978-1-7281-5347-6/19/$31.00 ©2019 IEEE

Predicting the Effect of Hardware Fault Injection

Etienne Boespflug
Dept.of Computer Science
University of Limoges
Limoges, France
etienne.boespflug@etu-unilim.fr

Romain Gourier
Dept.of Computer Science
University of Limoges
Limoges, France
romain.gourier@etu-unilim.fr

Jean-Louis Lanet
LHS
INRIA-RBA
Rennes, France
jean-louis.lanet@inria.fr

***Abstract*—Fault attacks on the hardware are now common to extract valuable information from a device. The most used means is the electromagnetic perturbation. Designers have to verify how robust are their designs against this kind of attack. We designed a prediction tool that generates mutants according to a fault model and then checks security properties on the mutants. We improved the generation of mutants at the instruction level and at the function level to reduce the evaluation time. we reduce that number between 32% and 84% according to the fault model.**

***Index Terms*—Fault Injection, Prediction, Pruning heuristic**

I. Introduction

Fault Injection (FI) is a classical technique used since the 60s [1]. The original aim was to evaluate the effect of cosmic radiation on the embedded equipment in a space environment. It consists to observe the behavior of a system in presence of faults which are defined thanks to a fault model. At the beginning, FI was applied on hardware components and now it is applied on the complete system. It becomes mandatory to verify how security software behave in presence of intentional fault injection.

The defender needs to predict the effect in order to evaluate the potential effects of the attack on the hardware. On the other side, if the attacker has access to the binary, she can target the valuable part to be attacked. We demonstrated in [2], the possibility to generate buffer overflow even in presence of canary, control flow hijacking, back-door activation or Return Oriented Programming (ROP) exploit execution.

Our contribution in this paper is to generate in an efficient way a prediction of the fault effect. Each instruction is mutated according to a fault model. A pruning algorithm is used to reduce the number of mutant code. We evaluate this approach on real code and we verify them on our FI platform. This paper is organized as follows. Section II presents pour motivation and the context of this work. SectionIII describes the fault models considered in this work. The contribution is developed in section IV in two parts, the generation of the mutant instruction and its optimization and then the heuristics at the function level. We evaluate our work in section V and we describe the evaluation bench insection VI. Finally, we present related works in section VII.

II. Motivation

Faults are events in circuit operation that occur for a short time period. A fault induces an error which in turn can lead the system into an undesirable state from a functional or security point of view. The effect of this event can modify the behavior of the computer but it can also modify the content of a memory cell, a register used into a computation or a data/instruction on the bus during its transit from memory to register. The stored code in memory is not the executed one under fault attack. The semantics of the program changes and the entire program should be verified to prevent it.

One of the solutions is to simulate by software the fault behavior, thus requiring to have a precise model of the fault effect and a precise description of the hardware. Fault simulators try to estimate through a fault model the effect of fault on the software. The challenges are related to precisely model the effect of the fault on the hardware and to manage the combinatorial explosion of states if one wants to evaluate all the possibilities. At the hardware level the fault can be as precise as a bit flip or can impact more elements like several byte at the same time.

The major drawback of simulation tools is related to the state space problem. Tools can enumerate all the possible faults leading to different mutant programs. Unfortunately the number of different FI is huge and most of the time generates a no effect behavior. Then the tool must verify if some properties are valid or not on all the mutants.

We developed a complete tool chain to verify if a given program can invalidate security properties under fault attacks. In [4], we already presented the verification process of the mutants. Firstly the security properties are meshed with the source code as annotations and the corresponding binary is generated. A golden run verifies if the properties hold in nominal mode. From the initial binary, we generate a set of mutant according to the fault model. The executable binary is transformed into an intermediate representation in Low Level Virtual Machine Intermediate Language (LLVM-IR). This model is checked using the Low Level Bounded Model Checker (LLBMC) to verify if it satisfies the properties or not. The SimFI tool is used to automatically generate the mutant binaries by injecting faults into the executable binary according to the chosen fault model.

The initial SimFi tool had a naive design and generate too much mutants that were not meaningful binaries. We focus in this paper in a new generation of the mutant generation process. This new mutant generator is based on an object representation of each field of an instruction. This allows to eradicate early in the process meaningless instructions and to apply new pruning algorithms. These algorithms used information available at the function under evaluation.

