Noname manuscript No.
(will be inserted by the editor)

Combined Software and Hardware Fault Injection Vulnerability Detection

Thomas Given-Wilson · Nisrine Jafri · Axel Legay

Received: date / Accepted: date

Abstract Fault injection is a well known method to test the robustness and security vulnerabilities of software. Software-based and hardware-based approaches have been used to detect fault injection vulnerabilities. Software-based approaches typically rely upon simulations that can provide broad and rapid coverage, but may not correlate with genuine hardware vulnerabilities. Hardware-based experiments are indisputable in their results, but rely upon expensive expert knowledge and manual testing yielding ad-hoc and extremely limited results. Further, there is very limited connection between software-based simulation results and hardware-based experiments. This work bridges software-based and hardware-based fault injection vulnerability detection by contrasting results of both approaches. This demonstrates that: not all software-based vulnerabilities can be reproduced in hardware; prior conjectures on the fault model for Electro-Magnetic Pulse attacks may not be accurate; and that there is a co-relation between software-based and hardware-based approaches. Further, combining both approaches can yield a vastly more accurate and efficient approach to detecting genuine fault injection vulnerabilities.

Keywords fault injection · vulnerability · statistical model checking · formal methods · EMP

1 Introduction

Fault injection is a commonly used technique to test the robustness or vulnerability of systems against potential physical fault injection attacks. Testing for system robustness is generally applied for systems that are deployed in hostile environments where faults are likely to occur. Such environments include aviation, military, space, etc. where atmospheric radiation, *Electo-Magnetic Pulse* (EMP), cosmic rays etc. may induce faults. Vulnerability of modifications to program behaviour via fault injection attacks is a useful way to detect places where a malicious attacker could exploit a system via targeted fault

T. Given-Wilson
Universite Catholique de Louvain
Place Sainte Barbe 2 bte bte L5.02.01
1348 Louvain-la-Neuve
Belgium
E-mail: thomas.given-wilson@uclouvain.be

N. Jafri
CEA
17 Avenue des Martyrs
38000 Grenoble
France
nisrine.jafri@cea.fr

A. Legay
Universite Catholique de Louvain
Place Sainte Barbe 2 bte bte L5.02.01
1348 Louvain-la-Neuve
Belgium
E-mail: axel.legay@uclouvain.be

injection. Since the underlying mechanism of a fault causing undesirable behaviour is common to both of these scenarios, the detection of potential *fault injection vulnerabilities* is an important area of research.

To support this research requires being able to reproduce the effect of some kind of fault in an experimental environment. There are two broad classes of approaches used to reproduce such faults, either simulating the fault injection using a *software based approach*, or (re)producing the fault injection with some specialised equipment as a *hardware based approach*.

The software based approach was first proposed as an alternative to requiring specialised hardware to (re)produce a fault [6, 22, 54]. The typical software approach is to perform a simulation based upon a chosen *fault model*; a model of how the fault affects the system.

The main advantage of software based approaches is that they are cheap and fast to implement since they require only development skills and normal computing systems without any specialised hardware. The main challenge for software based approaches is that they have not been validated against hardware based approaches to verify that their results are co-related, i.e. that the vulnerabilities found by the software based approaches are genuine.

The hardware based approach was proposed as a technique to study potential vulnerabilities which may be created by environmental factors or potential malicious attacks [31]. The hardware based approach consists of using specialised hardware to induce an actual fault on a specific device. Some examples include: setting up a laser that can target specific transistors in a chip [45], setting up an X-ray beam to target a transistor [3], mounting an EMP probe over a chip to disrupt normal behaviour [33], and many others [5, 7, 50]. The main advantage of such hardware based approaches is that any detected vulnerability is guaranteed to be genuine. The main challenge for such hardware based approaches is the cost of specialised hardware and expertise to configure such an environment and conduct the experiments.

Since both software and hardware based approaches have advantages (and disadvantages), research has proceeded using both approaches. Thus, there are many works that explore software based approaches [12, 30, 36] or hardware based approaches [7, 37, 46]. Very few consider both [15, 4] and these do not attempt to find co-relations between the two. Due to the relative cost and also the potential for broader and faster results, the more recent focus has been on improving software based approaches [21, 36, 39].

One challenge that has not been explored is the correspondence between software-based and hardware-based fault injection vulnerability detection. This paper sets out to remedy this and bridge the gap between software-based and hardware-based fault injection vulnerability detection. This is achieved here by performing both software-based and hardware-based fault injection vulnerability detection. The software-based detection uses a generic process [21, 20] and here implemented using software simulations of fault injection and formal methods analysis using statistical model checking [28]. The hardware-based detection is performed using an EMP generator. The results of the two approaches are compared to explore co-relations between them. The results of these experiments yielded several interesting outcomes.

- The software-based approach is able to find genuine fault injection vulnerabilities. However, there are also many false-positive results where the software-based approach claims a vulnerability exists that was not feasible to reproduce using the (EMP) hardware-based approach. This indicated that although software-based approaches may be useful in identifying *potential* fault injection vulnerabilities, not all such vulnerabilities are genuine.
- The hardware-based approach did *not* match any single model of how a fault affects the system using the software-based approach. By having comprehensive results from the software-based simulation it was possible to determine that the (EMP) hardware-based approach did not have a consistent or exact effect on the hardware.
- The two approaches have a co-relation: both approaches agree on the effect and the location of fault injection vulnerabilities (and other behaviours). The results here indicated that although the software-based approach had false-positives, there were no false-negative results when considering the fault models used here. This indicates that software-based detection can indicate likely locations for vulnerabilities, and hardware-based approaches can be used to confirm (or refute) their feasibility.
- Combining both approaches can be used to rapidly locate genuine fault injection vulnerabilities, even in code without known weaknesses. This paper presents a method to use the software-based approach to identify the most potentially vulnerable locations and then (with some calculation) these can be tested and confirmed (or refuted) using hardware-based approaches. In practice this combined approach can vastly reduce the number of hardware experiments required to demonstrate a vulnerability; here reducing the number of experiments from tens or hundreds of thousands to just

210. Further, when applied to code without known weaknesses this can be used to rapidly determine if vulnerabilities exist.

The key contributions of the paper are as follows:

- Performing software-based fault injection and hardware-based fault injection on common case studies.
- Comparing the results of software-based and hardware-based fault injections.
- Showing that software-based and hardware-based fault injections have a co-relation.
- Presenting a combined approach to fault injection vulnerability detection that is more effective than either software-based or hardware-based alone.

The structure of the paper is as follows. Section 2 recalls background information helpful for understanding this work. Section 3 presents the main case study used in this work, and the various translations and properties used. Section 4 overviews the experimental methodology. Section 5 considers the results of the experiments. Section 6 introduces a second case study for further experiments. Section 7 discusses the results and their implications. Section 8 explores how to combine software and hardware to improve fault injection vulnerability detection. Section 9 recalls related works and contrasts with the work presented here. Section 10 concludes.

2 Background

This sections recalls useful background information for understanding the rest of the paper. This includes an overview of the definition of fault injection and how to reason about fault injection by fault models and their typical classification. The two approaches (software and hardware) are both overviewed, along with background on how they are implemented and justification for the decisions made in this work.

2.1 Fault Injection

Fault injection is any modification at the hardware level which may change normal program execution. Fault injection can be environmental (e.g. background radiation, power interruption [6,31]) or intentional (e.g. induced EMP [16,33], rowhammer [41,47,56]).

Environmental fault injection is due to the environment the system is operating in [18,31] An example of this is one of the first observed fault injections where radioactive elements present in packing materials caused bits to flip in chips [6].

Intentional fault injection occurs when the injection is done by an *attacker* with the intention of changing program execution [33,41,47,56]. For example, fault injection attacks performed on cryptographic algorithms (e.g. RSA [13], AES [45], PRESENT [55]) where the fault is introduced to reveal information that helps in computing the secret key.

A fault injection *vulnerability* is a fault injection that yields a change to the program execution that is useful from the perspective of an attacker. This is in contrast to other effects of fault injection that are not useful, such as simply crashing a program, causing an infinite loop, or changing a value that is subsequently over-written. Observe that the definition of a vulnerability is not necessarily trivial or stable, the above example of a program crash may be a vulnerability if the attacker desires to achieve a denial of service attack.

One challenge in understanding and reasoning about fault injection is to be able to understand the effect that different kinds of faults can have upon the system that is affected. This requires some definition of how to characterise a fault and its behaviour in a manner that can be used experimentally.

2.2 Fault Model

There are a wide variety of approaches to reason about a *fault model* that describe how a system is affected by fault injection [52,39,32,25,42,17,21,20]. This section briefly overviews the key points regarding fault models as used in this work and their relation to the literature, however this does not include a comprehensive discussion of all possible fault model descriptions, approaches, or uses.

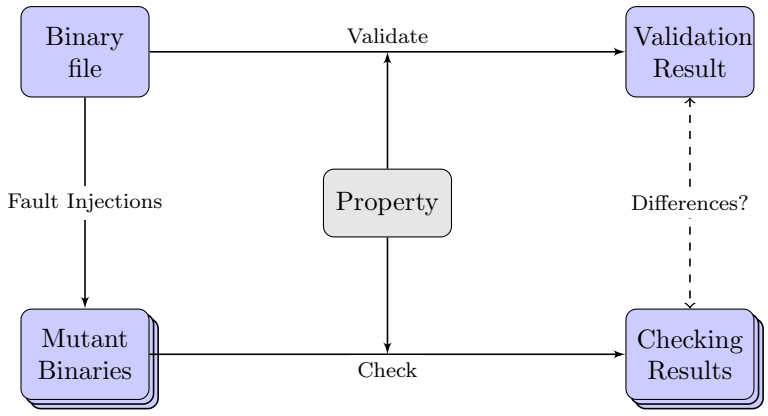

Fig. 1: Software Process Diagram

The goal of a fault model here is to describe how the system is affected by the fault injection attack in a manner that can be used for understanding or reasoning about the fault. Here the focus is on how the system is affected, and describing this in a manner that can be simulated and checked using the experimental methodologies considered. This means that the fault models considered here are typically related to modifications of the binary program, for example changing the value of a bit or byte within the program itself. These kinds of faults can be described as *compile time* faults [22,20] where the fault is persistent once injected into the program. Note that although this is described as "compile time", these faults can correspond to any modification that is persistent (as opposed to transient [58]). Compile time faults can correspond to faulting the binary before loading, while in memory, or for some parts, even faulting the cache. Also when the faulted part of the binary is only executed once, then compile time fault models correspond to transient fault models. Recent work has shown the effectiveness of persistent fault models in practice [35].

The fault models here are focused on describing the effect of the fault on the binary structure of the program. This is done to be able to contrast between software-based and hardware-based fault injection. Details of the fault models used here and their definitions appear in Section 4.1 when describing the experimental methodology.

2.3 Software Process

This section overviews the process used for software-based fault injection vulnerability detection. This process is adapted from previous work [21,20] where it was demonstrated to be effective.

