When fault injection collides with hardware complexity

Sebanjila Bukasa, Ludovic Claudepierre, Ronan Lashermes, Jean-Louis Lanet

▶ **To cite this version:**

Sebanjila Bukasa, Ludovic Claudepierre, Ronan Lashermes, Jean-Louis Lanet. When fault injection collides with hardware complexity. FPS 2018 - 11th International Symposium on Foundations & Practice of Security, Nov 2018, Montréal, Canada. p.243-256, 10.1007/978-3-030-18419-3_16 . hal-01950931

HAL Id: hal-01950931
https://inria.hal.science/hal-01950931

Submitted on 17 Jan 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

When fault injection collides with hardware complexity

Sebanjila Kevin Bukasa[1], Ludovic Claudepierre[1], Ronan Lashermes[1], and Jean-Louis Lanet[1]

LHS INRIA
{first.last}@inria.fr

Abstract. Fault Injections (FI) against hardware circuits can make a system inoperable or lead to information security breaches. FI can be used preemptively in order to detect and mitigate weaknesses in a design. FI is an old field of study and therefore numerous techniques and tools can be used for that purpose. Each technique can be used at different levels of circuit design, and has strengths and weaknesses. In this paper, we review these techniques to show their pros and cons and more precisely we highlight their shortcomings with respect to the complexity of modern systems.

1 Introduction

In the field of hardware security, Fault Injection (FI) is a technique to alter the correct execution of a program in a chip. The resulting errors can be harnessed in order to weaken the security of the device, by extracting cryptographic keys for example. In the case of hardware security, the distinction between errors (the internal system state is erroneous) and failures (the behaviour does not follow specifications) is blurred. Indeed, the attacker can observe, or deduce, the state of the device though its interaction with the environment; thus it is considered that the attacker can observe errors and exploit them. For example, a timing attack can leak a password during its verification. It is therefore common to use the term *errors* to designate either errors or failures.

A fault may be caused by radiation (laser pulses, electromagnetic pulses, alpha particles, ...), power glitches, clock glitches, abnormal temperatures, etc. Faults are naturally found in hardware, but can also be voluntarily caused by an attacker. In all cases, they can often be exploited for malicious activities. Therefore faults must be mitigated.

FI can be used to infer the faults that can be created in a system, to analyse the errors created as a consequence and whether they make the system vulnerable. The difficulty is in the trade-off between the size of the

state space to explore and the speed of the analysis. We will show that the complexity of modern system renders FI tools less precise because they cannot accurately model the erroneous states.

In this paper, after a context presentation in section 2, we review the techniques and tools to assess the vulnerability of a device to FI in section 3. The shortcomings of actual techniques will be presented in section 4 as well as a discussion on how to improve them. Finally, the conclusion is drawn in section 5.

2 Context safety/security

FI is an old research discipline [1,2,3,4], which originates from the study of fault tolerant systems, mainly from aerospace. FI is defined by Arlat [5] as a validation technique of dependability for fault tolerant systems. It consists in observing system behaviour in presence of faults defined with a fault model. At the beginning, FI was applied on hardware components. Consequently, corresponding fault models were comprised of effects that were deemed representative for failing logic elements, in particular stuck-at logical zero or one. One would be able to inject a fault at transistor level which models an unintended physical effect, such as a signal transition caused by a heavy ion hit and resulting in a communication error at system level for example. While this approach is close to reality, a practical implementation is barely possible.

All FI techniques aim to solve several problems:

- Injection of faults;
- Observation of their effects;
- Intrusiveness of the solution;
- Capacity to explore the entire state space.

The FI techniques have been recognized for a long time necessary to validate the dependability of a system by analysing the behaviour of devices when a fault occurs. More recently, secure devices have to face fault attacks which are similar to failure problems. Efforts have been made to develop techniques for injecting faults into a system prototype or model.

When considering information security, fault injection assumes that the attacker is able to target specific assets in the system. It means that she knows exactly what kind of behaviour she requires to reach her goal. In case of targeting cryptographic algorithms [6,7] or assets (keys, tokens, ...) several solutions have been proposed to protect them against fault

injections [8]. Applications can be designed to be resilient against FI, but this resilience mainly focus on software execution of these applications, in some cases this can be a problem, indeed a complete confidence is given to hardware.

