Analyzing Software Security against Complex Fault Models with Frama-C Value 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

Summary

I. Introduction

- **There are multiple ways to study the security of software against fault injection.**
- **Software methods 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]
- **The methods are usually closely coupled with a particular fault model**
- **Problem: there are hardware fault effects that are not modelled in typical software fault models [7]**

I. Introduction

- **Some effects obtained in simulation in the 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
  - …
- **Lot of complexity in modelling these models**
- **How to conduct efficient security analyses with these complex software fault models?**

Summary

II. Software fault injection with complex fault models

a. Problem: Complexity of the models

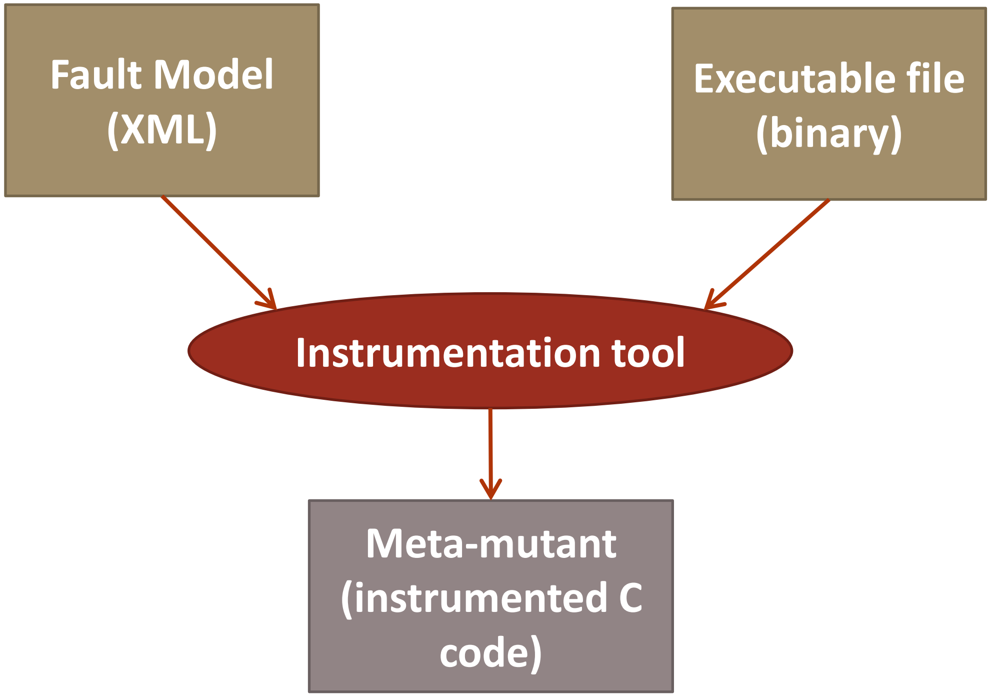
- **How to take these complex software models into account ?**

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

II. Software fault injection with complex fault models

b. Solution: Code instrumentation

- **Solution:**
  – From the executable, construct an instrumented C code to inject faults from complex software fault models



II. Software fault injection with complex fault models

b. Solution: Code instrumentation

- **Goal : reproduce at the software level the behavior of the hardware**

```
<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

II. Software fault injection with complex fault models

b. Solution: Code instrumentation

- **Goal : reproduce at the software level the behavior of the hardware**

```
<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. Software fault injection with complex fault models

b. Solution: Code instrumentation

- Goal : reproduce at the software level the behavior of the hardware

```
<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
```

UNIVERSITÉ Grenoble Alpes

Grenoble INP

II. Software fault injection with complex fault models

b. Solution: Code instrumentation

- Goal : reproduce at the software level the behavior of the hardware

```
<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
```