An overview of the process as depicted in Figure 1 is as follows. The process starts with the binary file and the file of the properties to check upon the binary. The binary file is then translated to the modelling language for the model checker. The properties are validated to hold on the model using statistical model checking (SMC). The fault injection is then simulated on the binary file in order to produce mutant binaries. The models corresponding to mutant binaries are generated in the same manner as before. The properties are then checked upon the mutant binaries using SMC. A difference in the results of the validation and checking the property indicates a fault injection vulnerability created by the simulated fault injection and instance of the fault model.

Note that this process adapts from the process proposed by Given-Wilson et al. [21] in two ways. Firstly, here the properties are specified independently of the binary whereas in Given-Wilson et al.'s work the properties are embedded in the binary. Secondly, here SMC is used in place of Model Checking (MC).

The rest of this section overviews background on the formal methods used in the software based process used in the experiments here.

The overall process described above (and adapted from [21,20]) does not rely upon a particular formal method for checking and validating the properties. In [21,20] MC was used and produced results although there were limitations with the model checker used and some results proved to be inconsistent due to the model checker being bounded [20].

The underlying benefit of using MC is that all possible states of the system can be explored, and so the property absolutely proven to hold or not [24]. Unfortunately due to the complexities of modeling low-level machine behaviour and handling complexities related to memory, stack, and other architectural details, in practice only bounded model checking [9] was used in prior work [21,20].

To address these issues this work uses SMC that is able to operate efficiently over much larger and more complex spaces [28]. The underlying mechanism of SMC is to run a single trace through the model and check the properties on this trace. Many such traces are run, and statistical methods used to approximate whether the properties hold (or not) and with some level of certainty.

The main limitation of SMC is that SMC cannot ensure all possible states have been checked, and so cannot give absolute results (except when the state space is small). This means that very rare or hard to find problems may not appear in the results, and that absolute guarantees cannot be given unless the entire state space is checked (as in MC).

SMC has another advantage that is useful here: unknown values can be randomly chosen with no significant impact on the efficiency of the results. Here this means that if a previously unknown memory address is accessed, the SMC engine can assign a random value and continue efficiently. By contrast, MC would have to assume all possible values for this new (and previously unknown) word and so cause an enormous increase in the state space to explore. (This is another instance of the state space explosion problem of MC [24,28])

In both MC and SMC properties are used to define the correct or incorrect behaviour of the model. Here properties are used to define specific vulnerabilities that may be introduced by fault injection.

In this work the properties are specified using *Bounded Linear Temporal Logic* (B-LTL). B-LTL is chosen here for being able to represent the key concepts required and being compatible with the SMC tool used for the experiments here (see Section 4.1). The properties here are mostly specified using simple (in)equality relations, however the temporal and bounding operations can be exploited to account for infinite loops induced by fault injection. The above provides a high level overview of the key concepts. Further detail of how this is done in the context of this work is shown in Sections 3 & 4.1.

2.4 Hardware Process

This section overviews the hardware process used for detecting fault injection vulnerabilities. This process is common to many prior works [19,43,48,38].

An overview of the process is as follows. The first step is to experiment on the chosen target hardware with the chosen hardware fault induction technique. This step concludes when a configuration is found that allows the hardware fault injection to change program execution. The second step is then to load a program onto the target hardware that has a believed vulnerable point. The third step is to try and align the injected fault with the believed vulnerable point to demonstrate a fault injection vulnerability. A vulnerability has been demonstrated if the fault injection can change the program execution in the desired way with significant consistency.

3 Case Study: Control Flow Hijacking

This section presents the main case study used in this work. This control flow hijacking case study (CFH) [48,11] is chosen to have a known class of vulnerability that is straightforward to understand. The example here is presented in C source code, and assembly code for ARMv7-M Thumb-2. Finally, for the case study the correct, vulnerable, and incorrect program executions are defined.

Note that the case study contains *trigger* instructions that change the voltage of some pins observable to the hardware fault injection tools. These were used to improve precision in the hardware calibration as described in Section 4.2. However, no result in this work relies upon the existence of these triggers.

This case study is chosen to demonstrate a control flow hijacking vulnerability. The goal for the attacker is to output a specific value (`0x55555555`) that can only be reached by hijacking the control flow of the program execution.

The `test_persistence` function that is of interest is shown below.

```
uint32_t test_persistence (void){
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, GPIO_PIN_SET);
```

```
 uint32_t status = 0;
 if (pin_correct==1) {
   status=0xFFFFFFFF;
 } else {
   status=0x55555555;
 }
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, GPIO_PIN_RESET);
 return status;
}
```

Listing 1: C Code

Here the attacker wishes to hijack the function control flow to return `0x55555555` even when `pin_correct` has the value `1`. Since in the code being experimented on `pin_correct` always has value `1`, the program behaviour can be defined to be one of the following outcomes. The *correct* behaviour for this case study is to return `0xFFFFFFFF`. The program is *vulnerable* when the return value is `0x55555555` (achieved via some form of fault injection). Any other return value is considered to be *incorrect* program execution. Note that if the program does not terminate or does not provide a return value this is classified as *crashed*.

The corresponding ARMv7-M Thumb-2 assembly instructions for the `test_persistence` function are shown below.

```
08000aa0 <test_persistence>:
 8000aa0: b510 push {r4, lr}
 8000aa2: 480a ldr  r0, [pc, #40]
 8000aa4: 2180 movs r1, #128
 8000aa6: 2201 movs r2, #1
 8000aa8: f001 f91c bl 8001ce4
 8000aac: 4b08 ldr  r3, [pc, #32]
 8000aae: 4807 ldr  r0, [pc, #28]
 8000ab0: 681b ldr  r3, [r3, #0]
 8000ab2: 2180 movs r1, #128
 8000ab4: 2b01 cmp  r3, #1
 8000ab6: bf0c ite  eq
 8000ab8: f04f 34ff moveq.w r4, #0xFFFFFFFF
 8000abc: f04f 3455 movne.w r4, #0x55555555
 8000ac0: 2200 movs r2, #0
 8000ac2: f001 f90f bl 8001ce4
 8000ac6: 4620 mov  r0, r4
 8000ac8: bd10 pop {r4, pc}
 8000aca: bf00 nop
```

Listing 2: File Address Locations, Machine Instruction Codes, and Assembly Code

There are several instructions that are of significance to correct program execution. The instruction at `8000ab0` that loads to `r3` the value at memory address `[r3, #0]`. The instruction at `8000ab4` that compares the register `r3` with the value `#1`. Then the instruction at `8000ab8` that loads the value `0xFFFFFFFF` into the register `r4` if the prior condition is satisfied. Similarly the instruction at `8000abc` loads the value `0x55555555` into `r4` when the prior condition is not satisfied. Then the instruction at `8000ac6` that moves to the return register `r0` the return value from register `r4`. Observe that faulting any of these would have an effect on correct program execution. There may also be other less obvious ways to use fault injections to alter the behaviour of the program, the above are provided to illustrate (and are not exhaustive).

Since the SMC used here is unable to operate on the binary code (or assembly instructions), it is necessary to translate the binary code into a language that can be input to the SMC engine. Listing 3 shows a part of the *Reactive Model Language* (RML) model corresponding to the assembly code in Listing 2 (header information and variable declarations are not shown here).

Each binary (assembly) instruction in Listing 2 is translated to one or more lines in the RML model. The structure of the lines of the RML model is as follows, illustrated using the first line as an example.

– `pc=66562` is the *guard* in the RML language. This represents the predicate over all the variables in the model. Here `pc` represents the program counter and `66562` is the address of the instruction in the binary.
– `-> 1` is the *probability* of the transition. This express the probability that the model will take this transition. Note that here all probabilities are `1` due to deterministic definitions of the instructions [14].
– After the colon, `(pc' = 66564) & (r0' = M0)` is the *update* done by the transition if the guard is true. Here the value of `pc` will be updated to the address of the next instruction, and the register `r0` will receive the value of the memory address `M0`.

Observe that on lines 11 and 12 (and also 13 and 14) multiple lines have the same program counter address but with different guards. These represent the conditional operation of line 13 (and 14) of the instructions in Listing 2.

```
1  [] pc=66562 -> 1 : (pc' = 66564) & (r0' = M0);
2  [] pc=66564 -> 1 : (pc' = 66566) & (r1' = 128);
3  [] pc=66566 -> 1 : (pc' = 66568) & (r2' = 1);
4  [] pc=66568 -> 1 : (pc' = 66572) & (lr' = 66572);
5  [] pc=66572 -> 1 : (pc' = 66574) & (r3' = M1);
6  [] pc=66574 -> 1 : (pc' = 66576) & (r0' = M1);
7  [] pc=66576 -> 1 : (pc' = 66578) & (r3' = M0);
8  [] pc=66578 -> 1 : (pc' = 66580) & (r1' = 128);
9  [] pc=66580 -> 1 : (pc' = 66582) & (flag' = ( r3 = 1 ) ) ;
10 [] pc=66582 -> 1 : (pc' = 66584) ;
11 [] pc=66584 & flag=true -> 1 : (pc' = 66588) & (r4' = MAXVAL);
12 [] pc=66584 & !flag=true -> 1 : (pc' = 66588) ;
13 [] pc=66588 & flag=false -> 1 : (pc' = 66592) & (r4' = 1431655765);
14 [] pc=66588 & !flag=false -> 1 : (pc' = 66592) ;
15 [] pc=66592 -> 1 : (pc' = 66594) & (r2' = 0);
16 [] pc=66594 -> 1 : (pc' = 66598) & (lr' = 66598);
17 [] pc=66598 -> 1 : (pc' = 66600) & (r0' = r4);
18 [] pc=66600 -> 1 : (pc' = 66604) & (sp' = 8);
19 [] pc=66602 -> 1 : (pc' = 66604) ;
```

Listing 3: Control Flow Hijacking RML Model

Observe that in Listing 3 the stack memory locations are instantiated to representative memory cells, e.g. `M0` and `M1`. These are initialised according to the binary instructions, and if no initial value is declared then the SMC chooses a value at random.

The specified property for this case study will check the values of register `r4`. Under normal program behavior the value of register `r4` will eventually be `0xFFFFFFFF` (or `MAXVAL`). The property to check for vulnerability of this program is whether `r4` can eventually have the value `0x55555555` (which is equal to `1431655765`). The property is expressed in the BLTL form as follows:

```
F<=#1000 (r4=1431655765)
```

The property operates as follows.

– `F` is the temporal operator for eventually.
– `<=#1000` expresses the bound, it gives the length of the run (number of steps) on which the property must hold, since here the property is expressed in the bounded linear temporal language (BLTL).
– `(r4=1431655765)` checks if the output value could be 0x55555555. Note that under normal execution and valid pin this case cannot happen without any fault injection.

Another property that is checked it to ensure that when the program counter is at the return point from the function then the value of the return register is not the maximum value as shown below..

```
F<=#1000 (pc=66604 & r4!=MAXVAL)
```

Observe that this will also detect the vulnerability property above, while failing both properties indicates correct behavior of the program. Thus, there are three possible outcomes from the checking:

– *Vulnerable*: when only the vulnerability property holds.
– *Incorrect*: when the incorrect property holds and the vulnerability property does not hold.
– *Correct*: when neither property holds and (for the properties tested) the program exits normally.