III. The fault model

There are different possibilities to generate faults in the hardware either by the natural environment (failure FI) or by an hostile attacker (security FI). We focus here on intentional failure by an attacker. In failure studies, the environment behaves randomly while for security FI, the attacker controls the fault distribution (time and space) up to its knowledge of the perturbation means. A fault has three properties, *i.e.* location, fault-type and time. Location denotes where the fault is injected, fault-type denotes which type of fault that is injected and time denotes when the fault is enabled. Sometime this last attribute is qualified as injection trigger and fault latency. Thus, the fault model as a third fault-space dimension leading to an important amount of potential behavior. If one fixes a specific fault model, then a two-dimensional fault space suffices to characterize faults reducing the amount of runs. Nevertheless, several optimizations are still needed if one wants to explore the entire space. For the fault type, it corresponds at the hardware level to a change in a transistor. At the bit level, we distinguish several types of fault: bit-set, bit-flip, bit-reset, stuck-at and random-value. It can manifest as single- or multi-bit faults but also whole byte or burst of bytes in memory. The memory cell can be part of internal CPU state, Instruction Set Architecture (ISA) visible CPU registers, or any other part of the memory hierarchy, including CPU cache SRAM or main memory DRAM. Fault persistence is an important fault property of hardware faults that may occur, vanish, and re-occur in a non-deterministic manner. For example, if the fault occurs in a cache, it is persistent until the cache is flushed. FI models have been already discussed in details in [3], [5].

For this study, we target the ARM architecture which is widely deployed in Secure Element (SE) but also in a huge variety of IoT devices and embedded systems. The architecture (ARMv7-M) has several characteristics that have to be taken into account. It uses a particular little-endian type based on the minimal instruction size and not on the maximum size (Thumb mode). The specific encoding of the first bits (111) of the instruction specifies if the instruction is coded on 32 bit or not.

Each instruction of the instruction set can often be represented by different encoding. An instruction consists in two parts: the op-code and the fields. The op-code is the part of the instruction that defines the nature of the instruction and the fields are an information specific to the instruction, such as the destination register.

We use the `MOV` immediate instruction as an example. This instruction is divided into three different possible encodings T1, T2 and T3. As shown in Figure 1, each encoding defines the properties differently. For the T1 encoding, the immediate value is represented by the eight lower bits of the instruction and the T3 encoding use the concatenation of four separated

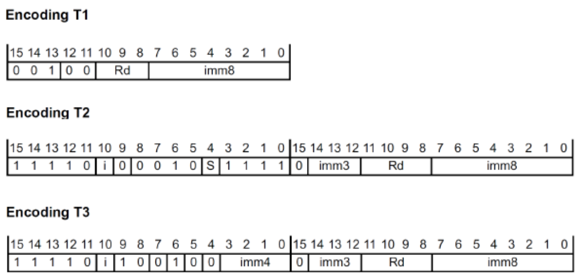


Fig. 1. The `MOV` encoding instruction

bit sets. For the T1 encoding, the fault model implies two possibilities either the low byte or the high byte. For T2 and T3 we have four possibilities. But some of them does not correspond to a valid instruction and thus must be eradicated.

IV. Generation of the mutants

Native Mutant Generator's Application Programming Interface (API) is the main part of the tool. It provides disassembling, ELF file extraction, object oriented instruction representation, Control Flow Graph (CFG) manipulation and heuristics implementation. The object oriented representation of the instruction define the mutation either on the instruction or on its properties such that it generates only valid mutant instructions. The tool has been designed within three layers:

- The API is in charge of loading an Executable and Linkable Format (ELF) file, representing all instructions as objects and executing and executing the mutations;
- The function layer extracts the CFG of the function and implements heuristics to verify the liveness of the mutation;
- The front end is in charge of applying the mutation strategy.