3 Fault Injection techniques

Several techniques exist to inject faults, all of them with advantages and disadvantages. Here is an overview of these techniques.

3.1 Hardware-based FI

Hardware based FI aims at disturbing hardware with physical and environmental parameters (heavy ion radiation, electromagnetic interferences [9], *etc.*), injecting voltage dips on power rails [10,11] laser fault injection [12] or modifying the value of some pins with circuit editing. The main advantage of this family of techniques over the other solutions is that they evaluate the final device. To achieve this kind of FI it is necessary to possess a final version of the evaluated device.

The effects of physical injections are difficult to control and repeatability of experiment is hard to achieve. To obtain repeatability, instead of injecting physically a fault, injection mechanisms emulate effects of physical perturbations on hardware such as pin-level FI [13].

Fault Injection system for Study of Transient fault effects (FIST) uses heavy-ion radiation or power disturbance faults to create faults inside a chip when it is exposed to radiation. It can cause single or multiple bit-flips producing transient faults at random locations directly inside a chip, which cannot be done with pin-level injections.

Messaline [5] is a pin-level fault forcing system. It uses both active probes and sockets to conduct pin-level fault injection. It can inject stuck-at, open, bridging and complex logical faults, among others. It can also tune the duration of the fault existence and its frequency. RIFLE [14] is also a pin-level fault injection system for dependability validation. This system can be adapted to a wide range of target systems and faults are mainly injected in processor pins. FI is deterministic and can be reproduced if needed. Different kind of faults can be injected and the fault injector is able to detect whether the injected fault has produced an error or not without specific hardware.

Obviously, hardware-based tools are also hardware dependent. Furthermore, the setup of these hardware-based injectors is rather complex.

3.2 Simulation-based FI

Simulation based hardware fault injection techniques simulate hardware description of tested circuit using high-level models (mostly Hardware Description Language (HDL) models). It consists in injecting faults into that model to evaluate their impacts. Most of the tools modify the hardware description of tested circuit to include the components necessary to inject faults. These fault injection components can be designed to inject different fault behaviours depending on the fault model. Faults can also be injected using hardware description language simulator commands which allow variables and signals of circuit being modified.

A major disadvantage of simulation based techniques is that they are extremely slow. Simulating the register transfer level (RTL) description of a circuit is multiple orders of magnitude slower than actual circuit operation speed. Hence, even for relatively small processors, simulation based fault injection tools can only evaluate fault propagation for a very short time interval.

VERIFY [15] (VHDL-based Evaluation of Reliability by Injection Faults Efficiently) uses an extension of VHDL for describing faults correlated to a component, enabling hardware manufacturers to express their knowledge of fault behaviour on their components. Multi-threaded fault injection which uses checkpoints and comparison with a golden run is used for faster simulation of faulty runs. Proposed extension to VHDL language unfortunately requires modification on language itself. VERIFY uses an integrated fault model which cannot be extended.

MEFISTO-C [16] conducts fault injection experiments using VHDL simulation models. The tool is an improved version of MEFISTO tool which was developed jointly by LAAS-CNRS and Chalmers. MEFISTO-C uses a VHDL simulator and injects faults via simulator commands in variables and signals defined by a VHDL model. It offers to users a variety of predefined fault models as well as other features to set-up and automatically conduct fault injection campaigns.

FAUMachine [17] is a tool allowing simulation of complete systems, it was the main core for different works in the field of fault injections [18,19]. Its particularity is that it allows to simulate various types of faults and in various devices connected to the system, while making possible the observation of the impacts on the total operation of the system

3.3 Emulation-based FI

System emulation uses hardware prototyping on Field Programmable Gate Arrays (FPGA) based logic emulation systems [20], [21]. This technique has been presented as an alternative solution in order to reduce time spent during simulation-based fault injection campaigns.

This technique allows designer to study the actual behaviour of circuits in application environment, taking into account real-time interactions. However, when an emulator is used, initial VHDL description must be complete and fully synthesizable. Modified circuit contains sequences of operations which can flip their output bit based on a control signal value. Such techniques require an additional control mechanism to specify time and location of fault injection in circuit. If such a control mechanism is implemented in circuit, its complexity increases with number of fault injectable memory elements.

