The State of Fault Injection Vulnerability Detection

Thomas Given-Wilson, Nisrine Jafri, Axel Legay

To cite this version:

Thomas Given-Wilson, Nisrine Jafri, Axel Legay. The State of Fault Injection Vulnerability Detection. Verification and Evaluation of Computer and Communication Systems, pp.3-21, 2018, 978-3-030-00358-6. 10.1007/978-3-030-00359-3_1 . hal-01960915

HAL Id: hal-01960915
https://hal.inria.fr/hal-01960915

Submitted on 19 Dec 2018

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

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

The State of Fault Injection Vulnerability Detection

Thomas Given-Wilson, Nisrine Jafri, and Axel Legay

Inria Rennes - Bretagne Atlantique, France
{thomas.given-wilson, nisrine.jafri, axel.legay}@inria.fr

Abstract. Fault injection is a well known method to test the robustness and security vulnerabilities of software. Fault injections can be explored by simulations (cheap, but not validated) and hardware experiments (true, but very expensive). Recent simulation works have started to apply formal methods to the detection, analysis, and prevention of fault injection attacks to address verifiability. However, these approaches are ad-hoc and extremely limited in architecture, fault model, and breadth of application. Further, there is very limited connection between simulation results and hardware experiments. Recent work has started to consider broad spectrum simulation approaches that can cover many fault models and relatively large programs. Similarly the connection between these broad spectrum simulations and hardware experiments is being validated to bridge the gap between the two approaches. This presentation highlights the latest developments in applying formal methods to fault injection vulnerability detection, and validating software and hardware results with one another.

Keywords: fault injection, vulnerability, model checking, formal methods, simulation

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, EMP, cosmic rays etc. may induce faults. Vulnerability against attacks is usually used to detect places where a malicious attacker may attempt to exploit a system with a targeted fault 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 the requiring specialised hardware to (re)produce a fault [4, 16, 39]. The typical software approach is to perform a simulation based upon a chosen *fault model*; a model of how the fault effects the system. The main advantage of software based approaches are 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 are that they have not been validated against hardware based approaches to verify that their results coincide, 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 [21]. 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 [33], setting up an X-ray beam to target a transistor [2], mounting an EMP probe over a chip to disrupt normal behaviour [23], and many others [3, 5, 37]. The main advantage of such hardware based approaches are that any detected vulnerability is guaranteed to be genuine and potentially reproducible. The main challenge for such hardware based approaches are 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 the software based approach [8, 20, 25], and also many works that explore hardware based approaches [5, 26, 34], but none that explore both. However, 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 [14, 25, 27].

One recent development in the software based approaches is the use of formal methods that can provide stronger claims about the existence (or more often absence) of a fault injection vulnerability [13, 14]. The benefit of formal methods is that the results allow for strong positive statements about the properties that have been proved (or disproved) formally. Thus, results can show that a particular attack cannot succeed [14], or that a particular counter-measure is effective against an attack [24].

However, a significant challenge for the software based approaches is in their breadth of applicability. Many software based approaches are only able to perform simulations of produce results for a single hardware architecture [11, 25]. Similarly, many are only able to produce results for a single fault model, that is they can only detect vulnerabilities against one kind of fault injection attack [27]. Further, some software based approaches only aim to formalise that very small fragments of a program are not vulnerable to fault injection attack (or that a counter-measure is effective), but cannot produce results on even whole functions (let alone whole programs) [24]. Lastly, many approaches do not operate on the binary and hardware model itself, but instead on a (higher-level) language that

is significantly abstracted away from the hardware and actual fault injections [9].

Another significant challenge for the software based approaches are in the accuracy and breadth of the tools they use. Various works have applied techniques that build upon exploiting tools to transform between languages or models [14, 15] or that rely on tools for checking or verifying [27]. However, many of these tools have their own limitations.

Recent works have started to address the above challenges by using an automated process to find fault injection vulnerabilities over whole functions and with many fault models [13, 14]. However, despite these showing good results they are still challenges due to the tools used, and are still targeting a single architecture (albeit upon the binary itself).

Similar issues appear in the hardware based approaches. For example, demonstrating a vulnerability by flipping a bit with a laser on one chip, says nothing about robustness or vulnerability: against flipping a different bit with the same laser; being able to flip the same bit with an X-ray beam; flipping the same bit on a different chip; or against EMP attacks, etc. Here the search for breadth in results is significantly harder to achieve, since testing every possible transistor of a single chip is already infeasible, let alone reasoning over all chips on the market.

