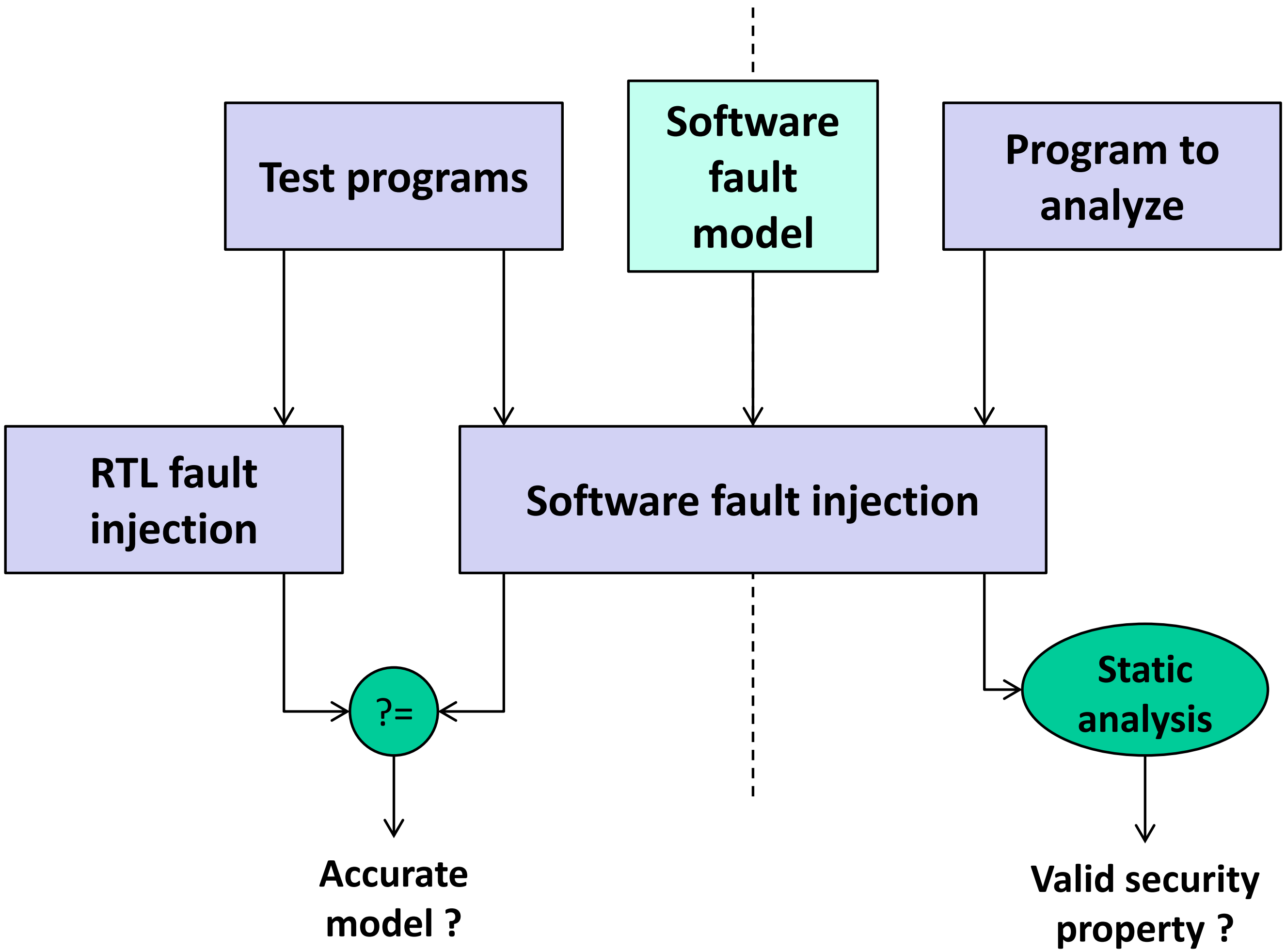
II. Approach

b. Software Fault Injection



- **Constraints:**
  - Models very different from one another
  - Need to model certain structures of the processor
  - Need to allow static analyses