Antoni *et al.* [22] proposed a technique to inject a fault on chosen memory elements at run time on a FPGA using runtime reconfiguration. This eliminates the need for having a complex control circuit to determine injection location. However, the time required to reconfigure the circuit could be significant when compared to the total application run time.

Civera *et al.* [20] proposed another solution to provide a more flexible control over runtime fault injection. They used modified flip-flop circuits capable of injecting faults based on a control bit associated with each flip-flop. All these control bits are tied together like a scan-chain and at run time can be programmed to inject fault in any desired flip-flop in the circuit.

3.4 Software implemented FI

The objective of these techniques consists in reproducing at software level errors that would have been produced by faults at hardware level. They are mostly used in order to detect and predict vulnerabilities with respect to hardware fault injection. Software implemented fault injection (SWIFI) tools use a software level abstraction of fault models in order to inject errors in software while it runs or by modifying programs before execution. This approach does not need any hardware modification. SWIFI provides a way to test complete systems including the operating system and the applicative layer. This makes SWIFI techniques quite popular and a large number of such tools exists, Table 1 summarizes some of them and explore their particularities.

The most common fault models are:

Table 1. Overview of some SWIFI techniques

SWIFI technique	Fault model	Fault target	Injection point
CEU [23]	Bit flip	Variables	Runtime (interruptions)
DOCTOR [24]	Bit flips	Communications, variables	Preruntime
EFS [25]	Bit flips, code insertion, data modifications	Control flow, variables	Runtime (OS service)
FERRARI [26]	Address, data or flags modifications	Control flow, variables	Runtime (parallel process)
FIES [27]	Bit flip, bridging and stuck-at faults	Control flow, variables	Runtime
XCEPTION [28]	Bit flip, bridging and stuck-at faults	Variables	Runtime (interruptions)

– instruction skip (one or several instructions are not executed),
– instruction modification (one or several instructions are modified according to a pattern such as single bit-flip, random change, ...).

Common software mechanisms used for run time FI, such as perturbation functions require a modification of the program. Unfortunately, this extra instrumentation causes execution overhead that will affect the system behaviour (speed, memory consumption, ...). For example, FERRARI [26] and EFS [25] tools require some context switches between its fault injection process and target system process.

A common problem with run time approach is the intrusiveness which refers to the alteration of the original system due to fault injection experiment setup (*e.g.* changes in program flow, additional components, temporal variation,...). Depending on the actual intention of fault injec-

tion, respective tools have to cope with completely different requirements. In contrast to an ideal tool which always provides low intrusiveness, high visibility and high performance, available tools are only specialized on a subset of these requirements.

The major drawback of SWIFI is related to state space problem. The tools often generate much more faults than any other techniques (since the abstraction level has a richer representation, *i.e.* there may be 2^{32} possible instruction values in a 32-bit system and less that 2^{32} wires in the chip). Yet most of the time generated faults do not lead to failures, the error may have been silently suppressed during the execution. The challenge is to either generate only a minimal set of faults (those that can lead to a Silent Data Corruption) or to prune them while they are generated. This leads to several optimization phases during simulation and remains a difficult challenge.

In the context of information security, errors can often be exploited even in the absence of failures. An error can cause copying of a secret in a vulnerable part of memory for example. Since SWIFI tools use a software level abstraction of fault models, they cannot capture such vulnerabilities.

4 Techniques validity

We consider ourselves as evaluators. When it comes to FI, we want to evaluate if a technique is more appropriate in order to evaluate behaviour of a device when a fault occur.

Various injection means exist and several techniques have been using them in different way and targeting several type of devices. Since simulation and emulation based techniques require a white-box model (access to HDL sources, ...) that are most of the time not available to evaluators.

In this section we limit ourselves to hardware-based and software-based injections techniques.

4.1 Experiences

In order to test the consistency of SWIFI models, in particular their software level abstraction of fault models with real observations, we conducted different experiments, which we will present here.

Faustine platform Our platform, called Faustine 1, is made of a Keysight 33509B pulse generator, a Keysight 81160A signal generator and a Milmega

80RF1000-175 power amplifier, connected in sequence to generate a signal. This signal then passes through a Langer RF probe RF B 0.3-3 located on the targeted chip to generate an Electromagnetic Fault Injection (EMFI).