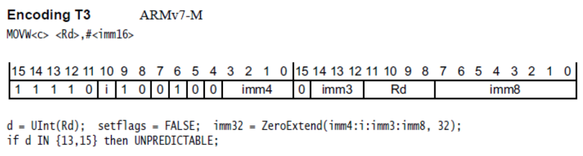
Mutant generation can be applied to the whole `.text` section, to multiple instructions sections or on a single function.

A. The API for instruction mutation

In order to disassemble and analyze the binary file, the tool knows the structure of each ARM instruction. The fields of the instructions are made of several data:

- the type (commonly a signed or unsigned integer) and the boundaries;
- the conditions on its value;
- the way it is stored into the instruction;
- and various other parameters (such as validity, effect on register, processor behavior).

We reuse the example of the `MOV-immediate-T3` instruction given Figure 2. The corresponding instruction is a pattern identified by the name `MOV(immediate) T3` and the bit pattern `11110x100100xxxx0xxxxxxxxxxxxxxx`. The fields (xx elements in the bit pattern) of this instruction are defined by `Rd` and `imm32`. `Rd` is the destination register and `imm32` the immediate value is obtained by the concatenation of `imm4, i, imm3` and `imm8`.



Encoding T3 ARMv7-M

```
MOVW<c> <Rd>,#<imm16>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

Fig. 2. The `MOV` T3 fields

The mutation of this instruction is polymorphic according to the fault model. The verification procedure checks if the resulting instruction is valid or not. In that case, if the concatenation of the elements of the fields `0, imm3, Rd` and `imm8` is a valid instruction, then the mutant is generated. In the second case, the instruction remains the same but only the fields change: destination register is `r0` and the immediate vale becomes the concatenation of `imm4, i` and `0`.

If the fault model is the `Single bit flip`, then the mutation for this instruction can replace the instruction by one having the same fields, change the value of the register `Rd` or change the immediate value. For the destination register, the strategy could affect a living register of the considered function or a non used register which leads to a side effect outside the scope of the function. This is the case if the register is read after the execution of the function but not write. The last strategy is to use register leading to unpredictable behavior. Unfortunately this last strategy often generates a run time abort of the program, according of the used chip.

B. Locally alive registers

For a given function, modifying a register which is not used has no effect inside the function. Of course, this can have an effect on the caller. The ARM Application Binary Interface (ABI) defines the set of register `r0-r3` as caller save register while `r4-r11` are callee-save register. Thus, the compiler generates a specific instruction at the beginning and at the end of the function according to the registers used. For example, the instruction `stmdb sp!, {r4, r5, r7, r9 sl, lr}` (or a simple `push` instruction if few registers) at the function entry. The instruction `stmfd sp!, {r4, r5, r7, r9, sl, lr}` (or a simple `pop` at the exit of the function. It means that beside the register `r0-r3`, the registers `r4, r5, r7` and `r9` are used in this function. Any instruction modifying a non alive register in that function is not generated even if it has a sens outside the function.

C. The immediate value heuristic

By hypothesis, we have both the source code and the binary code. Modifications are done at the binary level but heuristics can be also applied at the source level. A first heuristic concerns the boolean value. This information is available in the source code using the C type boolean. In secure programming, boolean are often used as `enum` type or with a `typedef` coded with specific value having the maximum of Hamming distance (*e.g.* 0x5555 and 0xAAAA).

Nevertheless, this information remains available whatever the form a boolean can take. Thus any immediate value in a field which corresponds to a boolean can be replaced by three values: the one that characterize `true, false` and an arbitrary value corresponding to a fault. We do not yet implement an integer range analysis to reduce the domain of the other integer type. This is reserved for future work, knowing that SE often use byte and short value only.

D. Control flow equivalence

