Bridging Software-Based and Hardware-Based Fault Injection Vulnerability Detection

Thomas Given-Wilson, Nisrine Jafri, Axel Legay

Bridging Software-Based and Hardware-Based Fault Injection Vulnerability Detection

Thomas Given-Wilson
Inria
Email: thomas.given-wilson@inria.fr

Nisrine Jafri
Inria
Email: nisrine.jafri@inria.fr

Axel Legay
UC Louvain
Email: axel.legay@uclouvain.be

***Abstract*—Software-based and hardware-based approaches have both been used to detect fault injection vulnerabilities. Software-based approaches can provide broad and rapid coverage, but may not correlate with genuine hardware vulnerabilities. Hardware-based approaches are indisputable in their results, but rely upon expensive expert knowledge and manual testing. 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 EMP attacks may not be accurate; and that there is a coincidence 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.**

I. Introduction

There are two main approaches to the detection of fault injection vulnerabilities: software-based and hardware-based. Software-based approaches simulate fault injection on some aspect of the program and detect whether some property of the program has been altered to yield a vulnerability [1], [2], [3], [4]. Hardware-based approaches use direct experimentation on the hardware and program being executed, with vulnerabilities being detected by observation of the outcome [5], [6], [7], [8]. Both approaches have advantages and disadvantages.

The advantages of software-based approaches are in cost, automation, and breadth. Software-based simulations do not require expensive or dedicated hardware and can be run on most computing devices easily [9]. Also with various software tools being developed, and matured, limited expertise is needed to plug together a toolchain to do fault injection vulnerability detection [1], [10]. Such a toolchain can then be automated to detect fault injection vulnerabilities without direct oversight or intervention. Further, simulations can cover a wide variety of fault models that represent different kinds of attacks and can therefore test a broad range of attacks with a single system. Combining all of the above has been demonstrated to support an automated process that can test a program for fault injection vulnerabilities against a wide variety of attack models, and with broad coverage of potential attacks.

The disadvantages of software-based approaches are largely in their implementations or in the confidence in the feasibility of their results. Many software-based approaches have shown positive results, but are often limited by the tools and implementation details, with limitations in architecture, scope, etc. However, the biggest weakness is the lack of confidence in the feasibility of their results: software-based approaches have not been proven to map to genuine vulnerabilities in practice.

The advantages of hardware-based approaches are in the quality of the results. A fault injection demonstrated in practice with hardware cannot be denied to be genuine. The disadvantages of hardware-based approaches are the cost, automation, and breadth. To do hardware-based fault injection vulnerability detection requires specialised hardware and expertise to conduct the experiments. This is compounded when multiple kinds of attacks are to be considered; since different equipment is needed to perform different kinds of fault injection (e.g. EMP, laser, power interrupt). Further, hardware-based approaches tend to be difficult to automate, since the experiments must be done with care and oversight, and also the result can damage or disrupt the hardware in a manner that breaks the automation. Lastly, hardware-based approaches tend to have limited breadth of application; this is due to requiring many different pieces of hardware to test different architectures, attacks, etc. and also due to the time and cost to test large numbers of configurations for fault injection vulnerability.

The correspondence between software-based and hardware-based fault injection vulnerability detection has not been widely explored. 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 (i.e. simulation) and hardware-based (i.e. EMP on hardware) fault injection vulnerability

detection on two case studies. The results of the two approaches are compared to explore how closely the two kinds of approaches coincide. 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 to any single fault model of 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. Although this is not surprising (specially since EMP is inexact at best), this indicates that simulations that consider only a single fault model may not correspond well to EMP.

The two approaches coincide: 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:

- Software-based approaches detect genuine fault injection vulnerabilities.
- Software-based approaches yield false-positive results.
- Software-based approaches did *not* yield false-negative results.
- Hardware-based EMP approaches do *not* have a simple fault model.
- Both approaches coincide.
- Combining software-based and hardware-based approaches yields a vastly more efficient method to detect genuine fault injection vulnerabilities.

The structure of the paper is as follows. Section II recalls background information useful for understanding this paper. Section III presents the main case study used for the experiments. Section IV describes the experimental methodology. Section V overview the key results. Section VI presents the results of experiments on a second case study. Section VII discusses the findings and broader context. Section VIII presents a combined approach that used both software and hardware to efficienctly find vulnerabilities. Section IX concludes.