Further to the above challenges, to date there has been no significant effort to correlate the software and hardware approaches on a common case study. Thus, there is very little information on whether the two approaches coincide, and whether many of the assumptions made about the approaches hold in practice.

This work provides an overview of some of these recent and significant developments and the general state of the art of fault injection vulnerability detection. This covers a background on the key components of fault injection vulnerability detection, and an overall explanation of how the software based and hardware based approaches operate. The main focus here is on the capabilities and challenges for the state of the art, with a view towards how to develop improved approaches to this area in future.

The latest in automated approaches that can be applied more generally to binary programs are also recalled. This highlights the strengths and capabilities of automating fault injection vulnerability detection, and recent developments in the software based approach. These results are able to show several vulnerabilities in cryptographic implementations. Significantly formal methods were able to be applied to relatively complex program behaviour to produce useful results with reasonable cost.

This paper also discusses the approach of ongoing work on connecting the software and hardware based approaches together by experimenting on the same case study. This overviews the requirements to yield useful results from these experiments, and also identifies some open questions that can be addressed by the results of this ongoing work.

More broadly this paper looks to the future of fault injection vulnerability detection and how to improve the approaches. Longer term the goal should

be to raise fault injection vulnerability detection from a niche and specialised area of software quality, to something that can be applied in the manner that we currently apply bug detection and standards compliance. That is, a future where fault injection vulnerability detection can be automated into development environments and processes to seamlessly integrate with the development environment and identify vulnerabilities efficiently and with negligible cost.

The structure of the paper is as follows. Section 2 recalls background information helpful for understanding this work Section 3 recalls recent related works on various approaches to fault injection vulnerability detection. Section 4 discusses challenges for current approaches and tools. Section 5 overviews some recent results on broad spectrum software approaches to detecting fault injection vulnerabilities. Section 6 considers how to combine software and hardware based approaches and the questions being addressed in ongoing work. Section 7 considers the future directions in fault injection vulnerability detection. Section 8 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 their main advantages and disadvantages.

2.1 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 [4, 21]) or intentional (e.g. induced EMP [10, 23], rowhammer [29, 35, 41]).

Unintentional fault injection is generally attributed to the environment [12, 21] 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 [4].

Intentional fault injection occurs when the injection is done by an *attacker* with the intention of changing program execution [23, 29, 35, 41]. For example, fault injection attacks performed on cryptographic algorithms (e.g. RSA [9], AES [33], PRESENT [40]) 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 effected. 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

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 granularity of the technique used to inject the fault, the granularity can be at the level of bit, byte, or multiple bytes.

According to their granularity fault models can be classified into the following kinds [31]. Bit-wise models: in these fault models the fault injection will manipulate a single bit. One can distinguish five types of bit-wise fault model [31]: bit-set, bit-flip, bit-reset, stuck-at and random-value. Byte-wise models: in these fault models the fault injection will modify eight contiguous bits at a time (usually in the same byte from the program or hardware perspective, *not* spread across multiple bytes). One can distinguish three types of byte-wise fault model: byte-set, byte-reset or random-byte. Wider models: in these fault models the fault injection will manipulate an entire word (defined for the given architecture). For this fault model a sequence of 8 to 64 bits will be modified depending on the architecture, e.g. changing the value of an entire word at once. This will typically target the modification of an entire instruction or single word value.

Based on the fault model classification presented in the paragraph above, a list of fault models used in the experiment results presented in Section 5.2 are as follows. The *bit flip* (FLP) fault model that flips the value of a single bit, either from 0 to 1 or from 1 to 0, this fault model is an example of a Bit-wise model. The *zero one byte* (Z1B) fault model that sets a single byte to zero (regardless of initial value), this fault model is an example of a Byte-wise model. The *unconditional jump* (JMP) and *conditional jump* (JBE) fault models that change the value of a single byte in the target of an unconditional or conditional jump instruction (respectively), these are examples of Byte-wise fault models. The *non-operation* (NOP) fault model that sets a byte to a non-operation code for the chosen architecture, this is an example of a Byte-wise fault model (but can also be implemented as a Wider model by changing the value of the whole instruction word). The *zero one word* (Z1W) fault model that sets a whole word to have the value zero (regardless of prior value), this is an example of the Wider model model.