4 Experimental Methodology

This section discusses the experimental methodology used to conduct the experiments in this paper. The overall methodology is as follows.

The first step is to take the case study, to perform extensive software fault injection simulations, and to use the software process (see Section 2.3) to identify as many potential vulnerabilities as possible. Incorrect program execution is also identified to help in later stages of the methodology. Finally, crashes and other failures of program execution are exploited for calibration as described later.

The second step is to perform hardware fault injections on the entire function and to identify which configurations yield statistically significant changes in program execution.

The third step is to compare the software and hardware results to: identify achievable fault injection vulnerabilities using the hardware results; identify likely hardware fault models using the software results; and to demonstrate that Software-Based Fault Injection and Hardware-Based Fault Injection techniques have a co-relation.

The rest of this section details the environment and implementation for the experiments.

4.1 Software

The software-based experiments were performed by an implementation of the process described in Section 2.3.

The implementation of the process presented in 2.3 can be seen in Figure 2. The implementation begins with a binary file for ARM-v7 architecture and the properties specified in B-LTL (in separate files).

The binary is translated to *Reactive Module language* (RML) using ArmL[1]. RML [26] is a state-based language based on the Reactive Modules formalism [2] and used as the input language for Plasma Lab [29]. The specified property is then validated to hold on the generated RML model using the SMC Plasma Lab [29]. The mutant binaries corresponding to simulated fault injections are generated using SimFI[2]. The RML models for the mutant binaries are generated using the ArmL tool. The properties are then checked on the mutant models using SMC with Plasma Lab. Finally the results of model checking the mutant models and the binary file model are compared for statistically significant differences. Note that as mentioned in Section 2.3, SMC cannot give absolute results without full enumeration of the whole space, and also previously unknown values are randomly instantiated. Note that it is possible for results to be different but not in a statistically significant manner (e.g. if detecting whether a property can be violated, detecting this 2 out of 100 runs or 1 out of 100 runs is unimportant, since the violation is demonstrated).

The properties (see Section 3) are written in a separate file with the extension `.bltl`. This file will be used by the model checker tool (Plasma Lab) to verify the generated mutants.

For software simulation various fault models can be simulated by SimFI. Since the Electro-Magnetic Pulse (EMP) used here (see Section 4.2 below) does not have a single consistent fault model [33], multiple fault models were considered here. The fault models tested here are as follows.

Z1B The *zero one byte* fault model (Z1B) simulates setting a single byte to zero (regardless of prior value). This fault model corresponds to a malicious attack that is commonly achievable attack in practice [53, 45].

Z4B The *zero four bytes* fault model (Z4B) represents setting four bytes to zero (again regardless of prior value). This is similar in concept to the Z1B fault model and attack, but captures behaviour more

[1] ArmL is a translation tool that translates from ARM-v7 binaries to RML models.

[2] The SimFI tool is a tool that simulates a wide variety of fault injection attacks on binaries. The tool takes a binary as an input (regardless of the binary's architecture). Based on the chosen fault model a mutant binary is generated, representing the simulation of the chosen fault injection attack.

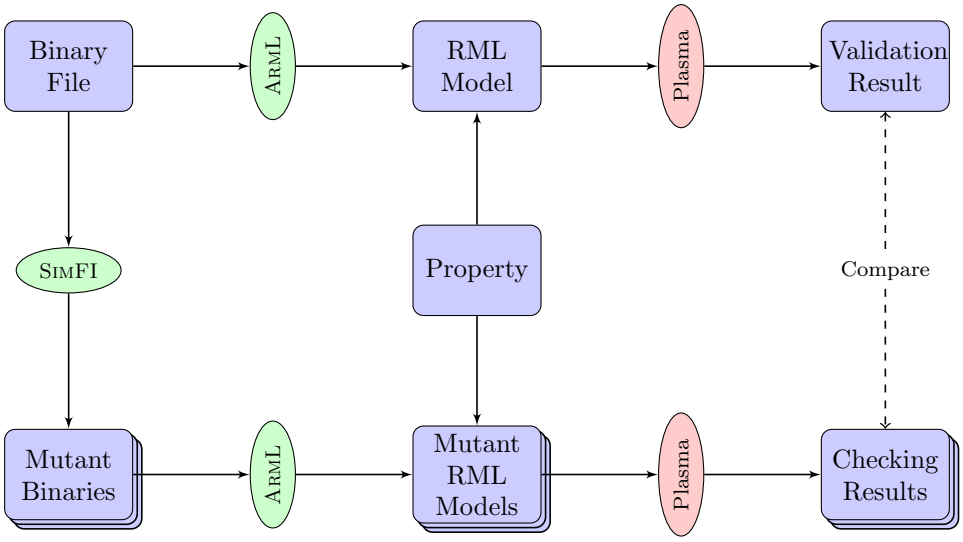

Fig. 2: Software Implementation Diagram

related to the hardware model, since it reflects faulting some piece of the hardware that operates on words rather than bits or bytes (such as the ARM Cortex-M3 bus used here) [14].

NOP The *ARM NOP* fault model (NOP) sets the targeted operation to a non-operation (NOP) instruction for the chosen architecture (in this case `0x00BF` for ARM-v7). The concept behind this model is that it simulates skipping an instruction, a common effect of many runtime faults [34].

FFB The *FF one byte* fault model (FFB) sets the value of byte to `0xFF`. This is opposite in concept to the Z1B fault model and attack, this may be an effect of EMP. The choice of using this here is to consider when an EMP may fault the chip with the opposite electromagnetic effect (i.e. set all bits to 1's instead of 0's).

FLIP The *flip* fault model (FLIP) simulates the flipping of a single bit, either from 0 to 1 or from 1 to 0. This fault model is highly representative of many kinds of faults that can be induced, ranging from those due to atmospheric radiation, to software effects such as the rowhammer attack [23].

The software simulation experiments were performed to simulate all listed fault models on all possible addresses within the target function of the case study. The outcomes were then classified as: correct, vulnerable, incorrect, or crashed as described in Section 3.

Observe that the Z4B and NOP affect multiple bytes (4 and 2 respectively). In practice the fault injection experiments here applied these fault models to all possible addresses in the binary. In practice this means that Z4B fault model faulted 4 bytes starting from the address targeted, i.e. faulting the bytes at the address X and then also $X+1$, $X+2$, & $X+3$. Similarly the NOP fault affects the byte following the targeted address.

The simulations were conducted on a virtual machine configured with one CPU, 11.7GB of RAM, and 179.4GB of disk space running Linux Ubuntu 16.04 LTS. The virtual machine was hosted on a Macbook Pro with 3.1 GHz Intel Core i7 processor, 16 GB of RAM, and running macOS High Sierra 10.13.3.

4.2 Hardware

The hardware process also follows the standard approach to Hardware-Based Fault Injection as presented in Section 2.4.

The chosen target hardware is a STM32 Value-Line Discovery (STM32VLDISCOVERY) STM32F100RB board with ARM Cortex-M3 core Micro Controller (μC) running at 24MHz.

The chosen fault injection induction method is to induce a fault via EMP. The EMP signal is initiated by a KEYSIGHT 33509B Waveform Generator that sends a signal through a KEYSIGHT 81160A Pulse Function Arbitrary Noise Generator (a high precision pulse generator that helps in the manipulation of the signal). The signal is then amplified using a MILMEGA 80RF1000-175 RF AMPLIFIER. Finally, it is sent to a Probe RF B 0.3-3 that is configured above the target hardware.

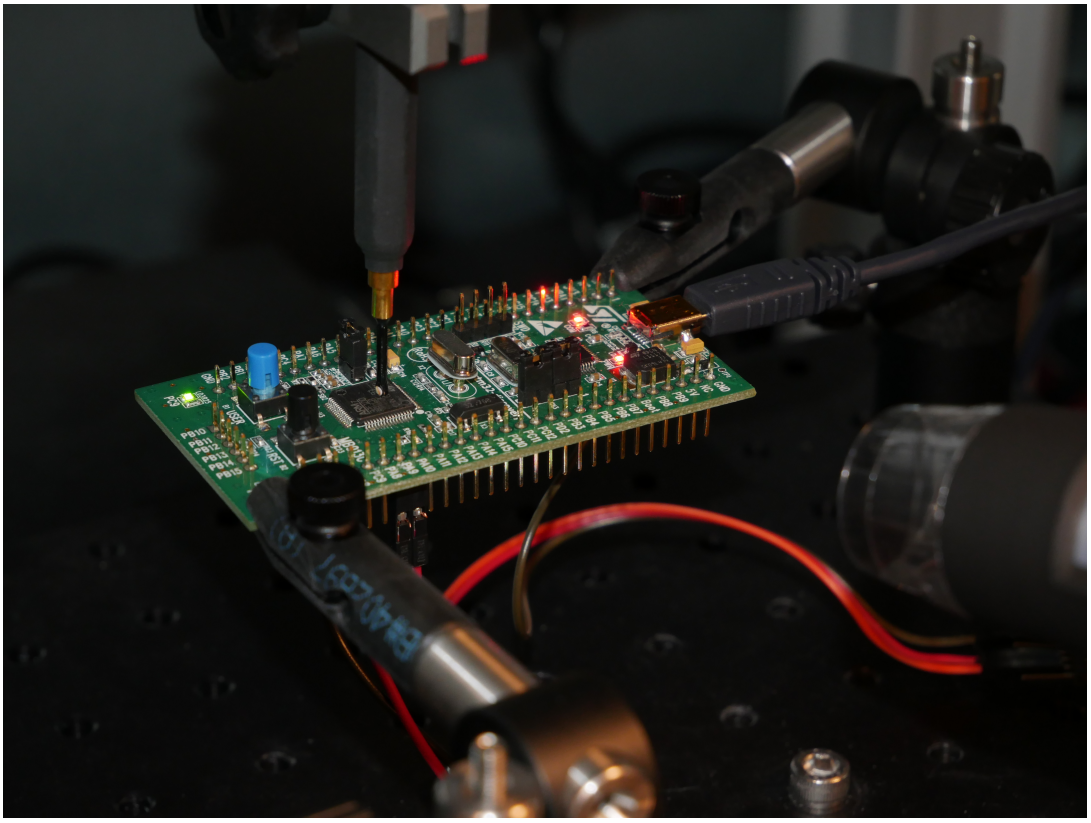

Fig. 3: Hardware-Based Fault Injection Probe Location

Initial experiments were then conducted to find a configuration that allowed consistent program execution disruption. In practice this was achieved by placing the probe above the chip as depicted in Figure 3.

Further experiments were conducted to calculate the latency of the various components. This allowed calculation of the timing between the injection of the fault and observing the effect. Further, this allowed calibration of the minimum and maximum possible delay between fault injection and observations of effects. The delay between the injection of the fault and the observed effect was 0.08μs to 0.12μs.

For the case study, the program was loaded onto the target hardware. The triggers were then used to calibrate the fault injection hardware tools and to verify the latency calculations were correct. Further, the minimum and maximum clock cycle[3] count was calculated for the case study functions (using the Cortex-M3 technical reference manual [14]). These were then used to find the earliest start point and latest end point of execution of the functions being considered (including a margin of error to ensure complete coverage).