This heuristic uses the observation that a dangerous mutant is one that does not execute all the paths. In particular in the SE domain, faults try to avoid security checks. The heuristic reduces the mutant to only those that have a different control graph. Thus the control graph of the golden run is known and compared to the one of the mutant code. If they do not differ, then the mutant is kept else it is eradicated. This heuristic is bounded to a function, meaning it has no side effect apart the selected function. It implies at the instruction level that no control flow instruction are transformed if possible according to the fault model to a control flow instruction and vice versa. For example, consider the instruction to mutate with a T2 encoding `ADD<c> SP,SP,#<imm7>`, says `1011 + 0000 0xx`. If the fault model is `Single bit flip`, says only one bit can change then we can transform it into a Compare and Branch on Zero (`CBZ`). `CBZ` is encoded `1011 + 0001 xxx`. But it can not be changed to a Compare and Branch on Non Zero (`CBNZ`)which requires 2 bits to flip.

E. Pruning the mutants

Each instruction has its own validating function such that a first round of pruning is done at instruction level. But some mutations can be valid at the instruction level but useless at the function level. Smith et *al.* are among the first concisely describing the classical def/use analysis technique [6] that was subsequently reinvented several times, *e.g.*, by Benso et al. [7], [8].

The basic idea is that all fault locations between a def (a `write`) or use (a `read`) of data in memory, and a subsequent use, are equivalent regardless of when exactly in this time frame a fault is injected there. The earliest point where it will become architecturally visible is when the corrupted data is read. Moreover, if the fault generates a `write` instruction which is followed in the program by a `write` before a `read` then, the fault will have a reduced effect. Instead of conducting one experiment for every point within this time frame, it suffices to conduct a single experiment (for example, at the latest possible time directly before the `read`), and assume that all similar faults in the interval will produce the same outcome. Thus, all those mutants should be killed. Similarly, all points in time between a `read` or `write` and a subsequent `write` are known to result in no effect, as the corrupted data will be overwritten in all cases.

Another useless mutation corresponds to an already checked mutation. For example, if the instruction `MOVE r0, r1,` `r2` is changed into `MOVE r0, r0, r0` the second instruction is a valid instruction and generates a valid mutant. Then, if a new generation transforms the initial instruction into a `MOVE r1, r1, r1` the resulting mutant is equivalent to the previous one and it can be pruned.

F. The WWR/RRW heuristic

The WWR and RRW heuristics can be applied on a particular instruction or in the whole program. In the first case, the WWR and RRW patterns are checked locally with the specified instruction. The objective is to verify if a mutation leads to a pattern, and thus to prune this mutation. The patterns are therefore checked with the instruction. The algorithm stores the list of the read and the written memory locations by this instruction. Each instruction following the original instruction will be checked. For each instruction, the list of the read and written memory locations is compared with the original list.

For instance, if the original instruction I_0 reads a memory location M_1 and write on M_2 and M_3 and the currently analyzed instruction I_n read M_1 and M_2 the program can deduce the following statements :

- Both I_0 and I_n read on M_1, thus there is a RRW pattern for the memory location M_1.
- The mutation does not imply a pattern for the memory location M_2 because I_0 writes on M_2 and I_n reads on M_2. Thus, the memory location is removed from the list of the read and written of I_1.
- This instruction does not use M_3, the next instruction will be checked.

G. Known issues

We have some known issues in our implementation of the new Simfi tool. The first drawback is related to the portability of the tool. The tool has been designed for one architecture (ARMv7-M) and the effort to adapt it to another ISA is not negligible even if we want to extend it to an ARM-V8 architecture. Each instruction model must be redefined, which represents a huge effort even if the classes corresponding to the instructions are clearly identified.
The second drawback is related to specific implementations. The program does not work properly in case of obfuscated binary due to the disassembling process. Currently, the CFG cannot handle the half-branching case and this pattern results with an error when trying to extract the CFG. Indeed, the support of this type of pattern implies that the program cannot be represented with a unique list of instructions and that every byte of the code could be detected many time differently during the CFG recursive traversal. Currently, our tool works correctly on non obfuscated programs.

H. Future works

At the function level, more optimizations can be done. We are working currently on two directions one is a preliminary analysis, known as domain equivalence, the second one is related to state equivalence.