2.3 Software-Based Fault Injection Approaches

Software-based approaches consists of reproducing at software level the effect that would have been produced by injecting a fault at the hardware level. Software based approaches can be achieved in a number of ways, two common ones

are described below. The first common approach is to simulate the program execution (sometimes including simulating the entire hardware stack as well) and then simulate the fault injection as part of the simulation [17]. The results of the simulation are then used to indicate the behaviour of the program under the fault injection performed. The second common approach is to take the program and use software to build a model of its behaviour [18]. The faults may be injected into the program before or after the model is constructed, but the model is then tested for specific behaviours or properties and the results used to reason about the behaviour of the program. The second is becoming more popular in recent works [13, 14] as formal methods can be used on the model that allow for reasoning about all possible outcomes, and verifying when properties of the model may hold. Note that a vulnerability can be defined rather abstractly in many software based approaches since no clearly observable behaviour is required, merely some definition of how to define vulnerability for the simulation or model.

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 [26]. Also with various software tools being developed and matured, limited expertise is needed to plug together a toolchain to do fault injection vulnerability detection [13, 14]. 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 allows for an easy automated process that can test a program for fault injection vulnerabilities against a wide variety of attack models, and with excellent coverage of potential attacks.

The disadvantages of software-based approaches are largely in their implementations or in the veracity 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 veracity of the results: software-based approaches have not been proven to map to actual vulnerabilities in practice.

2.4 Hardware-Based Fault Injection Approaches

Hardware-based approaches consists of disturbing the hardware at physical level, using hardware materiel (e.g EMP, Laser, Temperature, etc.). Hardware based approaches are usually achieved by configuring the specific hardware to be experimented on and loading the program to be tested for vulnerabilities. A special device is then used to perform fault injection on the hardware during execution, e.g. EMP a chip, laser a transistor, overheat a chip. The result of the execution of the program is observed under this fault injection, with some particular outcomes considered to be “vulnerable” and thus a vulnerability is considered to have been achieved. One typical requirement for this approach is to have idea of how a vulnerability is observable from program execution, since otherwise it

is unclear whether the outcome of execution is a vulnerability or merely some normal or faulty behaviour.

The advantages of hardware-based approaches are in the quality of the results. A fault injection that has been 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 interrupt 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 locations for fault injection vulnerability.

3 Existing 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.

3.1 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 [19]. Here the authors presents a symbolic LLVM-based Software-implemented Fault Injection (SWiFI) evaluation framework for resilience evaluation. InSWiFI 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) [25] 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 [28].

Lazart [27] 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 [32] the authors propose combining the Lazart process with the Embedded Fault Simulator (EFS) [6]. 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 2.2).

An entirely low level approach is taken by Moro et al. [24] 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 fault injection model that causes instruction skips and ignores other kinds of attacks. Further, the model checking is over only limited fragments of the assembly code, and not the program as a whole.

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. [11]. 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.

3.2 Hardware Based Approach

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

In [34] the authors applies the electromagnetic and the 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 [36] 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 [7] the authors presents a practical laser fault attack which target creating fault in the Deep Neural Networks (DNN) on a low-cost micro-controller.

An other type of hardware attack was presented in [5] 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 [26, 42] 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.

4 Challenges

This section discusses common challenges for fault injection vulnerability detection and how they impact the current state of the art in this and closely related areas.

Historically informal approaches (i.e. those that do not employ formal methods) while using a software based approach are unable to provide strong guarantees about the absence of fault injection vulnerabilities [30]. A similar challenge faces hardware based approaches that cannot guarantee that their inability to find a vulnerability ensures that no such vulnerability exists.

One solution to the above challenge is the employment of formal methods in the vulnerability detection approach [14, 25]. This allows results to guarantee that if no vulnerability is found, then no vulnerability exists in the program that was analysed. However, in practice most of these approaches are only able to formally show the lack of vulnerability for a very specific case, or the effectiveness of a counter-measure with limited scope. Thus they are still challenged to produce broad or general results that have the guarantee of formal correctness.

More generally another key challenge for both software and hardware based approaches is the limited scope considered. For either approach the results tend to be highly specific with respect to the architecture being considered [11, 25]. That is, although the results may be complete and correct for one program, they only hold for a single implementation executed on a single specific chip and against a single fault injection technique or fault model [27]. Although this does not limit the significance of finding a fault injection vulnerability, the absence of any vulnerability is not a particularly strong claim under these conditions. Thus the challenge here for both software and hardware based approaches is to find some way to generalise beyond very small and highly specific case studies.