Fig. 1. Overview of Faustine platform

In order to launch a fault injection, a synchronization signal (a trigger) is sent by the targeted chip General-Purpose Input/Output (GPIO) (controlled from the code) directly to the 33509B pulse generator. This experimental trick, possible when the attacker has control of the code (*i.e.* only for vulnerability assessment) is not mandatory. Other synchronization possibilities include sniffing communications with the target or measuring its EM emissions to find a relevant pattern.

The location of the probe on the chip was chosen after a scan that determined the most sensitive area on the chip. The same location was kept for all experiments.

Microcontroller We first analyse a Microcontroller (μC). The targeted board is an STM32VLDISCOVERY board with an STM32F100RB chip,

embedding an ARM Cortex-M3 core running at 24MHz (41.7ns clock period). As shown in Figure 2, probe is just on top of the chip.

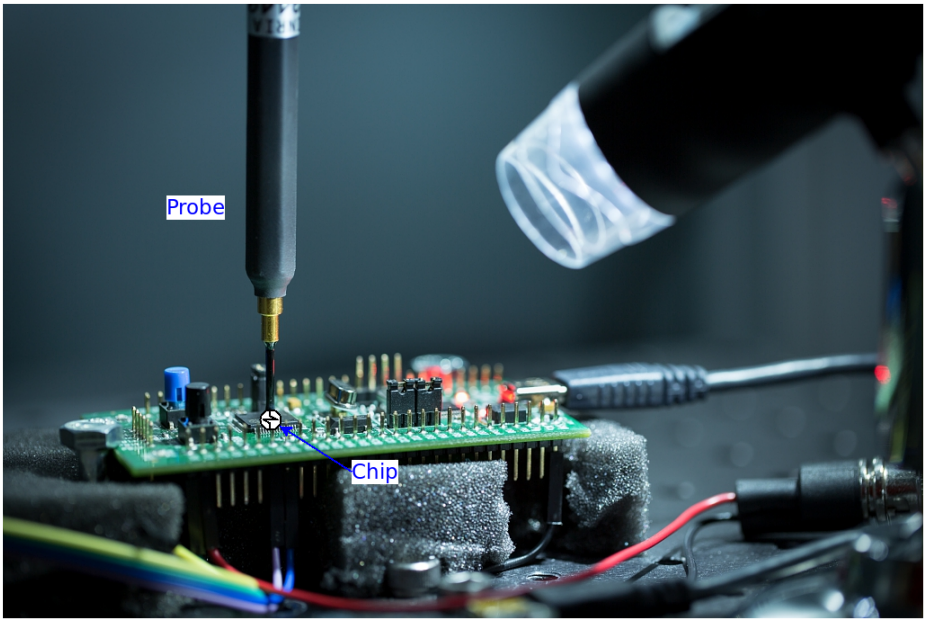

Fig. 2. STM32 under probe

On this board the tested software is a PIN code checker, the entered PIN code is compared with the internal PIN code if it is false (`false=1` in 1.1), the status variable takes the value `0xFFFFFFFF`, otherwise it takes `0x55555555`. Thus in the first case, access will be denied, in the second it will be granted.

```
if(false == 1) {
   status = 0xFFFFFFFF; }
else {
   status = 0x55555555; }
```

Listing 1.1. Targeted C code

```
cmp   r3, #1                ; r3 contains *false*
ite   eq                    ; if then else
moveq.w r4, #4294967295     ; 0xFFFFFFFF
movne.w r4, #1431655765     ; 0x55555555
```

Listing 1.2. Resulting assembly (thumb2)

As we can see on listing 1.2, in order to modify the behaviour of the program and thus get access without the PIN code, we can target the *if then else (ite)* instruction. If it is possible to not execute it, then the next two instructions will execute in sequence and, as their result is stored in the same register ($r4$), only the second assignment will have an impact (overwriting the first one).

In the case of SWIFI, we consider the software level abstraction of fault model by deleting (manual edition of the binary) this instruction which allows us to see that it is indeed the right target, then we target the execution of this instruction with a hardware fault.

In this way, when we inject our fault, we try to synchronize with the code snippet in listing 1.2 and target the instruction `ite eq`. In 10% of the cases, the execution is faulty (`status = 0x5555555`), proving that the SWIFI allows us in this case to find a point of sensitivity and thus to inject our fault effectively.