II. Background

This section recalls information useful to understanding the rest of the paper. Section II-A overviews fault injection, fault injection vulnerabilities, and approaches to detecting vulnerabilities. Section II-B recalls key points on formal verification techniques used in this and related works for software detection of fault injection vulnerabilities. Section II-C overviews the software process used to find fault injection vulnerabilities here. Section II-D overviews the hardware process used to find fault injection vulnerabilities here.

A. Fault Injection

Fault injection is any modification at the hardware level which may change normal program execution. Fault injection can be unintentional (e.g. background radiation, power interruption [11], [12]) or intentional (e.g. induced EMP [13], [14], rowhammer [15], [16], [17]).

Unintentional fault injection is generally attributed to the environment [11], [18]. An example of this is one of the first observed fault injections where radioactive elements present in packing materials caused bit flips in chips [12].

Intentional fault injection occurs when the injection is done by an *attacker* with the intention of changing program execution [14], [15], [16], [17]. For example fault injection attacks performed on cryptographic algorithms (e.g. RSA [19], AES [20], PRESENT [21]) 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. The focus of this paper is on such vulnerabilities that *will* change the program execution in a manner useful to an attacker.

There are two broad approaches to detecting (potential) fault injection vulnerabilities: software and hardware.

Software based fault injection (SBFI) vulnerability detection generally uses simulation of the software (and hardware) along with some simulated fault injection to identify behaviours of interest (e.g. vulnerabilities) [9]. Key to the software simulation approach is the definition of a *fault model* that describes the change made to the simulation [22], [23]. Fault models are used to specify the nature and scope of the induced modification. A fault model has two important parameters, location and impact. The location includes the spatial and temporal location of fault injection relating to the execution of the target program. The impact depends on the type and precision of the technique used to inject the fault, the granularity of the impact can be at the level of bit, byte, or multiple bytes.

Hardware based fault injection (HBFI) vulnerability detection generally begins by identifying a potential point-of-interest (PoI) of the hardware that can cause some effect on the program execution when faulted by some mechanism. The majority of the effort in this approach is in finding the PoI and then attempting to determine what the hardware effect for the application of the mechanism to the PoI is [24], [25].

Once the hardware effect has been determined, the typical approach is to exploit this on some program that has a known weakness to demonstrate that the weakness can be turned into a vulnerability [14]. For example, in [20] the authors demonstrate using a laser to change a single bit, which can be used to create a crypto-analytical vulnerability in the implementation of AES.

B. Formal Verification

This section recalls formal verification techniques related to the SBFI process used in this paper.

1) Model Checking: Model checking (MC) [26] is a formal verification technique used to verify if a given model satisfies specified properties. MC has the advantage that all possible states of the model are considered, and so is guaranteed to be able to answer whether or not a given property holds for a given model. MC has demonstrated its efficiency in verifying systems [27], [28], although MC still has some limitations. Due to exploring every possible state of the model, large or complex programs can have extremely large models that MC may fail to check in reasonable time [29].

2) Statistical Model Checking: Due to the limitations of using MC on large and complex programs, *Statistical Model Checking* (SMC) is an alternative approach that can rapidly find approximate results [30]. SMC performs several runs where a given property is checked, and then uses the results to have statistics which represent the probability of violation and/or verification of the given property holding for the given model.

Since unknown modifications to the execution of programs may lead to highly complex models, the choice in this work is to use SMC.

3) Properties: 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 Sections II-C & IV-A). 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.

C. Software Process

This section overviews the the process used for software-based fault injection vulnerability detection. This process is adapted from that of [1] 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 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 in [1] in two ways. Firstly, here the properties are specified independently of the binary whereas in [1] the properties are embedded in the binary. Secondly, here SMC is used in place of MC (this makes the results here incomplete in the sense that only statistical results are obtained, but this is sufficient to demonstrate existence of results and computationally much more efficient).

D. Hardware Process

This section overviews the hardware process used for detecting fault injection vulnerabilities. This process is common to many prior works [24], [25], [31], [32].

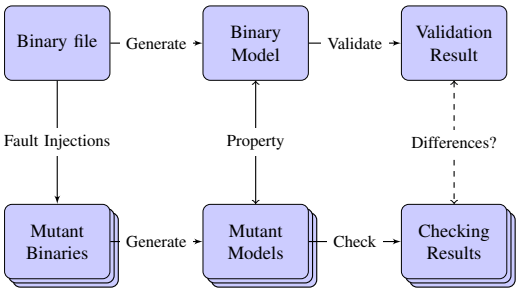

Fig. 1: Software Process Diagram

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.