Once the bounds of the execution had been calculated, hardware faults were injected at 4ns intervals starting from the earliest possible start point to the latest possible end point. The results as described in Section 3 for each execution and fault injection are then recorded. This is then repeated 200 times to gain significant information on the effects at each timing points. (This last step is done to account for minor inconsistencies in effects, and due to the general imprecision of EMP faults, as well as due to fault injection vulnerabilities not being achievable with high reliability in practice.)

4.3 Bridging Software And Hardware

This section shows how to bridge the software based and hardware based approaches (Sections 4.1 & 4.2 above) and then compare the results.

[3] Each clock cycle is approximately 40ns.

This comparison was done for each fault model from the software experiments with the results from the hardware experiments. The number of clock cycles were calculated (up to the fault injection point, since after this the results may be perpetuated), and then used to cross-reference with the address of the fault from the software experiments. Then, the alignment of the clock cycles were varied to see if there was a strong transition point where the hardware clearly changed from one instruction to another (since the clock cycles are not perfectly aligned, and the hardware experiments injected many faults at different times within each clock cycle's length).

The above comparison was also performed for combinations of fault models, and for subsets of fault models. Each combination of fault models was compared to see if multiple fault models combined matched well with the hardware experiments. Similarly, subsets of the results within fault models were used for some fault models. The Z1B and NOP in particular were tested with subsets of their results that considered only being applied to: every second byte (i.e. at the start or end of many instructions), to every fourth byte (i.e. at the start or end of many words), and to the first or second byte of every instruction (i.e. which can be two or four bytes since the instruction lengths vary).

5 Results

This section presents the results from the experiments. This includes: the results of the software simulation experiments alone; the results of the hardware experiments alone; and relations between both experiments.

5.1 Software

A detailed table of all the results can be seen in Figure 4. The byte address is the byte targeted by the fault injection, the value is the byte value, the assembly instruction is the instruction containing the targeted byte and the columns remaining columns refer to the fault model. A vulnerability is denoted by Vul and an incorrect result by Inc for all fault models, except FLIP where the red numbers indicate vulnerability at the bit within the byte and blue indicates incorrect results. A more compact overview is also presented in Figure 5. (The red coloured bytes ■ indicate the presence of vulnerabilities and the blue coloured bytes ■ indicate the presence of incorrect results, absence of any colour indicates correct behaviour.) Observe that all fault models indicated some vulnerabilities between bytes `800aad` and `8000ab8` . Additionally the FLIP fault model indicated a vulnerability earlier at byte `8000aa8`. Incorrect results were detected from byte `800aab9` to byte `8000abd` by all fault models except FFB. All fault models indicated vulnerabilities between bytes `8000ab0` & `8000ab1`, and `8000ab4` & `8000ab5`. However, there was no consensus on where the incorrect results of execution would appear amongst all the fault models (or even only the fault models that had incorrect results).

The instruction `ldr r3, [r3, 0]` at byte address `8000ab0` in Listing 1, loads the value of the variable `pin_correct` into the register `r3`. The simulation of a fault using the various fault models produces the following effects. Using all the fault models (except for NOP), one can change the `LDR` instruction to `MOV`, `CMP` or `STR` instruction. Using all the fault models, it is possible to change where the value of `pin_correct` from register `r3` to a different register (e.g. `r0`, `r7` or `r2`). Using the FLIP fault model, it is possible to modify the memory address from where the value will be loaded, yielding an unknown (or effectively random) value for `pin_correct`. Using the NOP fault model, it is possible to replace the instruction with a `NOP` instruction and so the value of `pin_correct` is implicitly set to whatever was in `r3` prior to this point in execution. All the above effects will not set the register `r3` to the correct value of the variable `pin_correct` and so affect the comparison done later on line 11 in Listing 1.

The instruction `cmp r3, 1` at byte address `8000ab4` in Listing 1 compares the value of the register `r3` with `1`, and updates the corresponding flags of the Application Program Status Register (APSR) based on the result of the comparison. The simulation of a fault using the fault models produce the following effects. Using the Z1B, Z4B and FLIP fault model, it is possible to change the `CMP` instruction to `MOV`, `ADD` or `LDR` instruction. Using all the fault models, it is possible to change the value of the number `1` the instruction compares with the register `r3`. Using the NOP fault model, it is possible to replace the instruction with a `NOP` instruction. The above fault model's modification will effect the comparison of

byte address	value	assembly instruction	Z1B	Z4B	NOP	FFB	FLIP
8000aa0	b5	push {r4, lr}					
8000aa1	10						
8000aa2	48	ldr r0, [pc, #40]					
8000aa3	0a						
8000aa4	21	movs r1, #128					
8000aa5	80						
8000aa6	22	movs r2, #1					
8000aa1	01						
8000aa8	f0	bl 8001ce4					0,3
8000aa9	01						
8000aaa	f9						
8000aab	1c						
8000aac	4b	ldr r3, [pc, #32]					
8000aad	08			Vul	Vul		
8000aae	48	ldr r0, [pc, #28]		Vul		Vul	
8000aaf	07			Vul	Vul		
8000ab0	68	ldr r3, [r3, #0]	Vul	Vul	Vul	Vul	1,2,4,5,6,7
8000ab1	1b		Vul	Vul	Vul	Vul	0,1,5,6,7
8000ab2	21	movs r1, #128		Vul			6
8000ab3	80			Vul			
8000ab4	2b	cmp r3, #1	Vul	Vul	Vul	Vul	0,1,2,3,4,5,6,7
8000ab5	01		Vul	Vul	Vul	Vul	0,1,2,3,4,5,6,7
8000ab6	bf	ite eq	Vul	Vul	Vul		0,2,3,5,7
8000ab7	0c		Vul	Vul		Vul	1,2,3,4,5
8000ab8	f0	moveq.w r4, #0xFFFFFFFF	Vul	Vul	Vul		0,1,2,3,5
8000ab9	4f		Inc	Inc			1
8000aba	34			Inc			1
8000abb	ff				Inc		1,2,3,4,5,6,7
8000abc	f0	movne.w r4, #0x55555555	Inc		Inc		0,1,2,3
8000abd	4f		Inc				1
8000abe	34						
8000abf	55						
8000ac0	22	movs r2, #0					
8000ac1	00						
8000ac2	f0	bl 8001ce4					
8000ac3	01						
8000ac4	f9						
8000ac5	0f						
8000ac6	46	mov r0, r4					
8000ac7	20						
8000ac8	bd	pop {r4, pc}					
8000ac9	10						
8000aca	bf	nop					
8000aca	00						

Fig. 4: Detailed Software-Based Fault Injection Control Flow Hijacking Results

the register `r3` value with `1`, which will impact the choice of the correct branching in the following three instructions.

The instruction `ite eq` on byte address `8000ab6` in Listing 1 defines the APSR flags to set to be used by the following two instructions. The simulation of a fault using the fault models produce the following effects. Using all the fault models (except FFB), it is possible to change the `IT` instruction to `MOV`, `ADD` or `LDR` instruction. Using the Z1B, Z4B and NOP fault models, it is possible to replace the instruction with a `NOP` instruction. Using the FFB and FLIP fault model, it is possible to change branching order the processor follows. All of these can yield changes to the APSR flags that will in turn alter the effects of the following two instructions. In general, the alterations allow the branching behaviour to be inverted, and thus yield an effective hijack of the control flow.

The instruction `moveq.w r4, #0xFFFFFFFF` on byte address `8000ab8` in Listing 1 sets the return register `r4` to the value `0xFFFFFFFF`. The simulation of a fault using all fault models (except FFB) produce the following effects. Using the Z1B, Z4B and FLIP fault model, it is possible to change the `MOV` instruction to `ADD`, `STR` or `LDR` instruction. Using all except the FLIP fault model, it is possible to replace the instruction with a `NOP` instruction. The above fault models modification will not set the

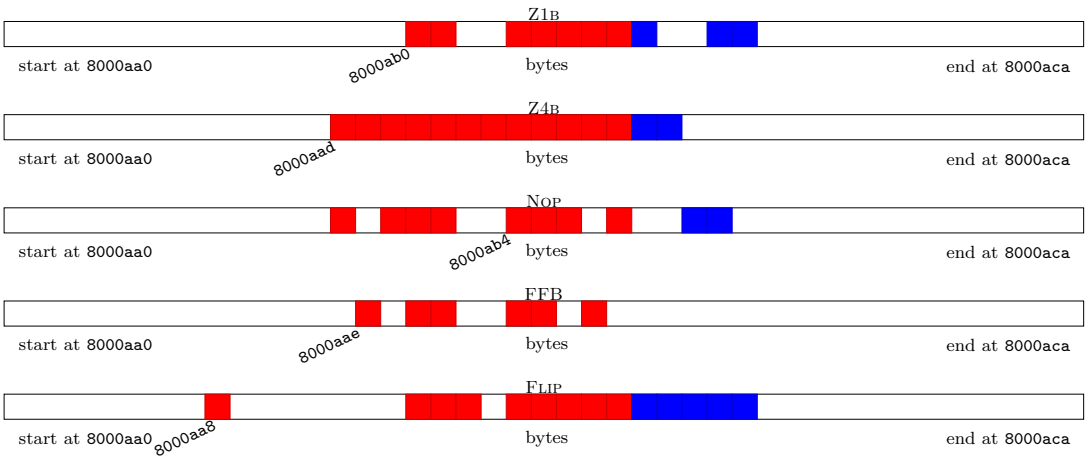

Fig. 5: Overview of Software-Based Fault Injection Control Flow Hijacking Results

	Control Flow Hijacking		
Fault Model	Mutants Number	Vulnerabilities Detected	Runtime
Z1B	44	7	2m14s
Z4B	44	9	1m52s
Nop	44	7	2m39s
FFB	44	3	1m42s
Flip	352	40	16m51s

Table 1: Experiment Runtime for Software-Based Fault Injection Control Flow Hijacking

return register to the correct value. So the returned value will be whatever was already in the register `r4` generally yielding an incorrect result.

Runtime information The experiment runtime is similar for each fault model except for the Flip fault model. For the Z1B, Z4B, Nop and FFB fault models the runtime ranged between 1 minutes and 42 seconds and 2 minutes and 39 seconds, to verify all the corresponding generated mutants (see Table 1). The Flip fault model experiment runtime was significantly longer with 16 minutes and 51 seconds due to the greater number of mutants to verify. Overall, the verification of a single mutant requires approximately 3 seconds. Note that the runtime information corresponds to the verification step using the Plasma tool, the fault injection simulation step time is insignificant.

5.2 Hardware

This section overviews the results of the hardware experiments. Using the calculations described in Section 4.2 the earliest possible start time for the `test_persistence` was calculated to be 0.8μs, and the latest possible end time to be 2.084μs. The hardware experiments were thus conducted within this range.

An overview of the results of the hardware experiments for the control flow hijacking case study can be seen in Figure 6. Observe that vulnerabilities were grouped together in two groups. The larger group between 1.192μs and 1.388μs, and the smaller group from 1.448μs to 1.492μs. The incorrect results are in three groups: one from 1.308μs to 1.348μs, another from 1.424μs to 1.472μs, and a third from 1.692μs to 1.744μs. There is also a single spike of incorrect results at 1.948μs.