However, we found that different timings (over a span of 5 instructions) were able to get our faulty behaviour. This can have several plausible explanations, such as the fact that several different skipped instructions can lead to the same impact, or that the `ite eq` instruction can be impacted at different levels of its execution pipeline.

System-on-Chip We then analysed a System-on-Chip (SoC). The targeted board is a Raspberry Pi3 board with a BCM2837 chip, which embeds 4 ARM Cortex-A53 cores, running at up to 1.2 GHz (833ps clock period).

```
while(1){
  wait(x*desynch_value+x);
  turn_on_LED(y);
  wait(x*activation_duration+x)
  turn_off_LED(y);
}
```

Listing 1.3. Targeted C code

Here we want to evaluate the impact of a fault and compare it to the SWIFI models. The goal is to see if a hardware generated fault can be explained by a software abstraction of the fault model, represented by software modification. Thus we inject faults at different timings during the execution of a loop (listing 1.3) on 2 of the 4 cores, others being used to communicate with the host, while desynchronizing them (they are not started at the same time). The 2 cores (x) are activating their own signal

(y) during a given time in parameter ($x * activation\_duration + x$), this leads to a time span visible in Figure 3.

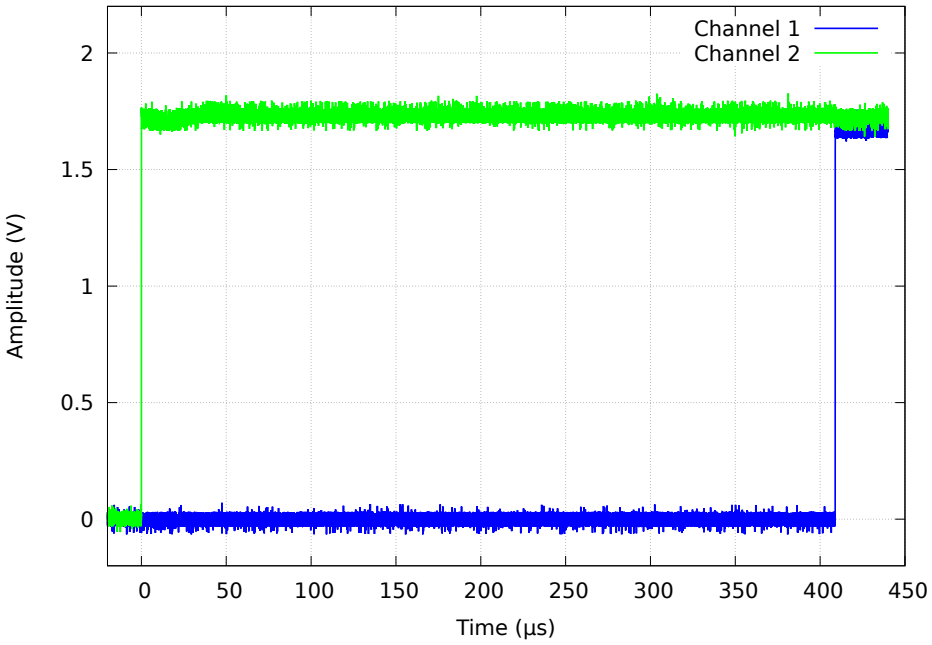

Fig. 3. Signals are desynchronized. Channel 1 for GPIO signal sent by core 1, channel 2 for GPIO signal sent by core 2. Time span between the two rising edges is due to "$x * desynch\_value + x$" in 1.3.

Whatever the timing of the injection, the impact was the same: this had the effect of largely modifying the execution time of the loop on each core, alternately faster or slower in a random way. Another effect is to synchronize the different cores between them (in Figure 4), but also to break one of the two channels of communication with our host (application channel on one core and debug channel using JTAG).

In this case we were not able to find a match with software abstraction of the fault model as usually used in SWIFI techniques. So this lead us to question what makes the difference between a μC and a SoC and thus what prevents us from using SWIFI in the second case.