To some extent the software based approaches can be generalised to incorporate multiple fault models and so offer broader coverage and vulnerability detection. However, this requires a software based approach that can be scaled effectively to multiple fault models [13, 14].

For the hardware based approaches the fault model is inherent to the attack and so does not need to be considered. On the other hand, there is limited opportunity to transfer or generalise results. Demonstrating a vulnerability with a laser offers very limited information about whether a vulnerability can be produced with an EMP. Thus a challenge here is to find ways to be able to

transfer or compare both positive and negative results between different kinds of hardware attacks.

Considering this, recent broad spectrum approaches to fault injection vulnerability detection by using automated software approaches show significant promise [13, 14]. However, even these are still limited to some specific architectures and known or implemented fault models.

This identifies yet another challenge area for the software based approaches: the limitations of the tools used in their software process. For many software based approaches there are specific tools developed for them, that tend to lack breadth and maturity [15, 27]. For others that employ tools (often from other domains), these tools tend to have limitations of their own such as being unable to handle everything required (e.g. not supporting all instructions of a given architecture), or being unreliable or inconsistent in their results [13, 14].

5 Broad Spectrum Simulation

This section recalls some recent approaches to addressing the various challenges discussed above. In particular, the focus here is on recent broad spectrum simulations that adopt an automated scalable formal process for detecting fault injection vulnerabilities in binary files [13, 14]. These works address some of the challenges described above and progress towards approaches that vastly reduce their limitations, and thus are more widely applicable.

5.1 Process

This section recalls the core concepts of the process used in [13, 14] for the broad spectrum detection of fault injection vulnerabilities. One of the main contributions of these works is in the development of this automated process that can apply formal verification techniques to the detection of fault injection vulnerabilities in binary files. An overview of the key concepts of the process is depicted in Fig. 1, the rest of this section discusses the key points of implementation and application of this process.

The process begins with a binary file that is to be checked for fault injection vulnerabilities. In [13, 14] the properties that define the correct and vulnerable behaviours are also in this file as annotations maintained by the compiler. (These properties may also define other behaviours such as incorrect or crashed, but these are used for exploration and precision rather than detection of fault injection vulnerabilities.)

The binary file is then translated into a model that represents the behaviour of the binary program in an intermediate language suitable for a formal verification tool. In [13, 14] this translation is done to LLVM-IR as an intermediate language that is then used by a model checker (see below). This translation to LLVM-IR also maintains the properties and converts them to known properties for the model checker.

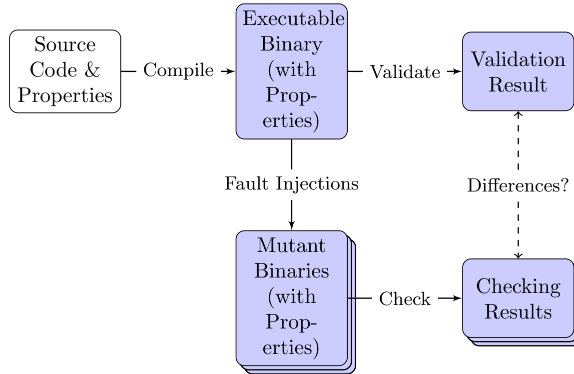

Fig. 1. Software Process Diagram

The properties are then checked by a model checker to validate that they do indeed hold on the original binary program. In [13, 14] this is done using LLBMC a bounded model checker for LLVM-IR. The purpose of this step is to verify that the properties hold and are correctly defined for the binary program, and so that later results can be compared with the validation.

The binary is then injected with a simulated fault injection according to a choice of fault model. This can include many different fault models, and they can be injected in many different locations within the binary. Each possible fault injection combination (of fault model and location) yields a new mutant binary. Note that some care is taken here to ensure fault injection does not effect the property annotations.

The mutant binary is then translated to a model in LLVM-IR, and this model is then checked with LLBMC, both of these are done in the same manner as for the original binary program. The results of this checking on the mutant binary are compared with the validation results for the original program, with any changes being attributed to the fault injection. Thus, the introduction of a “vulnerable” result by the simulated fault injection indicates a fault injection vulnerability.

Note that in [13] this process is refined to be vastly more efficient, but the core concepts are the same in both works.

5.2 Results

This section recalls the main results of these broad spectrum experiments and their implications for detecting fault injection vulnerabilities. The above process was applied to the PRESENT and SPECK cryptographic algorithm implementations. In both cases these algorithms are significantly complex and would be infeasible for a human to check for fault injection vulnerabilities manually. This infeasibility is particularly true for some of the more unusual fault injection mutations that cause instructions to be accessed at a different offset and so interpreted as different instructions.