Combining the known timing information with the clock cycle count for each instruction (from the Cortex-M3 technical reference manual [14]), it is possible to approximate which instructions are being loaded and executed at each fault injection timing. Note that for this particular ARM architecture the processor fetches 32 bits at a time, which means that for a 16 bit instruction the processor will fetch two instructions at a time. From all the above information the hardware fault injection vulnerabilities in Fig. 6 can be mapped to the addresses in Listing 1.

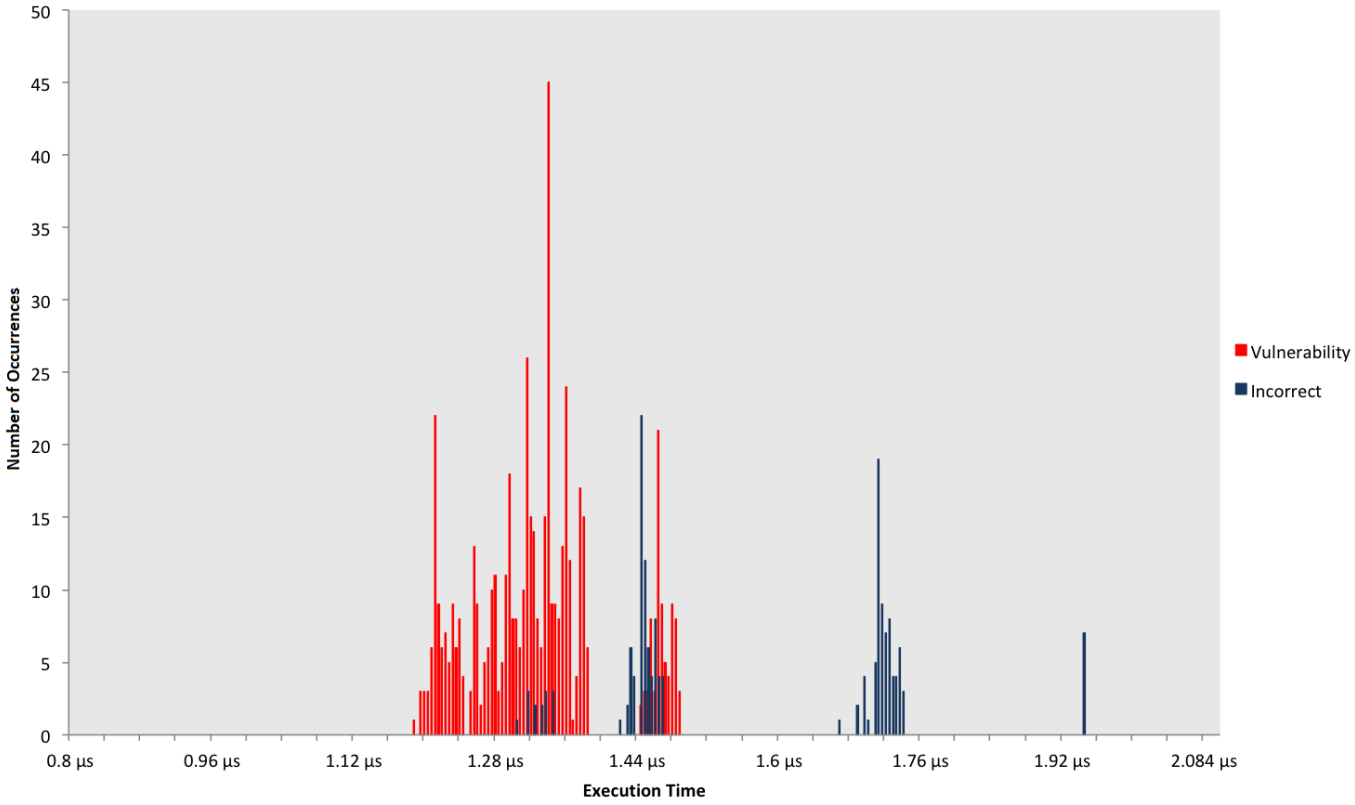

Fig. 6: Hardware-Based Fault Injection Control Flow Hijacking Results

The vulnerability detected between 1.192μs and 1.232μs corresponds to the `ldr r3, [pc, #32]` instruction at byte address `8000aac` in Listing 1. The vulnerability detected between 1.236μs and 1.312μs corresponds to the instructions `ldr r0, [pc, #28]` and `ldr r3, [r3, #0]` at byte address `8000aae` and `8000ab0`. The incorrect result in 1.424μs to 1.472μs corresponds to the instructions `movs r1, #128` and `cmp r3, #1` at byte address `8000ab2` and `8000ab4`. The vulnerability detected between 1.448μs and 1.492μs corresponds to the instructions `ite eq` and `moveq.w r4, #0xFFFFFFFF` at byte address `8000ab4` to `8000ab8`. The incorrect result in 1.672μs to 1.948μs corresponds to the instructions `mov r0, r4` and `pop {r4, pc}` at byte address `8000ac6` and `8000ac8`.

Runtime information For the hardware experiments, the runtime is related to: how long the program takes to execute, how long it takes to reset the devices, and the number of times to perform the fault injection. For the results here 59620 experiments were performed (after calibration). The runtime for the experiments (again, not including calibration) was approximately 5 hours. Due to re-calibration, occasional intervention, and other factors the time taken is not exact, however the average runtime is less than 0.3 seconds per experiment. Although each experiment is very fast, many need to be performed to gain significant results, here yielding $\sim$ 60 seconds per fault timing, significantly slower than the software-based process.

5.3 Comparison

This section compares the results of the software based and hardware based fault injection experiments presented in the previous two sections (5.1 & 5.2). Note that for brevity detailed comparison is omitted here although an attempt to align the results of the previous two sections is shown in Fig. 8. The rest of this section discusses the more interesting observations.

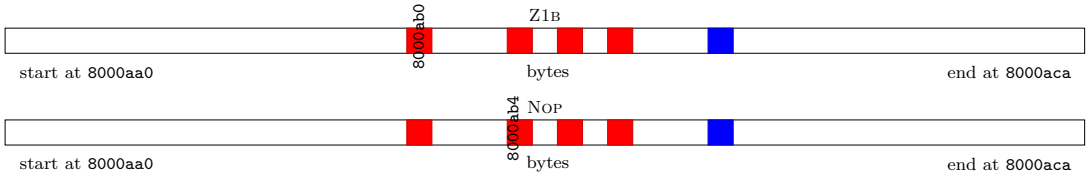

Fig. 7: Software-Based Fault Injection Control Flow Hijacking Results Only First Byte Instruction Results

byte address	value	assembly instruction	Z1B	Z4B	Nop	FFB	Flip	Hardware
8000aa0	b5	push {r4, lr}						
8000aa1	10							
8000aa2	48	ldr r0, [pc, #40]						
8000aa3	0a							
8000aa4	21	movs r1, #128						
8000aa5	80							
8000aa6	22	movs r2, #1						
8000aa1	01							
8000aa8	f0	bl 8001ce4					0,3	
8000aa9	01							
8000aaa	f9							
8000aab	1c							
8000aac	4b	ldr r3, [pc, #32]						Vul
8000aad	08			Vul	Vul			Vul
8000aae	48	ldr r0, [pc, #28]		Vul		Vul		Vul >Inc
8000aaf	07			Vul	Vul			Vul
8000ab0	68	ldr r3, [r3, #0]	Vul	Vul	Vul	Vul	1,2,4,5,6,7	Vul
8000ab1	1b		Vul	Vul	Vul	Vul	0,1,5,6,7	Vul
8000ab2	21	movs r1, #128		Vul			6	Vul
8000ab3	80			Vul				Vul
8000ab4	2b	cmp r3, #1	Vul	Vul	Vul	Vul	0,1,2,3,4,5,6,7	Inc > Vul
8000ab5	01		Vul	Vul	Vul	Vul	0,1,2,3,4,5,6,7	Inc > Vul
8000ab6	bf	ite eq	Vul	Vul	Vul		0,2,3,5,7	Inc > Vul
8000ab7	0c		Vul	Vul		Vul	1,2,3,4,5	Inc > Vul
8000ab8	f0	moveq.w r4, #0xFFFFFFFF	Vul	Vul	Vul		0,1,2,3,5	Vul > Inc
8000ab9	4f		Inc	Inc			1	Vul > Inc
8000aba	34			Inc			1	Vul > Inc
8000abb	ff				Inc		1,2,3,4,5,6,7	Vul > Inc
8000abc	f0	movne.w r4, #0x55555555	Inc		Inc		0,1,2,3	
8000abd	4f		Inc				1	
8000abe	34							
8000abf	55							
8000ac0	22	movs r2, #0						
8000ac1	00							
8000ac2	f0	bl 8001ce4						
8000ac3	01							
8000ac4	f9							
8000ac5	0f							
8000ac6	46	mov r0, r4						Inc
8000ac7	20							Inc
8000ac8	bd	pop {r4, pc}						Inc
8000ac9	10							Inc
8000aca	bf	nop						
8000aca	00							

Fig. 8: Detailed Combined Control Flow Hijacking Results

Overall observe that both approaches detected vulnerabilities in the instructions (starting) at byte addresses `8000aad`, `8000aae`, `8000ab0`, `8000ab4`, `8000ab6`, and `8000ab8` in Listing 1. However, no fault was detected by the hardware prior to `8000aad` (implying the Flip fault injection vulnerabilities here could not be observed in the hardware-based fault injection experiments).

Note that some "incorrect" results do not appear to co-relate since they are over-shadowed by vulnerable results. As presented in Section 3, the incorrect results were only shown when a vulnerability is not detected. For this reason co-relations on incorrect results are hidden when there is a vulnerability.

However, the FFB fault model did not indicate any incorrect results anywhere (implying that the FFB fault model may not be accurate representations of EMP effects).

Observe that since although all the fault models detected vulnerabilities in some of the same areas as the hardware experimental results, the above implications suggest that the FFB and Flip models do not appear to describe the effects of EMP accurately. This leaves the setting of byte(s) to zero (Z1B and Z4B) and skipping instructions (Nop) as the best fit between the software based results and the hardware based results.

The Z1B fault model matches quite well with having two groups of vulnerabilities, as well as two groups of incorrect results. This corresponds closely to the hardware results that also have two distinct groups of vulnerabilities, and of incorrect results (a third less clear group of incorrect results also exists).

The Z4B fault model matches well with the vulnerable results, but also has vulnerable results that are not confirmed by the hardware. That said, the faulting of a whole word tends to produce vulnerabilities that occur due to the faulting of a particular byte, that is the Z4B fault model in many cases induces the same fault as the Z1B by setting a following byte at a later address to zero. Thus, the lack of gaps in the vulnerabilities and the lack of a second group of incorrect results implies that while there is some co-relation, the Z4B fault model does not match the EMP effects well.