1) Integer domain: The idea is to generalize the optimization on the boolean value to byte and short values. A preliminary pass on the source code can determine the bounds of value that an integer can use. We envisage to use an abstract interpretation pass to compute statically approximate but sound value ranges for program variables. Actually only boolean values are mutated in instructions. Using such an analysis, we will be able to evaluate at least one value out of the bounds or severals according to the different bounds an integer can have.

2) Fault equivalence: If for two simulations, the machine state is completely identical to the state of another simulation at a given instruction location then it will yield to the same final state. Then any mutant which after a fault injection reaches an already seen state can be eradicated. The equivalence can be either a control equivalence or a data equivalence. The control equivalence is of a particular importance for SE programs and is already treated at the instruction level.

This reduction requires the execution of the mutant. The idea is to check the state of the program after the mutation. If it has already be visited, then it can be pruned. The state can be defined by the content of the registers. Taking into account the memory in the state would be another challenge.

V. Evaluation

The SimFi tools has only three fault models: `Stuck at zero` (byte or word), `Stuck at one` (byte or word), single `NOP` and `Single bit flip`. The improved version of SimFi adds the double `NOP` and `Multiple bit flips` for Control Flow equivalence heuristic. The double `NOP` fault model corresponds to a fault injected in the pipeline as published in [10]. The cortex-M3 uses a dual stage pipeline and a fault can eliminate the two instructions pushed in the pipeline. This version does not improve the results of the previous tools for the fault models `Stuck at one` (word) and `Stuck at zero` (word). They do not correspond to a valid instruction.

The following metrics are related to the example given in [9] on the control flow hijacking.

For the `Stuck at zero` (byte) we reduce the number of mutant by 40% and on the `Stuck at one` (byte) we reduce to only 32%.

The dual `NOP` fault model introduces more vulnerable codes without generating false mutants. It corresponds to what is observable on physical fault injection.

The more interesting is the improvement corresponding to the `Single bit flip`. We have obtained between 63% and 84% of mutant reduction. The explanation comes mainly on the definition of the ARM instruction set. There are many cases where a bit flip correspond to an `Undefined` instruction which corresponds sometime to `NOP` operation. In that case, we generate only one mutation for a `NOP`, the other are skipped. In other cases, instructions are invalid and we do not generate them.

VI. The bench for verify the prediction

Experiments have been performed in our labs within our fault injection platform. The targeted board is an STM32VLDISCOVERY board embedding an ARM Cortex-M3 core. Fault injection is performed with a signal forming chain consisting in a pulse generator, a signal generator and a power amplifier. The signal is connected to a home made probe located on the targeted chip as shown in Figure 3. A synchronization signal is sent by the targeted chip to the pulse generator.

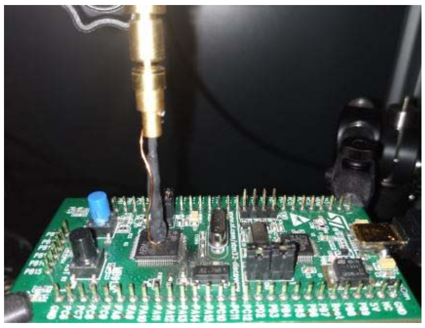


Fig. 3. The probe injection

The first step is to experiment on the chosen target hardware with the chosen hardware fault induction technique. This step consists of exploring the spacial surface of the target hardware while manipulating the parameters to induce the fault using the induction technique. This step requires a lot of time and expertise, since exploring all the possible combinations of spacial, temporal, and induction parameters is not feasible in reasonable time. Once the best spatial alignment is detected we synchronise the injected fault with the believed vulnerable point to demonstrate a fault injection vulnerability. Several physical parameters are to be adjusted to try and detect the known vulnerability in the loaded program. The three most important parameters are the power, the duration of the pulse and it starts. We demonstrated that the predicted effects where realistic vulnerabilities according to the fault model.

VII. Related Work

There has been much work in the area of fault detection to predict the impact of hardware faults.