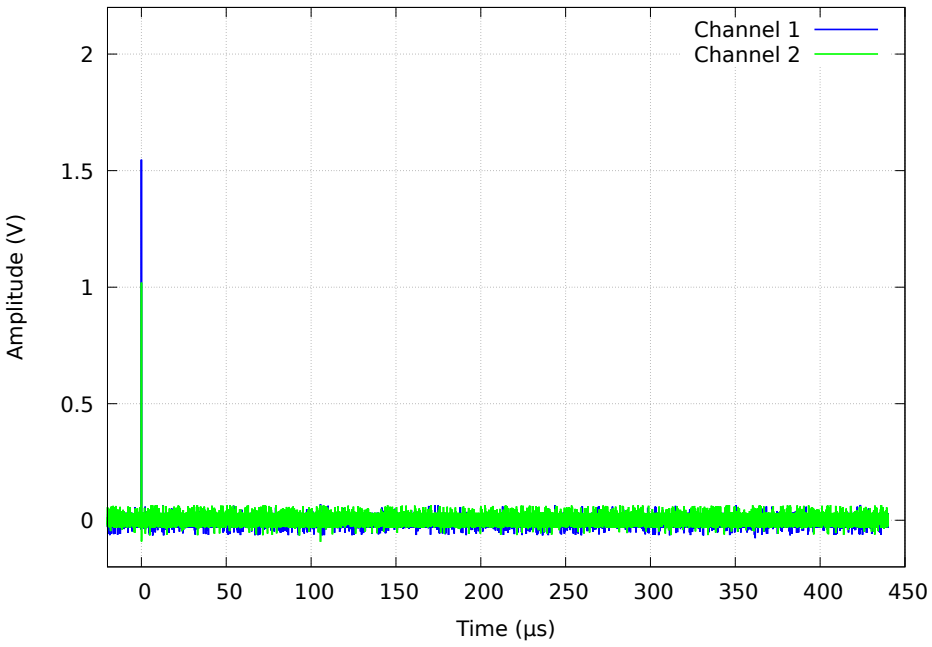

Fig. 4. Signals are shorter and synchronized. First, time span seems to have disappeared, then "$x * activation\_duration + x$" (in 1.3) seems to have changed to be equal in the 2 cores.

4.2 System complexity

Abstraction layers A computing system is a complex device. In order to allow humans to build mental models of how such systems work, this complexity is often hidden behind abstraction layers as visible in Figure 5.

There is a main division between these layers corresponding to the hardware/software interface constituted by the Instruction Set Architecture (ISA). On the upper side, software is constituted of a succession of instructions. On the lower side, the micro-architecture (hardware) is responsible for upholding this abstract representation.

The micro-architecture is widely different if we consider a μC or a SoC. In the first case, the instruction execution flow is quite simple, with a single core, a simple memory hierarchy, in-order execution, *etc.* In the case of a SoC, the micro-architecture can be quite complex. Several core can share the same memory space, with a complex memory hierarchy (several cache levels, shared or not). Instructions can be executed out-of-order or even speculatively. What happens in hardware differs from the simple model provided by the ISA.