III. Case Study: Control Flow Hijacking

This section presents the main case study used in this work. This control flow hijacking case study (CFH) [31] 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 ARM-v7. 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 IV-B. 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: Control Flow Hijacking Case Study C Code

Here the attacker wishes to hijack the function contorl flow to return `0x55555555` even when `pin_correct` has the value `1`. Since in the code being experiment 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 ARM-v7 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, #4294967295
 8000abc: f04f 3455 movne.w r4, #1431655765
 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: CFH Case Study 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 `#4294967295`(=`0xFFFFFFFF`) into the register `r4` if the prior condition is satisfied. Similarly the instruction at `8000abc` loads the value `#1431655765`(=`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. (Although this does not mean that faults in other instructions cannot also changes to program execution.)

IV. 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 and perform extensive software simulations 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 SBFI and HBFI techniques coincide.

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

A. Software

The software-based experiments were performed by an implementation of the process described in Section II-C. The implementation of the process presented in II-C 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 TOOL1[1]. (RML [33] is a state-based language based on the Reactive Modules formalism [34] and used as the input language for Plasma Lab [35].) The specified property is then validated to hold on the generated RML model using the SMC Plasma Lab [35]. The mutant binaries corresponding to simulated fault injections are generated using TOOL2[2]. The RML models for the mutant binaries are generated using the TOOL1 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[3].

For software simulation various fault models can be simulated by TOOL2. Since the ElectroMagnetic Pulse (EMP) used here (see Section IV-B below) does not have a single consistent fault model [14], 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 [36], [20].
- 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 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) [37].
- 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 [38].
- 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 [39].

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 III.

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

[1] TOOL1 is a translation tool that translates from ARM-v7 binaries to RML models.

[2] The TOOL2 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.

[3] Due to using SMC minor differences can occur due to the statistical model checking.

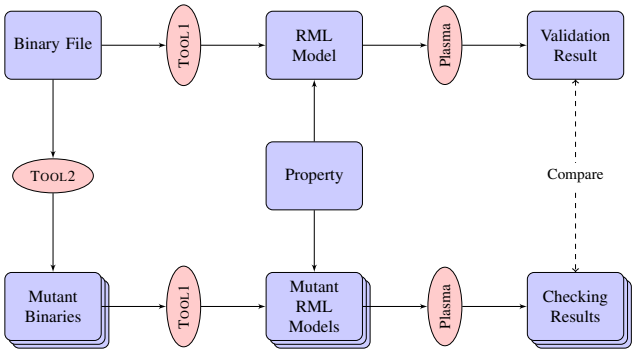

Fig. 2: Software Implementation Diagram

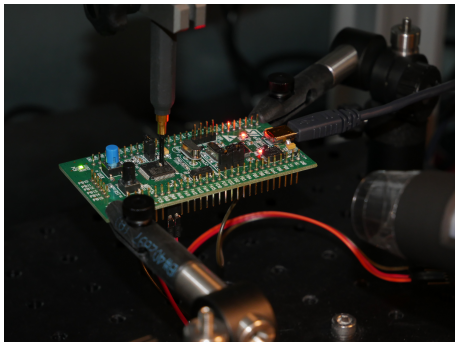

Fig. 3: HBFI Probe Location

3.1 GHz Intel Core i7 processor, 16 GB of RAM, and running macOS High Sierra 10.13.3.

B. Hardware

The hardware process also follows the standard approach to HBFI as presented in Section II-D.

The chosen target hardware is a STM32 Value-line discovery board with ARM Cortex-M3 core micro controller 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.

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[4] count was calculated for the case study functions (using the Cortex-M3 technical reference manual [37]). 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 III for each execution and fault injection are then recorded. This is then repeated a large number of times to gain statistical 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.)

C. Bridging Software And Hardware

This section shows how to bridge the software based and hardware based approaches (Sections IV-A & IV-B above) and then compare the results.

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.

[4]Each clock cycle is approximately 40ns.

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).

V. 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.

A. Software

An overview of results of the software simulation for the control flow hijacking case study can be seen in Figure 4. (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 2, 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 2.