II. Approach



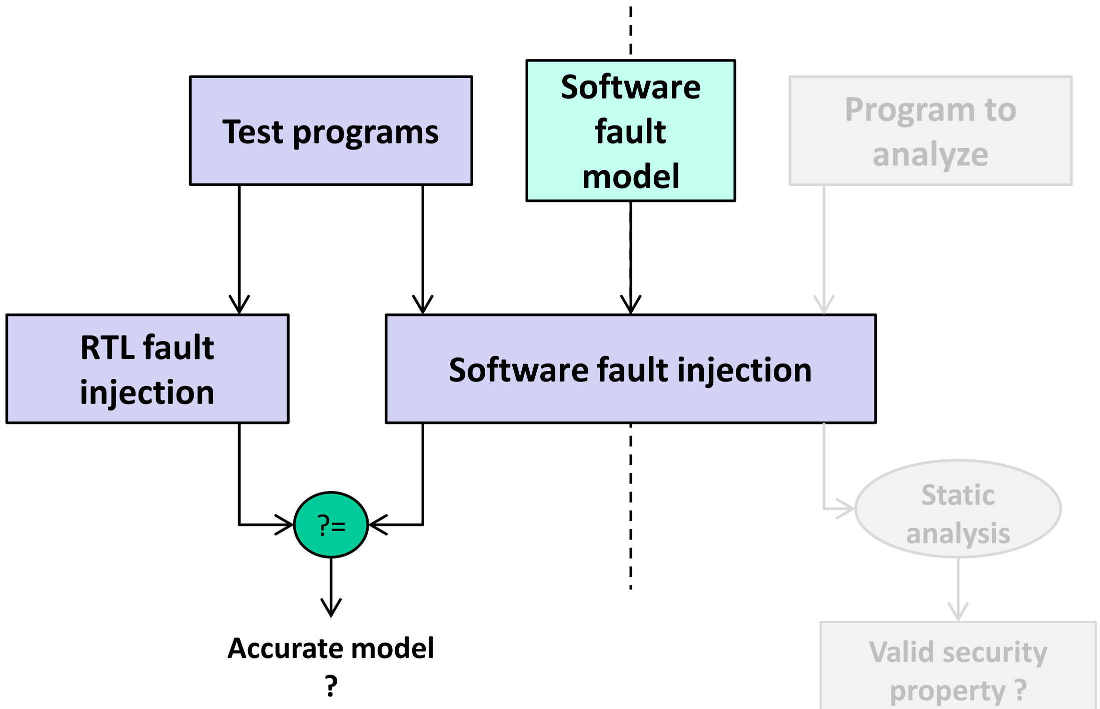
Test programs
Software fault model
Program to analyze
RTL fault injection
Software fault injection
?=
Static analysis
Accurate model ?
Valid security property ?

III. Case study

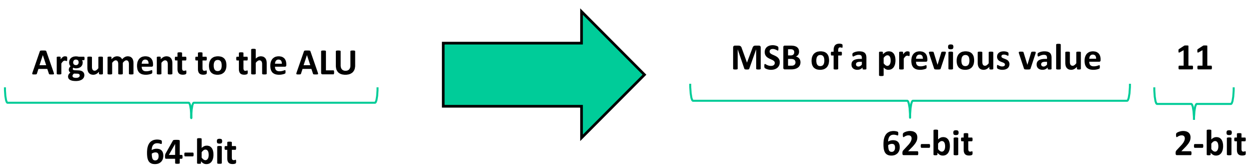
- VerifyPIN is a protected 4-digit PIN verification from the FISSC library [10], with the following countermeasures:
  - Hardened Booleans (0x55 for false and 0xAA for true)
  - Verification of the loop counter at the end of the loop
  - Duplicated Boolean tests.

```
  diff=FALSE; status=FALSE;
  for(i=0 ; i<4 ; i++){
     if(userPIN[i] != secretPIN[i])  diff=TRUE;
  }
  if(i != 4) countermeasure();

if(diff==FALSE){
       if(FALSE==diff)   status=TRUE;
       else countermeasure();
 } else status=FALSE;

return status;
```