The Nop fault model is similar to the Z1B fault model in having groups of vulnerabilities that match very well with the hardware experiments. The lack of two groups of incorrect results however implies that the Nop fault model does not accurately represent the EMP effect on the hardware. Considering combinations and subsets of the fault models is straightforward from the above results and for the Z1B and Nop fault models applied only to the first byte of each instruction those displayed in Fig. 7. Observe the two fault models now matches but that no single fault model alone exactly matches the hardware results. Considering the Z1B and Nop instructions combined (or combined, but taking only the Nop targeting the first byte of each instruction) provides the closest match to the hardware results.

6 Additional Case Study: Backdoor

This section presents an additional case study with a weakness designed to be exploitable by fault injection and not detectable by code analysis. Section 6.1 introduces the code and weakness. Section 6.2 presents the software experimental results for the backdoor case study. Section 6.3 highlights the hardware experimental results. Section 6.4 overviews the comparison of the software and hardware results. Note that the experimental methodology here is the same as in Section 4.

The choice to consider this additional case study was to both consider more complications in the software-based process, and to attempt a different kind of hardware disruption. The further complications in the software-based process come from needing to handle multiple functions and movement between them. The hardware disruption is also different since here the expected fault model to achieve a vulnerability is an instruction skip (not altering values as in the control-flow hijacking case study).

6.1 Backdoor attack

This section recalls the Fault Activated Backdoor program from [48]. The core of the weakness in the code is a `backdoor` function (shown in Listing 4) that is hidden in the program but cannot be reached by any execution path. The normal behaviour of the program includes encryption with AES [51] yielding a ciphertext. The `backdoor` function (when executed) replaces the ciphertext with the AES key, thus allowing an attacker to observe the "ciphertext" and in practice learn the key. However, under normal conditions the `backdoor` function can never be executed, and so will should not be detected by static or dynamic code analysis.

The weakness here is built into the code in the `blink_wait` function shown in Listing 4. The value of `wait_for` is defined to be `3758874636`, which has two special properties. Firstly, this value is too large to be loaded within a single ARMv7-M Thumb-2 instruction and so the value is stored as a separate word in the assembly code. Secondly, this value if interpreted as an instruction corresponds to a jump to a specific location (in practice the location of the `backdoor` function).

```
void blink_wait(){
  unsigned int wait_for=3758874636;
  unsigned int counter;
  for(counter=0;counter<wait_for;counter+=8000000);
}
void backdoor(void) {
  int i;
  for(i = 0; i < DATA_SIZE; i++){
     ciphertext[i] = key[i];
  }
  HAL_GPIO_WritePin(LED3_GPIO_PORT, LED3_PIN, GPIO_PIN_SET);
}
```

Listing 4: Backdoor Case Study C code

The corresponding assembly code for the `blink_wait` function is shown in Listing 5. Observe that the value of `wait_for` is stored at the end of the function at address `80005cc` immediately after the `POP` instruction at address `80005ca`. Thus, an attacker that can cause this `POP` instruction to be skipped or interpreted as something else (e.g. a `MOV`, `ADD` or `LDR` as observed in Section 5.1) would then execute this value as a jump to the `backdoor` function.

```
08000598 <blink_wait>:
8000598: b580 push {r7, lr}
800059a: b082 sub sp, #8
800059c: af00  add r7, sp, #0
800059e: 4b0b ldr r3, [pc, #44]
80005a0: 603b st r3, [r7, #0]
80005a2: 2300 movs r3, #0
80005a4: 607b str r3, [r7, #4]
80005a6: e005 b.n 80005b4
80005a8: 687b ldr r3, [r7, #4]
80005aa: f503 03f4  add.w r3, r3, #7995392
80005ae: f503 5390 add.w r3, r3, #4608
80005b2: 607b str r3, [r7, #4]
80005b4: 687a ldr r2, [r7, #4]
80005b6: 683b ldr r3, [r7, #0]
80005b8: 429a cmp r2, r3
80005ba: d3f5  bcc.n 80005a8
80005bc: f7ff ffe2 bl 8000584
80005c0: 2003 movs r0, #3
80005c2: f000 f8af bl 8000724
80005c6: 3708 adds      r7, #8
80005c8: 46bd mov sp, r7
80005ca: bd80 pop {r7, pc}
80005cc: e00be00c
```

Listing 5: Backdoor Case Study Assembly

For the software experiments the *correct* behaviour is to never enter the `backdoor` function and the *vulnerable* behaviour is any entry into the `backdoor` function. (Note that this precludes *incorrect* results appearing since the ciphertext is not modified in the `blink_wait` function.) For the hardware experiments the *correct* behaviour is to output the ciphertext as usual, *vulnerable* behaviour is to output the key in the place of the ciphertext, and *incorrect* behaviour is to output some other value. (Note that incorrect behaviour was never observed in the experiments.)

The choice to operate on the behaviour for the software on entering the `backdoor` function rather than output was to detect any possible exploit that allows access to the hidden code, since the code inside can be padded or modified to handle different access paths. That no incorrect results can be detected is not interesting for the software experiments since these mostly indicate a fault that would store a value at an incorrect address.

6.2 Software Experiment Results

An overview of the backdoor case study software experiment results can be seen in Figure 9. Observe that all the fault models indicated possible vulnerabilities around bytes `80005ca`. The Flip fault model indicated in addition vulnerabilities at the byte `80005a7`.

The vulnerabilities detected at byte `80005ca` by all but one fault model correspond to the `POP` instruction at `80005ca` in Listing 5.

The instruction `bd80 pop {r7, pc}` at byte address `80005ca` in Listing 5, stores the top value of the stack into registers `r7` and `pc`. This instruction indicates the end of the `blink_wait` function. The simulation of the fault injection using the fault models produces the following effects. Using all the Z1B, Z4B, and Flip fault models, it is possible to change the `POP` instruction to `LSL`, `MOV`, `ADD`, ... instructions. Using the Nop fault model, it is possible to replace the instruction with a `NOP` instruction. Using the Flip fault model, it is possible to modify the registers that will be modified after the pop. Here instead of loading values of the stack into registers `r7` and `pc`, it will only load the value into register `r7`. All the above modifications will skip the execution of the `POP` instruction, and so execute the `wait_for` value corresponding to a branching instruction to the `backdoor` function.

An interesting vulnerability which was detected at byte `80005a7` by the Flip fault model, corresponds to the instruction `b.n 80005b4` at `80005a6` in Listing 5. This instruction is a branching instruction, which will jump to the instruction at `80005e8` in Listing 5. The effect of (simulated) fault injection using the Flip fault model was to change the target address of the branch directly to the `backdoor` function.

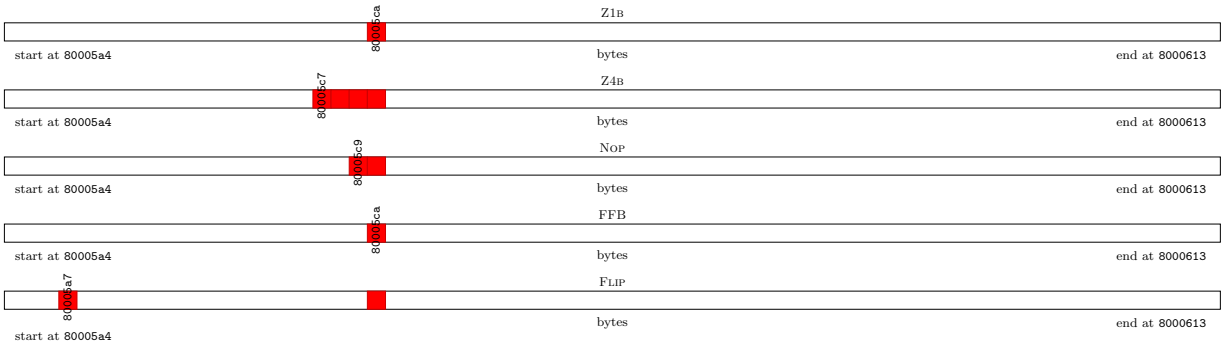

Fig. 9: Software-Based Fault Injection Backdoor Results

6.3 Hardware Experiment Results

An overview of the backdoor case study hardware experiment results can be seen in Figure 10. As before (see Section 5.2) various measurements and experiments were performed to ensure the correct timing for the fault injection, and 200 experiments were run to yield the results. Observe that the only vulnerabilities were detected between 1.224μs and 1.260μs.

By calculating the execution to for the instructions, clock cycles, hardware latency, etc. the fault injection at time 1.224μs to 1.260μs corresponds to the `POP` instruction at `80005ca` in Listing 5.

6.4 Comparison

This section compares the results of the software based and hardware based fault injection experiments from the previous two sections (6.2 & 6.3). Both approaches detected vulnerabilities in the instruction at byte addresses `80005ca` in Listing 5.

Due to the very limited hardware results (only a single spike of vulnerabilities and no incorrect results), the comparison is both trivial and less interesting. All the fault models were able to detect a vulnerability in the instruction at byte addresses `80005ca` in Listing 5.

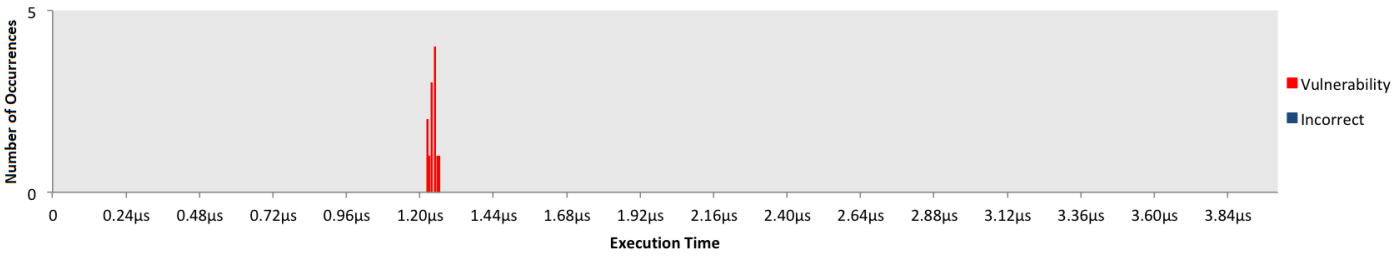

Fig. 10: Hardware-Based Fault Injection Backdoor Results

The Z1B and FFB fault models detected a fault injection vulnerability at the exact same address as the hardware approach and nowhere else. The NOP fault model also found a fault injection vulnerability at `80005c9` since the NOP fault model changes the value of two bytes and so will impact the instruction at byte address `80005ca`. The Z4B fault model found faults at four byte addresses `80005c7` to `80005ca`, but in practice this was merely due to the size of the fault model, since all Z4B faults starting from `80005c7` set the byte `80005ca` to zero. The FLIP fault model was the only one to have a significant difference also finding a fault injection vulnerability in the instruction at byte address `80005a7` in Listing 5.

Although this case study was small and had limited results some new information was still gained. Further evidence for the lack of co-relation between the FLIP fault model and EMP effects aligns with the results of Section 5.3. The additional case study also demonstrated that both software-based approach can operate over function boundaries, something that was not tested in the control flow hijacking case study. Finally, the NOP fault model *can* co-relate with instruction skip behaviour in practice.

7 Discussion of Results

This section discusses the experimental results and what we can learn from them.