The instruction `cmp r3, 1` at byte address `8000ab4` in Listing 2 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 models modification will effect the comparison of 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 2 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, 4294967295` on byte address `8000ab8` in Listing 2 sets the return register `r4` to the value `4294967295` which corresponds 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 return register to the correct value. So the returned value will be whatever was already in the register `r4` generally yielding an incorrect result.

B. Hardware

This section overviews the results of the hardware experiments. Using the calculations described in Section IV-B 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.

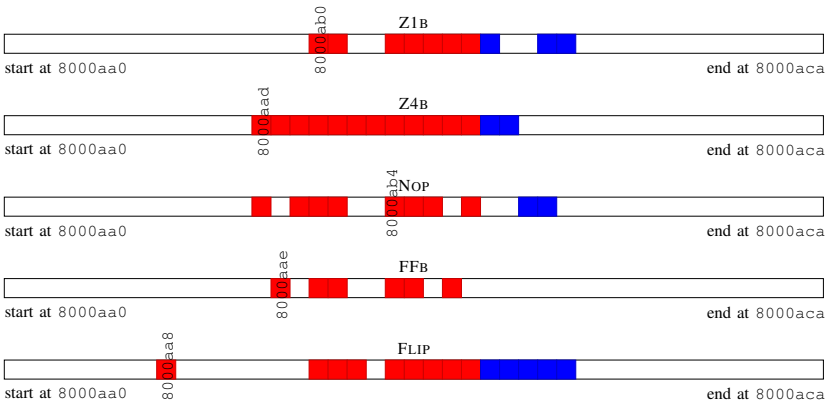

Fig. 4: SBFI Control Flow Hijacking Results

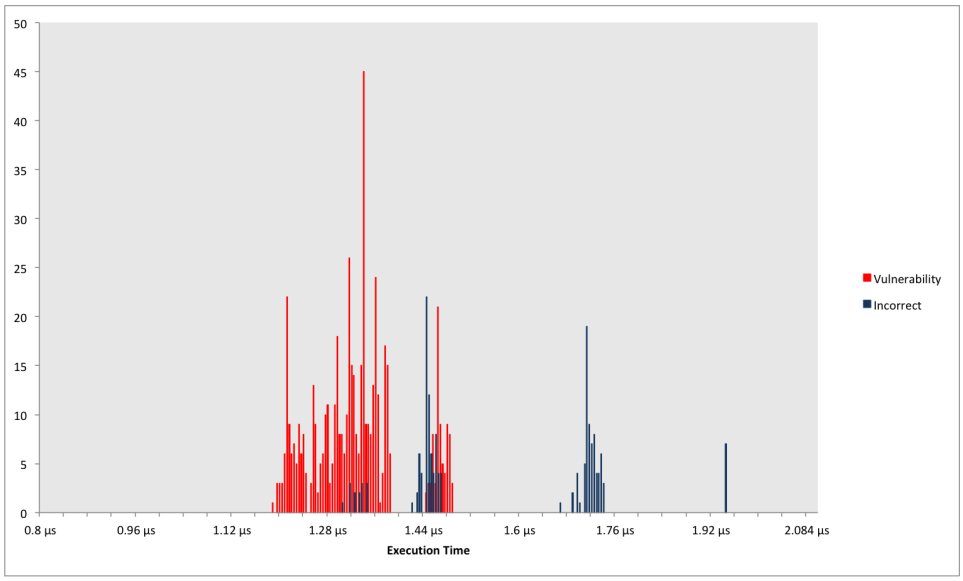

Fig. 5: HBFI Control Flow Hijacking Results

An overview of the results of the hardware experiments for the control flow hijacking case study can be seen in Figure 5. 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 [37]), 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 2 instruction at a time. From all the above information the hardware fault injection vulnerabilities in Fig. 5 can be mapped to the addresses in Listing 2. The vulnerability detected between 1.192μs and 1.232μs corresponds to the instruction at byte address `8000aac` in Listing 2. The vulnerability detected between 1.236μs and 1.312μs corresponds to the instructions at byte address `8000aae` and `8000ab0`. The incorrect result in 1.424μs to 1.472μs corresponds to the instructions at byte address `8000ab2` and `8000ab4`. The vulnerability detected between 1.448μs and 1.492μs corresponds to the instructions at byte address `8000ab4` to `8000ab8`. The incorrect result in 1.672μs to 1.948μs corresponds to the instructions at byte address `8000ac6`

and 8000ac8.