III. Case study



Software Fault Model obtained through RTL simulation:



III. Case study

- **Frama-C Value analysis is based on *abstract interpretation***
- **Abstract interpretation [9] is used to abstract the semantics of an application. More precisely, it computes results on intervals instead of concrete values**
  - Instead of analyzing the program with individual values, we can analyze “simultaneously” many values.

```
int a = {0..9}
a++;              // a = {1..10}
```

- **It computes an over-approximation of the results (*sound* and *incomplete)***

```
int a = {0..9}
a++;              // a = {1..10}
a = pow(a,2);     // a = {1..100}
```

III. Case study

- **Security property to check:**

  For any user PIN different from the secret PIN, do not authenticate

- **The user and secret PINs are abstracted.**

III. Case study

- **There are 50 injection times possible:**
  - For 45, Frama-C *proves* that the property is secure against all user inputs
  - The other 5 (which point to the same instruction) are *potentially* vulnerabilities

- **A manual analysis showed that: if the first digit of the secret PIN has a value 0, 1, 2 or 3, the fault can reduce the program to two loop iterations instead of four**
  - → The countermeasures are not effective in this case (in particular the one that checks the loop counter)
  - → 40% of the possible secret PIN are vulnerable

- **How easy would it be to find the vulnerability with classical tools (with concrete values) ?**
  - The attack is successful if the first secret digit is 0-3 (40%) AND two loop iterations succeed (1%) ➔ overall, only 0.4% to find the vulnerability with concrete values for a given injection.

III. Case study

- **The attack was simulated at RTL**

- **This case study shows that:**
  - Complex fault models lead to undetected successful attacks
    - ➔ Justifies the use of the instrumentation tool

  - Some attacks only happen under specific circumstances, difficult to find using random, concrete data
    - ➔ Justifies the use of static analysis

IV. Discussion

a. Invariant properties

- **The properties have to be *invariant* relative to the abstracted states**

- **Example**
  - First idea: set all digits to {0..9}   (secret: XXXX ; user: XXXX)
    with the property : “if the PIN are different, do not authenticate”
  - Problem: Value analysis does not keep track of *relations* between variables
  - Solution: manually set a secret digit to a concrete value, and the corresponding user digit to everything except that value
    (secret: 0XXX ; user: ≠XXX)
    with the property: “do not authenticate”

IV. Discussion

b. Performances

- **How efficient is the method to analyze a program, compared to testing every value individually ?**
  - With the property: authentication ? 2.5x
  - With the property: loop count = 4 ? 10x
  - With 7-digit PIN instead of 4-digit: 2.5Mx and 10Mx

- **While very random, performances are better than simple executions of the program**

IV. Discussion

c. False positives

- **Value analysis computes an over-approximation of the states → false alarms**
  - No counter-examples
  - Need further analysis (with other tools or manually)
- **False alarms mean that the property could not be proved, but do not mean that it is not valid**

V. Conclusion

- **Our tool generates a C code that embeds complex software fault models**

- **Frama-C Value analysis can then be used to verify that security properties hold for any user inputs.**

V. Perspectives

- Other types of analysis ? Other tools ?
- Multiple injections ?
- Structure of the mutant has been designed to play nicely with Frama-C Value analysis, but needs to de adapted for other forms of analyses.

Thanks for your attention !

Questions ?

References

[1] Joint Interpretation Library, “Application of Attack Potential to Smartcards.” Jan-2013.

[2] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, “Formal verification of a software countermeasure against instruction skip attacks,” presented at the PROOFS 2013, 2013.

[3] M. L. Potet, L. Mounier, M. Puys, and L. Dureuil, “Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections,” in *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, 2014, pp. 213–222.

[4] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, “Random Additive Signature Monitoring for Control Flow Error Detection,” *IEEE Trans. Reliab.*, vol. 66, no. 4, pp. 1178–1192, Dec. 2017.

[5] M. Christofi, B. Chetali, L. Goubin, and D. Vigilant, “Formal verification of an implementation of CRT-RSA algorithm,” presented at the Security Proofs for Embedded Systems (PROOFS), 2012, pp. 28–48.