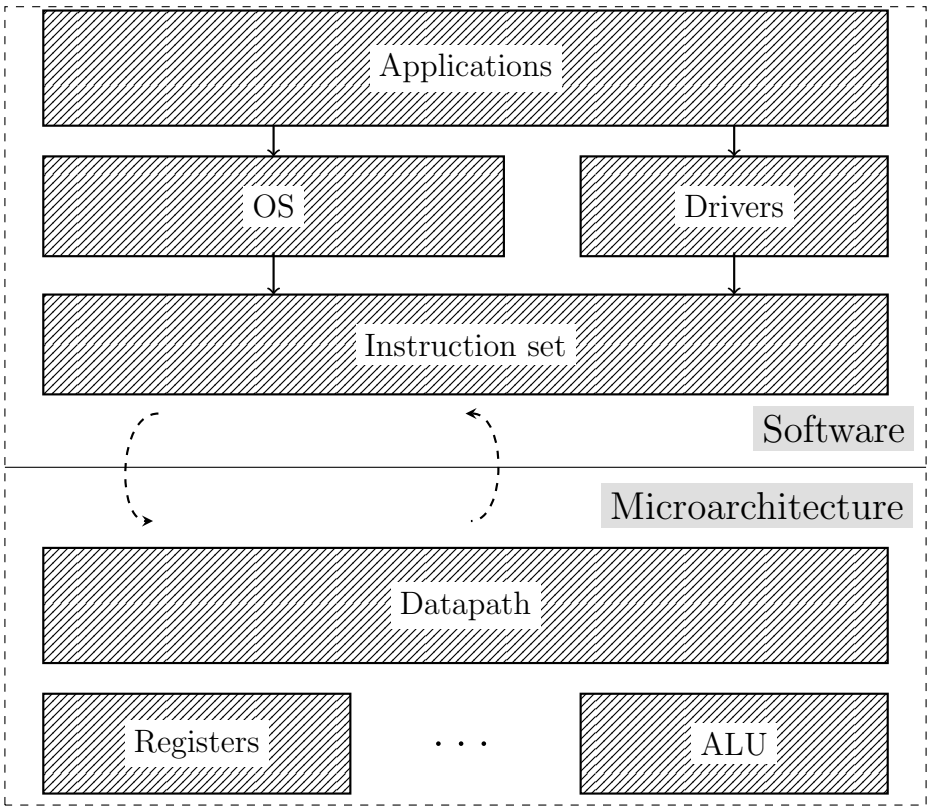

Fig. 5. Abstraction layers

SWIFI shortcomings The hardware part is mostly fixed, the application designer cannot modify it whereas she controls the software part of the application. In consequences, in order to protect her application, she will act on the software only. This fact remains a main reason that SWIFI techniques are quite popular: they allow the application developer to act upon the results. Therefore SWIFI techniques are preferred by software developers whereas hardware-based fault models are preferred by hardware designers in order to secure the system.

The problem is that the application still executes on a given hardware that may or may not be vulnerable to fault injection. The application developer would like to free herself from this responsibility by considering only software.

Yet SWIFI cannot capture the full extent of hardware fault injection consequences. Indeed they are not able to analyze the range of interac-

tions and components present at the hardware level (the microarchitecture in 5) by abstracting the behaviour at the software level. Consider a Direct Memory Access (DMA) transfer for example. In this case, a section of memory is copied to another without the Central Processing Unit (CPU) involvement. Instructions are present to describe the desired memory transfer then it is enforced in parallel of the program execution. Therefore, any fault on the DMA transfer cannot be captured by a SWIFI technique.

Complexity evolution It can be argued that cases that cannot be captured by SWIFI, such as DMA transfers, are special cases not representative of classical applications.

But as we have show in section 4.1, if these asynchronous behaviours are seldom present in simple systems, they are ubiquitous in modern SoCs. In order to squeeze the maximum performance out of modern SoCs, a lot of processing is done in parallel of the instruction flow execution.

The recent trend is in more complex systems, not simpler. As a consequence, SWIFI techniques are less and less able to capture the extent of possible errors in these systems.

5 Conclusions

FI tools are quite useful in the context of dependability and information security. They can be used to assess the security of a system with respect to fault attacks. Application developers mostly use SWIFI tools to predict the behaviour of their program in the event of a fault according to a software abstraction of the fault model. However, we have seen that the part targeted by the fault attacks is at the microarchitecture level which is the physical representation of the system, we have seen that in the case of a simple system, such as an μC (also in [29,30]), it was possible to find an abstraction at the software level of behaviour occurring at the hardware level. Nevertheless, through the experiments we conducted it appeared to us that on systems where the microarchitecture is more complex, as in the case of the SoC it became complex to find an abstraction at the software level of the models of faults corresponding to those generally considered by SWIFI methods (bit-flip, stuck-at, skip instruction, etc.). As a consequence, SWIFI is less and less relevant for such systems.