II. Software fault injection with complex fault models

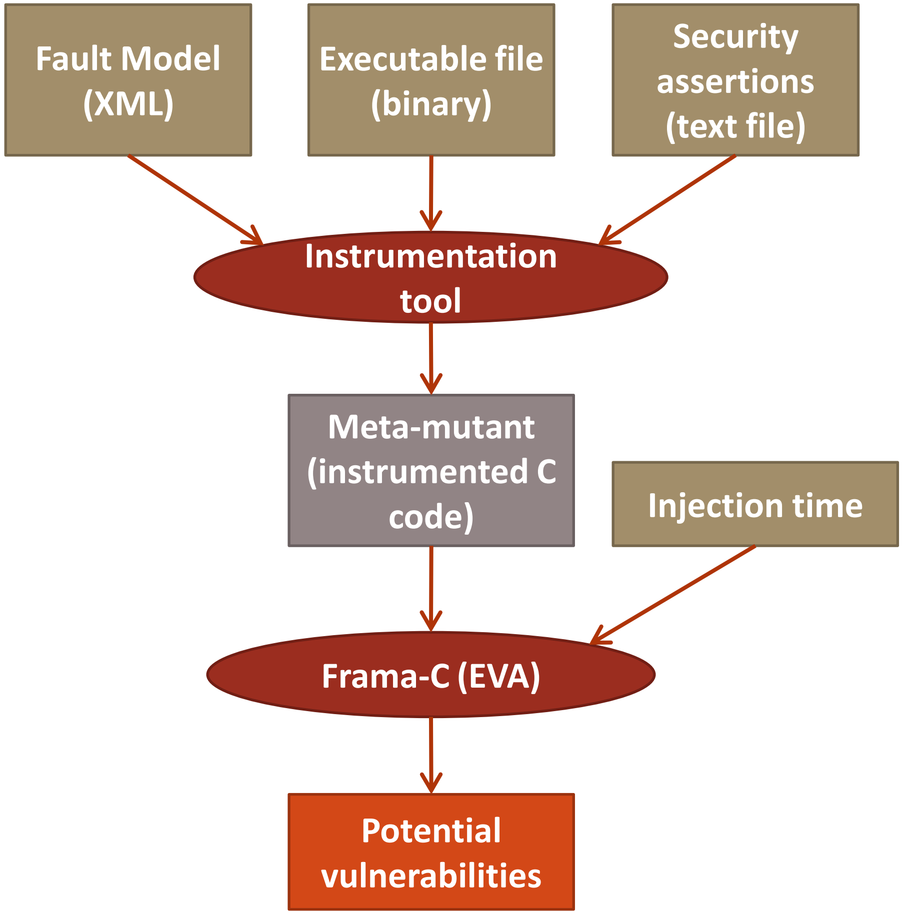
b. Solution: Code instrumentation

- **Verification of the method (fault-free):**
  - Goal: verify that the generated code behaves correctly
  - Use of RISC-V test vectors
  - Test of each instruction with different values and contexts, and comparison with pre-computed values

Summary

III. Security analysis with Frama-C Value Analysis

- **Static analysis is used to prove the validity of security properties (for example, check the number of loop iterations)**