[6] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, “QEMUBased Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks,” in *2015 Euromicro Conference on Digital System Design*, 2015, pp. 530–533.

[7] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra, “Quantitative evaluation of soft error injection techniques for robust system design,” in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[8] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, “On the importance of Analysing Microarchitecture for Accurate Software Fault Models,” in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 561–564.

[9] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, USA, 1977, pp. 238–252.

[10] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, “FISSC: A Fault Injection and Simulation Secure Collection,” 2016, pp. 3–11.

V. Perspectives

- **Example:**
  - In the mutant, should we represent memory as an array for 64-bit data, or 8-bit ? (in any case, loading and storing instructions is made so that every byte is accessible).
  - Both work for a simple execution of the code

Initially	**Mem[0]=0x00000000**	**Mem[0]=0x00 Mem[1]=0x00 Mem[2]=0x00 Mem[3]=0x00**
Store {0..FF} at address 0	Mem[0]= min: 0x00000000 max: 0x000000FF	Mem[0]=0x00 – 0xFF Mem[1]=0x00 Mem[2]=0x00 Mem[3]=0x00
Store {0..FF} at address 3	Mem[0]= min: 0x00000000 max: 0xFF0000FF	Mem[0]=0x00 – 0xFF Mem[1]=0x00 Mem[2]=0x00 Mem[3]=0x00 – 0xFF

II. Approach

b. Software Fault Injection

```
<model name=”FFM”>
    <globals>      long fwd1 = 0,   fwd2 = 0;      </globals>
    <gold_end>     fwd2 = fwd1;    fwd1 = res;      </gold_end>
    <fault_ini>   if(injection_time==count) arg1=fwd2;  </fault_ini>
</model>
```

```
[…]
0x06ac:    ADDI  x15 = x0 + 85
[…]
```

Instrumentation tool

UNIVERSITÉ Grenoble Alpes

Grenoble INP

II. Approach

b. Software Fault Injection

```
<model name="FFM">
    <globals>      long fwd1 = 0,   fwd2 = 0;      </globals>
    <gold_end>     fwd2 = fwd1;    fwd1 = res;      </gold_end>
    <fault_ini>   if(injection_time==count) arg1=fwd2;  </fault_ini>
</model>
```

```
            [...]
0x06ac:  ADDI  x15 = x0 + 85
            [...]
```

Instrumentation tool

1

```
l06ac: // ADDI x15, x0, 85
  arg1 = reg[0];  arg2 = 85;        // Decode
  res = arg1 + arg2;                // Execute
  reg[15]=res;                      // Write-Back
```

II. Approach

b. Software Fault Injection

```
<model name=”FFM”>
    <globals>     long fwd1 = 0,  fwd2 = 0;     </globals>
    <gold_end>     fwd2 = fwd1;    fwd1 = res;    </gold_end>
    <fault_ini>   if(injection_time==count) arg1=fwd2;  </fault_ini>
</model>
```

```
                 […]
0x06ac:    ADDI  x15 = x0 + 85
                 […]
```

Instrumentation tool

2

```
l06ac: // ADDI x15, x0, 85

  arg1 = reg[0];  arg2 = 85;          // Decode

 res = arg1 + arg2;                   // Execute
 fwd2=fwd1; fwd1=res;
 reg[15]=res;                         // Write-Back
```

II. Approach

b. Software Fault Injection

```
<model name=”FFM”>
    <globals>     long fwd1 = 0,   fwd2 = 0;     </globals>
    <gold_end>     fwd2 = fwd1;    fwd1 = res;      </gold_end>
    <fault_ini>  if(injection_time==count) arg1=fwd2;  </fault_ini>
</model>
```

```
                 […]
0x06ac:    ADDI  x15 = x0 + 85
                 […]
```

Instrumentation tool

3

```
l06ac: // ADDI x15, x0, 85

 arg1 = reg[0];  arg2 = 85;          // Decode
if(injection_time==count) arg1=fwd2;
res = arg1 + arg2;                   // Execute
fwd2=fwd1; fwd1=res;
reg[15]=res;                         // Write-Back
```

UNIVERSITÉ Grenoble Alpes

Grenoble INP