SimPLFIED [11] aims at finding all faults that escape detection and lead to software faults. It uses a symbolic execution method to abstract the state of erroneous values in the program. It injects such a symbolic error at all possible instructions and uses model checking with the abstract execution technique to explore all possible paths.

Feng and al. [12] design a static analysis that identifies instructions where faults are likely to be detected quickly. Such a fault appears if there is a short path in the dataflow graph from the fault to a symptom generating instructions. Only rest of the faults are considered vulnerable and they duplicate instructions to mitigate the fault.

The Relyzer tool [13] proposes methods to determine when application-level faults are equivalent, enabling comprehensive analysis by injecting faults in only one instruction at the binary level per equivalence class. The fault model is a single bit fault. They developed several pruning algorithm to reduce the fault space (Known-outcome pruning technique, def-use analysis, control equivalence, store-equivalence, bounding branch targets...).

VIII. Conclusion

We try to predict how a fault in the hardware can have security consequences at the software level. We developed a complete tool chain than can verify the impact of a EM fault. The tool is based on the generation of mutants. Unfortunately, the generation is too basic and leads to too many mutants to be checked.

In this paper, we present an improvement of the tool regarding the number of mutants. It is based on a smarter generation of mutants and pruning heuristics. We evaluate our solution on some binaries related to security software. We have improved the initial solution reducing drastically the time required for the prediction.

Other heuristics can be implemented at function level to recognize states already evaluated but at a higher cost.

References

[1] Hardie, Fred H and Suhocki, Robert J, *Design and use of fault simulation for Saturn computer design'*, IEEE Transactions on 412–429, 1967
[2] Bukasa, Sebanjila K and Lashermes, Ronan and Lanet, Jean-Louis and Leqay, Axel, *Let's shock our IoT's heart: ARMv7-M under (fault) attacks'*, Proceedings of the 13th International Conference on Availability, Reliability and Security, pp.33, 2018
[3] Blömer, Johannes and Otto, Martin and Seifert, Jean-Pierre, *A new CRT-RSA algorithm secure against bellcore attacks*, Proceedings of the 10th ACM conference on Computer and communications security, pp. 311–320, 2003
[4] Given-Wilson, Thomas and Jafri, Nisrine and Lanet, Jean-Louis and Legay, Axel, *An automated formal process for detecting fault injection vulnerabilities in binaries and case study on PRESENT*, 2017 IEEE Trustcom/BigDataSE/ICESS, pp. 293–300, 2017
[5] Wagner, David, *Cryptanalysis of a provably secure CRT-RSA algorithm*, Proceedings of the 11th ACM conference on Computer and communications security, pp. 92–97, 2004
[6] Smith, D Todd and Johnson, Barry W and Profeta, Joseph A and Bozzolo, Daniele G, *A method to determine equivalent fault classes for permanent and transient faults*, Reliability and Maintainability Symposium, 1995. Proceedings., Annual, pp. 418–424, 1995
[7] Benso, Alfredo and Rebaudengo, Maurizio and Impagliazzo, Leonardo and Marmo, Pietro, *Fault-list collapsing for fault-injection experiments*, Reliability and Maintainability Symposium, Proceedings., Annual, pp. 383–388, 1998
[8] Berrojo, Luis and González, Isabel and Corno, Fulvio and Sonza Reorda, Matteo and Squillero, Giovanni and Entrena, Luis and Lopez, Celia, *New techniques for speeding-up fault-injection campaigns*, Proceedings of the conference on Design, automation and test in Europe, pp. 847, 2002
[9] Jafri, Nisrine, *Formal fault injection vulnerability detection in binaries : a software process and hardware validation*, PhD, 2019, url = http://www.theses.fr/2019REN1S014/document
[10] Ludovic Claudepierre and Philippe Besnier, *Microcontroller Sensitivity to Fault-Injection Induced by Near-Field Electromagnetic Interference*, IEEE International Symposium on Electromagnetic Compatibility and IEEE Asia-Pacific Symposium on Electromagnetic Compatibility (EMC/APEMC), 2019

[11] Pattabiraman, Karthik and Nakka, Nithin and Kalbarczyk, Zbigniew and Iyer, Ravishankar, *SymPLFIED: Symbolic program-level fault injection and error detection framework*, IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN) , pp. 472-481, 2008

[12] Feng, Shuguang and Gupta, Shantanu and Ansari, Amin and Mahlke, Scott, *Shoestring: probabilistic soft error reliability on the cheap*, ACM SIGARCH Computer Architecture News, Vol38, number 1, pp. 385-396, 2010

[13] Hari, Siva Kumar Sastry and Adve, Sarita V and Naeimi, Helia and Ramachandran, Pradeep, *Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults*, ACM SIGPLAN Notices, Vol. 47, number 4, pp. 123–134, 2012