C. Comparison

This section compares the results of the software based and hardware based fault injection experiments presented in the previous two sections (V-A & V-B). Note that for brevity detailed comparison is omitted here, and the more interesting observations as highlighted.

Overall observe that both approaches detected vulnerabilities in the instructions (starting) at byte addresses 8000aac, 8000aae, 8000ab0, 8000ab4, 8000ab6, and 8000ab8 in Listing 2. However, no fault was detected by the hardware prior to 8000aad (implying the FLIP fault injection vulnerabilities here could not be realised).

Overall both approaches detected incorrect results in the instructions (starting) at byte addresses 8000ab2 and 8000ab4 in Listing 2. 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 coincidence, 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. 6. 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.

VI. 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 VI-A introduces the code and weakness. Section VI-B presents the software experimental results for the backdoor case study. Section VI-C highlights the hardware experimental results. Section VI-D overviews the comparison of the software and hardware results. Note that the experimental methodology here is the same as in Section IV.

A. Backdoor attack

This section recalls the Fault Activated Backdoor program from [31]. The core of the weakness in the code is a backdoor function (shown in Listing 3) that is hidden in the program but cannot be reached by any execution path. The normal behaviour of the program includes encryption with AES [40] 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 3. 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 ARM-v7 instruction and so the value is stored as a seperate 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];
  }
```

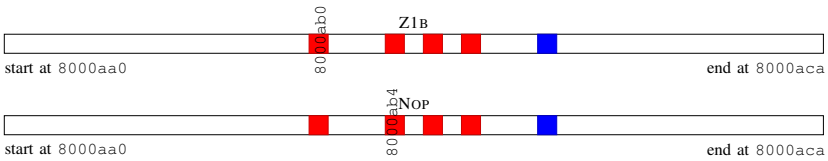

Fig. 6: SBFI Control Flow Hijacking Results Only First Byte Instruction Results