III. Security analysis with Frama-C Value Analysis

- **Value analysis is based on *abstract interpretation***
- **Abstract interpretation [9] is used to abstract the semantics of an application. Concretely, 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}
```

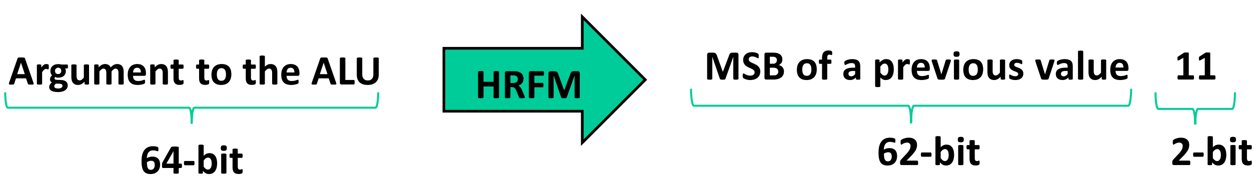
Summary

IV. Case study: VerifyPIN

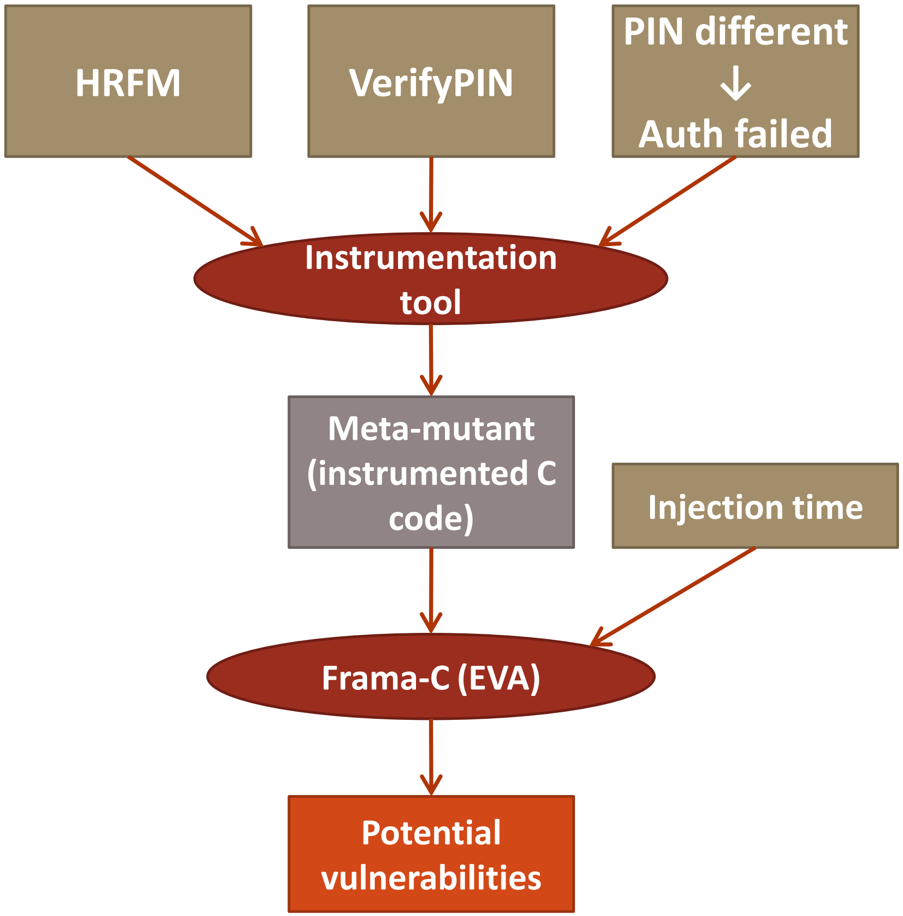
- **VerifyPIN is a protected 4-digit PIN verification from the FISSC library [10], with the following countermeasures:**
  - Hardened Booleans (0x55 for false and 0xAA for true)
  - Verification of the loop counter at the end of the loop
  - Duplicated Boolean tests.

- **Security property: If secret and user PIN are different, do not authenticate (the secret digits and user digits are abstracted (detailed later))**

- **Software Fault model HRFM (Hidden Register Fault Model): model obtained through RTL simulation**



IV. Case study: VerifyPIN



IV. Case study: VerifyPIN

- **There are 50 injection times possible:**
  - For 45, the property is *proven* 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

IV. Case study: VerifyPIN

- **The attack was simulated at RTL**
- **It 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

Summary

V. Discussion

a. Invariant properties

- **How did we abstract the values in the case study ?**
  - 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”
- **The properties have to be *invariant* regarding the abstracted states**
  - In the example, we know that for all the abstracted states, the authentication has to fail
  - If we wanted to check the number of loop iterations, we could use secret:XXXX and user:XXXX, since the loop count should always be 4

V. 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 ? 10x
  - With 7-digit PIN instead of 4-digit: 2.5Mx and 10Mx

- **Performances are difficult to estimate**
  - Depends on multiple factors: abstraction, property used, fault model, application, semantic unrolling, etc
  - Experiments on AES AddRoundKey: 2^{256} states at once; analysis can take minutes or hours depending on the settings

V. Discussion

c. False positives

- **False alarms mean that the property is not correct, but do not mean that there is a vulnerability**
- **Value analysis computes an over-approximation of the states → false alarms**
  - No counter-examples
  - Need further analysis (with other tools or manually)
- **The primary goal of the method is to prove the correctness of security properties, not to find vulnerabilities**
  - In the AES experiments, on the 190 possible injection times, 141 were proven safe and 49 undecidable

VI. Conclusion

- Our tool can generate a C code that embeds complex software fault models
- Frama-C Value analysis can then be used to verify security properties whatever the user inputs.
- Although its performances are good compared to a simple execution with concrete values, it can be difficult to define correct properties
- The analysis either proves a property (is correct), or does not (but that does not mean that there is a vulnerability). The remaining cases have to be studied more closely.

UNIVERSITÉ Grenoble Alpes

Grenoble INP

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.