References

1. F. H. Hardie and R. J. Suhocki, "Design and use of fault simulation for saturn computer design," *IEEE Transactions on Electronic Computers*, no. 4, pp. 412–429, 1967.
2. D. Armstrong, "A deductive method for simulating faults in logic circuits," *IEEE Transactions on Computers*, vol. 21, no. undefined, pp. 464–471, 1972.
3. E. G. Ulrich, T. Baker, and L. Williams, "Fault-test analysis techniques based on logic simulation," in *Proceedings of the 9th Design Automation Workshop*. ACM, 1972, pp. 111–115.
4. P. R. Menon and S. G. Chappell, "Deductive fault simulation with functional blocks," *IEEE Transactions on Computers*, vol. 27, no. 8, pp. 689–695, 1978.
5. J. Arlat, "Validation de la sûreté de fonctionnement par injection de fautes, méthode- mise en oeuvre- application," Ph.D. dissertation, 1990.
6. M. Joye and M. Tunstall, *Fault analysis in cryptography*. Springer, 2012, vol. 147.
7. R. Lashermes, J. Fournier, and L. Goubin, "Inverting the final exponentiation of tate pairings on ordinary elliptic curves using faults," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 365–382.
8. A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov 2012.
9. N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," *arXiv preprint arXiv:1402.6421*, 2014.
10. N. Timmers, A. Spruyt, and M. Witteman, "Controlling pc on arm using fault injection," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on*. IEEE, 2016, pp. 25–35.
11. M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, C. A. Ardagna and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 224–233.
12. S. Buchner, D. Wilson, K. Kang, D. Gill, J. Mazer, W. Raburn, A. Campbell, and A. Knudson, "Laser simulation of single event upsets," *IEEE Transactions on Nuclear Science*, vol. 34, no. 6, pp. 1227–1233, 1987.
13. J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.
14. H. Madeira, M. Rela, F. Moreira, and J. G. Silva, "Rifle: A general purpose pin-level fault injector," in *European Dependable Computing Conference*. Springer, 1994, pp. 197–216.
15. V. Sieh, O. Tschache, and F. Balbach, "Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*. IEEE, 1997, pp. 32–36.
16. P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. IEEE, 1998, pp. 284–293.

17. V. Sieh, "Faumachine." [Online]. Available: http://www3.informatik.uni-erlangen.de/EN/Research/FAUmachine/description.shtml
18. S. Potyra, V. Sieh, and M. D. Cin, "Evaluating fault-tolerant system designs using faumachine," in *Proceedings of the 2007 workshop on Engineering fault tolerant systems*. ACM, 2007, p. 9.
19. M. Sand, S. Potyra, and V. Sieh, "Deterministic high-speed simulation of complex systems including fault-injection," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 211–216.
20. P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and A. Violante, "Exploiting fpga for accelerating fault injection experiments," in *On-Line Testing Workshop, 2001. Proceedings. Seventh International*. IEEE, 2001, pp. 9–13.
21. R. Leveugle, "Fault injection in vhdl descriptions and emulation," in *Proceedings-IEEE-International-Symposium-on-Defect-and-Fault-Tolerance-in-VLSI-Systems*. IEEE Comput. Soc, Los Alamitos, CA, USA, 2000, pp. 414–19.
22. L. Antoni, R. Leveugle, and M. Feher, "Using run-time reconfiguration for fault injection in hardware prototypes," in *Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002. Proceedings. 17th IEEE International Symposium on*. IEEE, 2002, pp. 245–253.
23. R. Velazco, S. Rezgui, and R. Ecoffet, "Predicting error rate for microprocessor-based digital architectures through c.e.u. (code emulating upsets) injection," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2405–2411, Dec 2000.
24. S. Han, K. G. Shin, and H. A. Rosenberg, "Doctor: An integrated software fault injection environment for distributed real-time systems," in *Computer Performance and Dependability Symposium, 1995. Proceedings., International*. IEEE, 1995, pp. 204–213.
25. L. Riviere, J. Bringer, T.-H. Le, and H. Chabanne, "A novel simulation approach for fault injection resistance evaluation on smart cards," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 2015, pp. 1–8.
26. G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Transactions on computers*, vol. 44, no. 2, pp. 248–260, 1995.
27. A. Höller, T. Rauter, J. Iber, and C. Kreiner, "Diverse compiling for microprocessor fault detection in temporal redundant systems," in *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, Oct 2015, pp. 1928–1935.
28. J. Carreira, H. Madeira, J. G. Silva *et al.*, "Xception: Software fault injection and monitoring in processor functional units," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 245–266, 1998.
29. B. Yuce, N. F. Ghalaty, and P. Schaumont, "Improving fault attacks on embedded software using risc pipeline characterization," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2015 Workshop on*. IEEE, 2015, pp. 97–108.
30. L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of armv7-m architectures," *arXiv preprint arXiv:1510.01537*, 2015.