By comparing the software and hardware experimental results it is possible to determine which software fault models best correspond to the EMP effects observed. Here the Z1B, Z4B and NOP fault models had the closest correlation with the observations of the EMP faults induced. To some extent this agrees with previous work [33] that observed that the most accurate fault model is an instruction skip (or here NOP). However, there is also strong evidence from this work that other fault models, in particular setting all of a byte or word to zeros (i.e. Z1B or Z4B), also correlate strongly with the effects of EMP.

Observe also that the software vulnerabilities generated using the FLIP and FFB fault models did not correspond well to the EMP fault injection results. Although the FLIP and FFB fault models detected some similar vulnerabilities to the other fault models, both fault models also detected a lot of vulnerabilities which do not correspond to the hardware results. In particular the FFB fault model never produced an incorrect result despite many being observed, and the FLIP fault model had many vulnerable or incorrect results that did not correlate with the EMP results.

Using the software results to learn about the hardware results is also possible. The hardware experiment results do not indicate *how* the fault was achieved or what the actual fault model/effect was, only the outcome. Knowing the specific effect of the fault injection on the hardware is a nontrivial task, specially when using imprecise hardware techniques such as EMP. Hardware experiment results alone are able to show that the injection of the fault create the desired vulnerability, but do not give detailed information of what, where, or how the injected fault created the vulnerability. The results here indicate that the strongest correlation is with instructions simply being faulted to have alternate or no effect (i.e. the Z1B, Z4B, and NOP fault models). Further, since none of these fault models correlates exactly, this implies (along with the inconsistent nature of achieving a vulnerable or incorrect outcome) that EMP fault effects may vary and not have a single fault model.

From the experiment results one can observe that the hardware and software results do co-relate but they do not exactly match. There are clearly locations in the assembly code where many fault models indicate a vulnerability (or incorrect result) and these correlate very strongly with the locations where

the hardware experiments were able to produce vulnerabilities (or incorrect results, respectively). This clearly indicates that there is a co-relation between the software based and hardware based approaches.

Considering the results further, one key insight is that the *software based experiments did not have any false negatives*. That is, every place where the hardware was able to produce a genuine vulnerability (or incorrect result), the software based approaches indicated a vulnerability (or incorrect result, respectively) for at least one fault model. (Indeed, this holds even when only considering the Z1B, Z4B, and NOP fault models.) Thus, absence of any vulnerabilities or incorrect results according to software based experiments implies that no such vulnerabilities or incorrect results should exist in practice.

The software based approach does produce false positive results. This outcome is not surprising since many fault models were tested here, including ones unlikely to be possible with the hardware based EMP fault injection. However, even when considering only the Z1B, Z4B, and NOP it is not clear that *every* vulnerability or incorrect result can be reproduced by the EMP experiments. The conclusion here is that software-based simulations can find vulnerabilities (or other behaviours) that may be infeasible to reproduce in the hardware, or at least extremely difficult to achieve.

From all the above one can conclude that: on one hand software alone is not sufficient to claim that vulnerability exist and is real, on the other hand hardware alone is not feasible to explore all the possible configurations and locations in the target program. That is, the software can be quickly used to find many potential vulnerabilities (or other results), but that these cannot be guaranteed to exist in practice. The hardware can guarantee a vulnerability (or other outcome) when one is produced, but finding these is extremely expensive in time and equipment, and this may be infeasible on larger programs.

8 Combined Approach

The natural extension of these hardware and software results is to consider how they could be combined. This section discusses how this can be achieved to rapidly find genuine vulnerabilities that would be infeasible with either approach alone. Observe that this approach does *not* rely upon any prior knowledge of weaknesses in the code. If only the software-based approach is used then although the results are quick to compute and require only a moderate amount of computational resources, there is no guarantee that any of the results hold. Indeed, attempting to address too many false positives would be intensive on developer resources and a waste of effort if the vulnerabilities are not genuine.

If only the hardware-based approach is used this is extremely expensive if not infeasible to test larger programs. This requires many experiments to test each possible timing/location of fault injection on the program over the program's entire execution life-cycle.

The proposed combined approach is to use the software-based simulations to quickly find all the *potential* vulnerabilities in a given program. This can be easily applied and automated [21,20] to yield information on all the locations in the code that may be vulnerable. The hardware-based approach can then be applied to test the most vulnerable locations to rapidly confirm (or refute up to some margin of confidence) the existence of the vulnerability. In practice this requires some small amount of computational resources for the simulations, and then only limited time and some calculation prior to testing with the hardware to accurately target the right locations.

The rest of this section explores how the above combined approach could be applied to the case studies here, and demonstrates the efficacy of the combined approach.

For the control flow hijacking case study, ~ 59620 hardware experiments were conducted to generate the results shown in Fig. 6. (This number accounts only for experiments after calibration, latency tests, etc.) Overall, these experiments indicated a vulnerability with probability less than 0.00469, and only in certain locations. Thus, to find one requires some significant investment in time to scan the entire function and test each location frequently enough to be likely to find a genuine vulnerability. However, if the software experiments are used to guide the hardware experiments, it is possible to target exactly the timing 1.344μs, which could then demonstrate a vulnerability with 0.999 probability requiring only 10 passes over 21 timings (total 210 experiments). Thus, this approach could bring the number of hardware experiments required down orders of magnitude and still confidently confirm or refute a fault injection vulnerability. For the backdoor case study the possibility to find the vulnerability using hardware alone is significantly lower since the location is unique and has a low probability of success. Overall the combined probability of both targeting the right timing and inducing a fault in an experiment is 0.0000957. However,

if guided by the software results that all indicated a specific location to test (i.e. 1.248μs) then the probability to detect a fault is 0.999.

Observe that in both the case studies vulnerabilities were already expected and the locations could be guessed or calculated in advance. However, using the combined approach described here does *not* require this prior knowledge since the software simulations can be performed to find the likely locations to confirm or refute with hardware experiments.

This implies that there is no need to know in advance whether a fault injection vulnerability exists. The software can be used to locate any potential fault injection vulnerabilities, and the hardware used to confirm or refute their feasibility of exploitation. This combined approach is more accurate than software simulations alone (since the false positives are refuted), and much cheaper than the hardware alone since much less experiments are required to demonstrate or refute vulnerabilities.

9 Related Work

This section recalls recent works related to the detection of fault injection vulnerabilities. These are divided according to their general approach as being either software or hardware based.

9.1 Software Based Approach

This section recalls recent related works that use software based approaches for detection of fault injection vulnerabilities.

One recent work which uses formal methods to detect vulnerabilities is [27]. Here the authors present a symbolic LLVM-based Software-implemented Fault Injection (SWiFI) evaluation framework for resilience evaluation. In SWiFI the fault injection simulation and the vulnerability detection are done on the intermediate language LLVM-IR, which limits accurate simulation of fault models closely related to low level hardware effects.

The Symbolic Program Level Fault Injection and Error Detection Framework (SymPLFIED) [36] is a program-level framework to identify potential vulnerabilities in software. The vulnerabilities are detected by combining symbolic execution and model checking techniques. The SymPLFIED framework is limited as SymPLFIED only supports the MIPS architecture [40].

Lazart [39] is a tool that can simulate a variety of fault injection attacks and detect vulnerabilities using formal methods. The Lazart process begins with the source code which is compiled to LLVM-IR. The simulated fault is created by modifying the control flow of the LLVM-IR. Symbolic execution is then used to detect differences in the control flow, and thus detect vulnerabilities. One of the main limitations of Lazart is that it is unable to reason about or detect fault injection attacks that operate on binaries rather than the LLVM-IR.

In [44] the authors propose combining the Lazart process with the Embedded Fault Simulator (EFS) [8]. This extends from the capabilities of Lazart alone by adding lower level fault injection analysis that is also embedded in the chip with the program. The simulation of the fault is performed in the hardware, so the semantics of the executed program correspond to the real execution of the program. However, EFS is limited to only considering instruction skip faults (equivalent to NOPs of Section 4.1).

An entirely low level approach is taken by Moro et al. [34] who use model checking to formally prove the correctness of their proposed software countermeasures schemes against fault injection attacks. The focus is on a very specific and limited model of fault injection that causes instruction skips and ignores other kinds of attacks. Further, the model checking is over only limited fragments of the assembly code.

A less formal approach is taken in [1] where experiments are used for testing the TTP/C protocol in the presence of faults. Rather than attempting to find fault injection attacks, they injected faults to test robustness of the protocol. They combined both hardware testing and software simulation testing, comparing the results as validation of their approach.

A fault model inference focused approach is taken by Dureuil et al. [17]. They fix a hardware model and then test various fault injection attacks based upon this hardware model. Fault detection is limited to EEPROM faults on the ARMv7-M architecture. The fault model is then inferred from the parameters of the attack and the embedded program. The faults are simulated upon the assembly code and the results checked with predefined oracles on the embedded program.

9.2 Hardware Based Approach

This section recalls recent related works that use hardware based approaches for detection of fault injection vulnerabilities.

In [46] the authors apply electromagnetic and optical attacks to the RSA algorithm, a well known algorithm used in various cryptographic systems. The authors presented a successful attack on the RSA algorithm implementation over an 8-bit architecture micro-controller. Experiments showed that the faults can affect program flow as well as the SRAM content and the flash memory.

Skorobogatov [49] showed using a laser, one can effect certain memory cells SRAM and cause them to switch. The experiments were conducted on an PIC16F84 micro-controller. The advantage of using a laser is that they can accurately target a single bit to modify.

In [10] the authors present a practical laser fault attack which target creating fault in the Deep Neural Networks (DNN) on a low-cost micro-controller.

Another type of hardware attack was presented in [7] where the authors showed that they can perform successful attacks by alternating the power supply. The experiments were performed on a software implementation of the AES and RSA crypto algorithm running on a ARM9 CPU. The result showed that it was possible to retrieve the full 256-bit key of the AES crypto algorithm, and reproduce with cheaper equipment a known attack against RSA.

In [37,57] the authors present a survey of the different hardware based approach techniques used to inject a fault. The authors also refer to relevant works where the various fault injection techniques are used. For many other recent and older works on hardware fault injection and their approaches we refer the reader to these works.

9.3 Software and Hardware Based Approaches

Although very few works in the literature considered both software and hardware fault injection together, this section briefly discusses these works.

In [15] the authors attempt to detect faults at runtime that have been inserted at runtime using either software or hardware-based techniques. They concluded that their software-based approach could not emulate the hardware-based approach of single bit-level faults, although they considered this possible in future. Compared to the work presented in this paper, the experiments conducted were limited, the objective was to test the system using different approaches and no combination proposition was given.

In [4] the authors consider four different approaches to fault injection (one software and three hardware) and examine what kinds of faults were created, and how well they were detected by runtime systems. The only software-based fault model considered was the Flip model used here, and this did not co-relate with the other hardware-based fault injection methods; EMP, heavy-ion radiation, and pin-forcing. The authors concluded that the four fault injection techniques are complementary. No attempt was done to propose a combined approach using the software and the hardware one.

10 Conclusion

Both software based and hardware based approaches have been used to detect fault injection vulnerabilities. However, the two approaches have not been directly compared before in the manner presented in this paper. This work presents both software based formal methods analysis and hardware based experiments performed on the same case studies. The results of these experiments are compared to explore what can be learned by bridging between the two approaches.