```
  HAL_GPIO_WritePin(LED3_GPIO_PORT, LED3_PIN,
      GPIO_PIN_SET);
}
```

Listing 3: Backdoor Case Study C code

The corresponding assembly code for the `blink_wait` function is shown in Listing 4. 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 V-A) 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 4: 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. For both software and hardware *crashes* were failure to terminate or provide output.

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.

B. Software Experiment Results

An overview of the backdoor case study software experiment results can be seen in Figure 7. 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 4.

The instruction `bd80 pop {r7, pc}` at byte address `80005ca` in Listing 4, 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 4. This instruction is a branching instruction, which will jump to the instruction at `80005e8` in Listing 4. 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.

C. Hardware Experiment Results

An overview of the backdoor case study hardware experiment results can be seen in Figure 8. As before (see Section V-B) various measurements and experiments were performed to ensure the correct timing for the fault injection, and a large number of 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 4.

D. Comparison

This section compares the results of the software based and hardware based fault injection experiments from the previous two sections (VI-B & VI-C). Both approaches detected vulnerabilities in the instruction at byte addresses `80005ca` in Listing 4.

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 in the instruction at byte addresses `80005ca` in Listing 4.

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 4.

The comparison here offers little useful information in improving the understanding of the relation between software based and hardware based approaches. The ruling out of the FLIP fault model as being likely for EMP effects aligns with the results of Section V-C, but little else was learned here that can improve over the information gained from Section V-C. (That said, the agreement on the FLIP fault model and lack of contradiction at least supports the prior conclusions.)

VII. DISCUSSION

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 [14] 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 zero's (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 did not correspond well to the EMP fault injection results. Although the FLIP and FFB 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 coincide 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

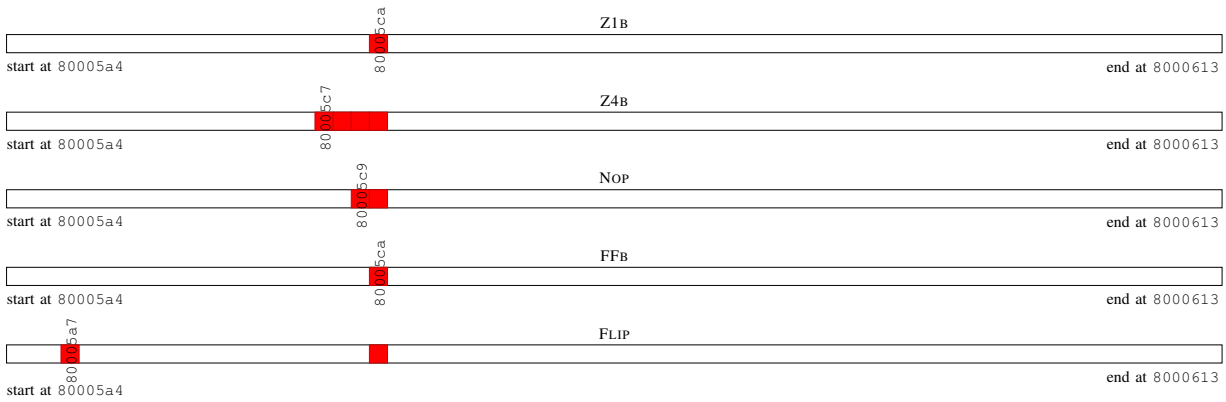

Fig. 7: SBFI Backdoor Results

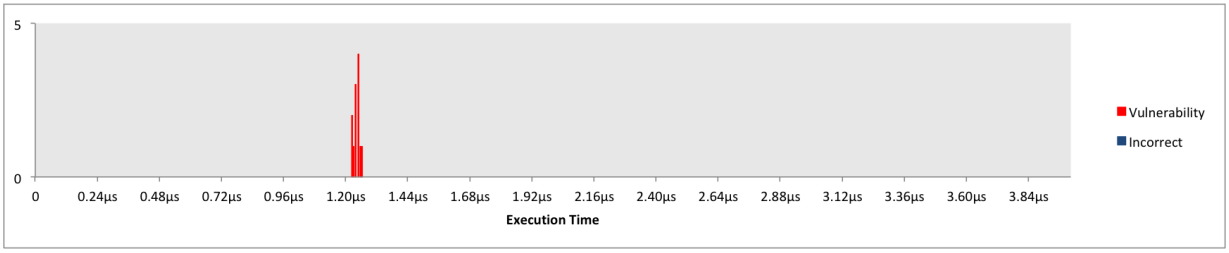

Fig. 8: HBFI Backdoor Results

there is a coincidence 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 do 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) even on relatively large programs, 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.

VIII. 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 not 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 programs entire execution life-cycle, which may be impossible for programs designed to run for years.

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 [1], [10] 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, ~ 117035 hardware experiments were conducted to generate the results shown in Fig. 5. (This number accounts only for experiments after calibration, latency tests, etc.) Overall, these experiments indicated a vulnerability 0.469% of the time, 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 0.00957%. 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 means 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 of 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 many less experiments are required to demonstrate or refute vulnerabilities.

IX. 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. This work presents both broad spectrum software based formal methods analysis and large scale hardware based experiments performed on the same case study. 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 form some false positives, they (when done with multiple fault models) *do not have any false negative results*. This allows for software based approaches to provide useful information about potential fault injection vulnerabilities, and strong guarantees about the absence of fault injection vulnerabilities. The results here also showed that (contrary to prior work [41]) 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 coincidence between both approaches. This gives support to research that uses software based approaches to simulate or approximate hardware experiments. Further as mentioned above, the coincidence 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] T. Given-Wilson, N. Jafri, J. Lanet, and A. Legay, "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*. IEEE, 2017, pp. 293–300. [Online]. Available: https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.250

[2] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 213–222.

[3] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "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)*. IEEE, 2008, pp. 472–481.

[4] 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.