The results here show that software based approaches do find genuine fault injection vulnerabilities. Although software based approaches may suffer from some false positives, they (when done with multiple fault models) *do not have any false negative results.* (Although the benchmarks here are of course limited in scope, this provides insight into broader challenges in this area.) This allows for software based approaches to provide useful information about potential fault injection vulnerabilities. The results here also showed that (contrary to prior work [32]) EMP effects do not have a single fault model. The results here indicated that multiple fault models together best represent the effects of EMP fault injection attack.

In practice, these fault models correspond to an EMP effect either wiping a byte or word (by setting all the bits to zero) or skipping an instruction.

More generally the results show that there is a co-relation between both approaches. This gives support to research that uses software based approaches to simulate or approximate hardware experiments. Further as mentioned above, the co-relation can be used to influence our knowledge about both approaches and refine our understanding of them.

Combining both software based and hardware based approaches is also vastly more effective in isolating and confirming the existence of a fault injection vulnerability. In practice by combining both approaches finding previously unknown vulnerabilities on whole programs becomes feasible. In the future this should allow the much more rapid discovery of genuine fault injection vulnerabilities that do not require prior knowledge or intuition on the part of the researcher.

References

1. Ademaj, A., Grillinger, P., Herout, P., Hlavicka, J.: Fault tolerance evaluation using two software based fault injection methods. In: On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International, pp. 21–25. IEEE (2002)
2. Alur, R., Henzinger, T.A.: Reactive modules. Formal methods in system design **15**(1), 7–48 (1999)
3. Anceau, S., Bleuet, P., Clédière, J., Maingault, L., Rainard, J.l., Tucoulou, R.: Nanofocused X-ray beam to reprogram secure circuits. In: International Conference on Cryptographic Hardware and Embedded Systems, pp. 175–188. Springer (2017)
4. Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., Leber, G.H.: Comparison of physical and software-implemented fault injection techniques. IEEE Transactions on Computers **52**(9), 1115–1133 (2003)
5. Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on, pp. 105–114. IEEE (2011)
6. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. IACR Cryptology ePrint Archive **2004**, 100 (2004). URL `http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html\#Bar-ElCNTW04`
7. Barenghi, A., Bertoni, G.M., Breveglieri, L., Pelosi, G.: A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA. Journal of Systems and Software **86**(7), 1864–1878 (2013)
8. Berthier, M., Bringer, J., Chabanne, H., Le, T.H., Rivière, L., Servant, V.: Idea: embedded fault injection simulator on smartcard. In: International Symposium on Engineering Secure Software and Systems, pp. 222–229. Springer (2014)
9. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in computers **58**, 117–148 (2003)
10. Breier, J., Hou, X., Jap, D., Ma, L., Bhasin, S., Liu, Y.: Practical fault attack on deep neural networks. arXiv preprint arXiv:1806.05859 (2018)
11. Bukasa, S.: Analyse de vulnérabilité des systèmes embarqués face aux attaques physiques. Ph.D. thesis, Rennes 1, Rennes (2019)
12. Carreira, J., Madeira, H., Silva, J.G., et al.: Xception: Software fault injection and monitoring in processor functional units. Dependable Computing and Fault Tolerant Systems **10**, 245–266 (1998)
13. Christofi, M., Chetali, B., Goubin, L.: Formal verification of an implementation of CRT-RSA vigilant's algorithm. In: PROOFS Workshop: Pre-proceedings, p. 28 (2013)
14. Cortex, A.: Cortex-M3 technical reference manual. Rev. r1p1 (2006)
15. Czeck, E.W., Siewiorek, D.P., Segall, Z.Z.: Software-implemented fault insertion: An FTMP example (1987)
16. Dehbaoui, A., Dutertre, J.M., Robisson, B., Orsatelli, P., Maurine, P., Tria, A.: Injection of transient faults using electromagnetic pulses-practical results on a cryptographic system. IACR Cryptology EPrint Archive **2012**, 123 (2012)
17. Dureuil, L., Potet, M.L., de Choudens, P., Dumas, C., Clédière, J.: From code review to fault injection attacks: Filling the gap using fault model inference. In: International Conference on Smart Card Research and Advanced Applications, pp. 107–124. Springer (2015)
18. Ecoffet, R.: In-flight anomalies on electronic devices. In: Radiation Effects on Embedded Systems, pp. 31–68. Springer (2007)
19. Entrena, L., López-Ongil, C., García-Valderas, M., Portela-García, M., Nicolaidis, M.: Hardware fault injection. In: Soft Errors in Modern Electronic Systems, pp. 141–166. Springer (2011)
20. Given-Wilson, T., Heuser, A., Jafri, N., Legay, A.: An automated and scalable formal process for detecting fault injection vulnerabilities in binaries. Concurrency and Computation: Practice and Experience **31**(23) (2019). DOI 10.1002/cpe.4794. URL `https://doi.org/10.1002/cpe.4794`
21. Given-Wilson, T., Jafri, N., Lanet, J., Legay, A.: An automated formal process for detecting fault injection vulnerabilities in binaries and case study on PRESENT. In: 2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017, pp. 293–300. IEEE (2017). DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.250. URL `https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.250`
22. Hsueh, M.C., Tsai, T.K., Iyer, R.K.: Fault injection techniques and tools. Computer **30**(4), 75–82 (1997). DOI 10.1109/2.585157. URL `http://dx.doi.org/10.1109/2.585157`
23. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ACM SIGARCH Computer Architecture News, pp. 361–372. IEEE Press (2014)

24. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Proactive detection of computer worms using model checking. IEEE Transactions on Dependable and Secure Computing **7**(4), 424–438 (2010)
25. Kooli, M., Di Natale, G.: A survey on simulation-based fault injection tools for complex systems. In: Design & Technology of Integrated Systems In Nanoscale Era (DTIS), 2014 9th IEEE International Conference On, pp. 1–6. IEEE (2014)
26. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: International conference on computer aided verification, pp. 585–591. Springer (2011)
27. Le, H.M., Herdt, V., Große, D., Drechsler, R.: Resilience evaluation via symbolic fault injection on intermediate code. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018, pp. 845–850. IEEE (2018)
28. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: International Conference on Runtime Verification, pp. 122–135. Springer (2010)
29. Legay, A., Traonouez, L.M.: Plasma lab statistical model checker: Architecture, usage and extension. In: 43rd International Conference on Current Trends in Theory and Practice of Computer Science (2017)
30. Marinescu, P.D., Candea, G.: LFI: A practical and general library-level fault injector. In: Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on, pp. 379–388. IEEE (2009)
31. May, T.C., Woods, M.H.: A new physical mechanism for soft errors in dynamic memories. In: Reliability Physics Symposium, 1978. 16th Annual, pp. 33–40. IEEE (1978)
32. Moro, N.: Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués. Ph.D. thesis, Université Pierre et Marie Curie-Paris VI (2014)
33. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on, pp. 77–88. IEEE (2013)
34. Moro, N., Heydemann, K., Encrenaz, E., Robisson, B.: Formal verification of a software countermeasure against instruction skip attacks. Journal of Cryptographic Engineering **4**(3), 145–156 (2014)
35. Pan, J., Bhasin, S., Zhang, F., Ren, K.: One fault is all it needs: Breaking higher-order masking with persistent fault analysis. Cryptology ePrint Archive, Report 2019/008 (2019). `https://eprint.iacr.org/2019/008`
36. Pattabiraman, K., Nakka, N., Kalbarczyk, Z., Iyer, R.: SymPLFIED: Symbolic program-level fault injection and error detection framework. In: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), pp. 472–481. IEEE (2008)
37. Piscitelli, R., Bhasin, S., Regazzoni, F.: Fault attacks, injection techniques and tools for simulation. In: Hardware Security and Trust, pp. 27–47. Springer (2017)
38. Portela-Garcia, M., Lopez-Ongil, C., Garcia-Valderas, M., Entrena, L.: A rapid fault injection approach for measuring seu sensitivity in complex processors. In: On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International, pp. 101–106. IEEE (2007)
39. Potet, M.L., Mounier, L., Puys, M., Dureuil, L.: Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp. 213–222. IEEE (2014)
40. Price, C.: MIPS iv instruction set (1995)
41. Qiao, R., Seaborn, M.: A new approach for rowhammer attacks. In: Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on, pp. 161–166. IEEE (2016)
42. Rivière, L., Bringer, J., Le, T.H., Chabanne, H.: 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, pp. 1–8. IEEE (2015)
43. Rivière, L., Najm, Z., Rauzy, P., Danger, J.L., Bringer, J., Sauvage, L.: High precision fault injections on the instruction cache of ARMv7-M architectures. In: Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on, pp. 62–67. IEEE (2015)
44. Rivière, L., Potet, M.L., Le, T.H., Bringer, J., Chabanne, H., Puys, M.: Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks. In: International Symposium on Foundations and Practice of Security, pp. 92–111. Springer (2014)
45. Roscian, C., Dutertre, J.M., Tria, A.: Frontside laser fault injection on cryptosystems-application to the AES' last round. In: Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on, pp. 119–124. IEEE (2013)
46. Schmidt, J.M., Hutter, M.: Optical and EM fault-attacks on CRT-based RSA: Concrete results. na (2007)
47. Seaborn, M., Dullien, T.: Exploiting the DRAM rowhammer bug to gain kernel privileges. Black Hat (2015)
48. Sebanjila, K.B., Lashermes, R., Lanet, J.L., Legay, A.: Let's shock our IoT's heart: ARMv7-M under (fault) attacks (2018)
49. Skorobogatov, S.: Optically enhanced position-locked power analysis. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 61–75. Springer (2006)
50. Skorobogatov, S.: Optical fault masking attacks. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on, pp. 23–29. IEEE (2010)
51. Standard, N.F.: Announcing the advanced encryption standard (AES). Federal Information Processing Standards Publication **197**, 1–51 (2001)
52. Thomas, A., Pattabiraman, K.: LLFI: An intermediate code level fault injector for soft computing applications. In: Workshop on Silicon Errors in Logic System Effects (SELSE) (2013)
53. Tunstall, M., Mukhopadhyay, D., Ali, S.: Differential fault analysis of the advanced encryption standard using a single fault. WISTP **6633**, 224–233 (2011)
54. Verbauwhede, I., Karaklajic, D., Schmidt, J.M.: The fault attack jungle-a classification model to guide you. In: Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on, pp. 3–8. IEEE (2011)
55. Wang, G., Wang, S.: Differential fault analysis on PRESENT key schedule. In: Computational Intelligence and Security (CIS), 2010 International Conference on, pp. 362–366. IEEE (2010)

56. Yim, K.S.: The rowhammer attack injection methodology. In: Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on, pp. 1–10. IEEE (2016)
57. Yuce, B., Schaumont, P., Witteman, M.: Fault attacks on secure embedded software: Threats, design, and evaluation. Journal of Hardware and Systems Security pp. 1–20 (2018)
58. Ziade, H., Ayoubi, R.A., Velazco, R., et al.: A survey on fault injection techniques. Int. Arab J. Inf. Technol. **1**(2), 171–186 (2004)