[5] A. Barenghi, G. M. Bertoni, L. Breveglieri, and G. Pelosi, "A fault induction technique based on voltage underfeeding with application to attacks against aes and rsa," *Journal of Systems and Software*, vol. 86, no. 7, pp. 1864–1878, 2013.

[6] J.-J. Quisquater, "Eddy current for magnetic analysis with active sensor," *Proceedings of Esmart, 2002*, pp. 185–194, 2002.

[7] S. Skorobogatov, "Optical fault masking attacks," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE, 2010, pp. 23–29.

[8] J. Balasch, B. Gierlichs, and I. Verbauwhede, "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*. IEEE, 2011, pp. 105–114.

[9] R. Piscitelli, S. Bhasin, and F. Regazzoni, "Fault attacks, injection techniques and tools for simulation," in *Hardware Security and Trust*. Springer, 2017, pp. 27–47.

[10] T. Given-Wilson, A. Heuser, N. Jafri, J.-L. Lanet, and A. Legay, "An automated and scalable formal process for detecting fault injection vulnerabilities in binaries," 2017.

[11] T. C. May and M. H. Woods, "A new physical mechanism for soft errors in dynamic memories," in *Reliability Physics Symposium, 1978. 16th Annual*. IEEE, 1978, pp. 33–40.

[12] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks." *IACR Cryptology ePrint Archive*, vol. 2004, p. 100, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html\#Bar-ElCNTW04

[13] A. Dehbaoui, J.-M. Dutertre, B. Robisson, P. Orsatelli, P. Maurine, and A. Tria, "Injection of transient faults using electromagnetic pulses-practical results on a cryptographic system-." *IACR Cryptology EPrint Archive*, vol. 2012, p. 123, 2012.

[14] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE, 2013, pp. 77–88.

[15] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, 2015.

[16] K. S. Yim, "The rowhammer attack injection methodology," in *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*. IEEE, 2016, pp. 1–10.

[17] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 161–166.

[18] R. Ecoffet, "In-flight anomalies on electronic devices," in *Radiation Effects on Embedded Systems*. Springer, 2007, pp. 31–68.

[19] M. Christofi, B. Chetali, and L. Goubin, "Formal verification of an implementation of CRT-RSA vigilant's algorithm," in *PROOFS Workshop: Pre-proceedings*, 2013, p. 28.

[20] C. Roscian, J.-M. Dutertre, and A. Tria, "Frontside laser fault injection on cryptosystems-application to the aes'last round," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 119–124.

[21] G. Wang and S. Wang, "Differential fault analysis on present key schedule," in *Computational Intelligence and Security (CIS), 2010 International Conference on*. IEEE, 2010, pp. 362–366.

[22] I. Verbauwhede, D. Karaklajic, and J.-M. Schmidt, "The fault attack jungle-a classification model to guide you," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. IEEE, 2011, pp. 3–8.

[23] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: Filling the gap using fault model inference," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2015, pp. 107–124.

[24] L. Entrena, C. López-Ongil, M. García-Valderas, M. Portela-García, and M. Nicolaidis, "Hardware fault injection," in *Soft Errors in Modern Electronic Systems*. Springer, 2011, pp. 141–166.

[25] 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," in *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 62–67.

[26] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 424–438, 2010.

[27] E. A. Emerson, "The beginning of model checking: A personal perspective," in *25 Years of Model Checking*. Springer, 2008, pp. 27–45.

[28] S. Yamane, R. Konoshita, and T. Kato, "Model checking of embedded assembly program based on simulation," *IEICE TRANSACTIONS on Information and Systems*, vol. 100, no. 8, pp. 1819–1826, 2017.

[29] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[30] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *International Conference on Runtime Verification*. Springer, 2010, pp. 122–135.

[31] K. B. Sebanjila, R. Lashermes, J.-L. Lanet, and A. Legay, "Lets shock our iots heart: Armv7-m under (fault) attacks," 2018.

[32] M. Portela-Garcia, C. Lopez-Ongil, M. Garcia-Valderas, and L. Entrena, "A rapid fault injection approach for measuring seu sensitivity in complex processors," in *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*. IEEE, 2007, pp. 101–106.

[33] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *International conference on computer aided verification*. Springer, 2011, pp. 585–591.

[34] R. Alur and T. A. Henzinger, "Reactive modules," *Formal methods in system design*, vol. 15, no. 1, pp. 7–48, 1999.

[35] A. Legay and L.-M. Traonouez, "Plasma lab statistical model checker: Architecture, usage and extension," in *43rd International Conference on Current Trends in Theory and Practice of Computer Science*, 2017.

[36] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault." *WISTP*, vol. 6633, pp. 224–233, 2011.

[37] A. Cortex, "Cortex-m3 technical reference manual," *Rev. r1p1*, 2006.

[38] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014.

[39] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*. IEEE Press, 2014, pp. 361–372.

[40] N.-F. Standard, "Announcing the advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, pp. 1–51, 2001.

[41] N. Moro, "Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués," Ph.D. dissertation, Université Pierre et Marie Curie-Paris VI, 2014.