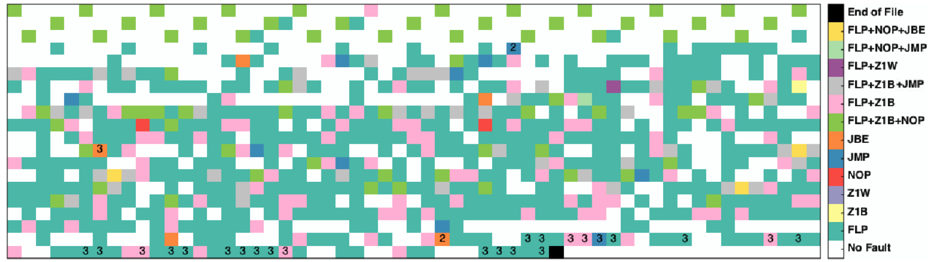

Fig. 2. Model Checking Results for PRESENT (taken from [13])

An overview of the results of applying the process to PRESENT with six fault models described in Section 2.2 (flipping one bit FLP, zeroing one byte Z1B, zeroing one word Z1W, nopping one instruction NOP, modifying a on-conditional jump address JMP, and modifying a conditional jump address JBE) can be seen in Fig. 2. Each square in the diagram indicates a byte of the program that was analysed, with white indicating no impact on execution. Overall 73 vulnerabilities to various properties were found in PRESENT using this automated fault injection vulnerability detection process. Only a small number of these (9 occurrences, indicated with a "2" on the square in the diagram) were found to violate a property that allowed an attacker to send the plaintext in place of the ciphertext. The majority (64 occurrences, indicated with a "3" on the square in the diagram) were crypto-analytical vulnerabilities that allow the encryption key to be calculated from a number of ciphertexts by an attacker. The remaining non-white squares indicated a different "incorrect" behaviour due to fault injection. (Note that a square with a "2" may contain multiple fault injection vulnerabilities depending on the fault model used.)

Overall these broad spectrum experiments indicated that several significant fault injection vulnerabilities existed in the PRESENT algorithm. Further, the results indicated which fault model and location would be able to implement a successful fault injection attack in practice.

The same process with the same six fault models (from Section 2.2) was applied to SPECK yielding the results overviewed in Fig. 3. Overall there were only 9 vulnerabilities found in SPECK, and all of them were fault injection vulnerabilities that allowed the attacker to directly access the plaintext (indicated by a "2" on the square corresponding to the byte in the program in the diagram). Other colours are as described above for PRESENT.

Again these results indicated there exist vulnerabilities in the SPECK implementation, as well as the exact fault model and location to implement the attack.

6 Connection to Hardware

Another major challenge is the lack of connection between the software and hardware experiments in fault injection vulnerability detection. The natural pro-

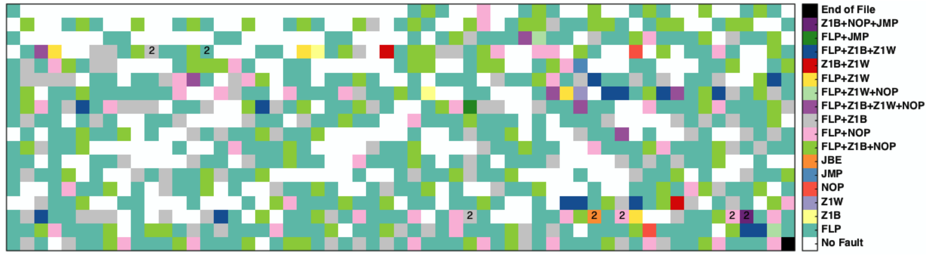

Fig. 3. Model Checking Results for SPECK (taken from [13])

gression in this area would be to combine the software based simulations with hardware based experiments. This section presents some of the key concepts of ongoing work, and the questions being addressed by this work.

The experimental approach used is to take a case study that has a variety of behaviours, and includes weaknesses that are believed to be vulnerable to some kinds of fault injection (i.e. some fault models). This case study is then experimented on using both software and hardware based approaches. The braod spectrum software based approach of Section 5.1 is applied to *the whole program* with a variety of fault models. A hardware based fault injection technique is also used to test *the whole program* for fault injection vulnerabilities.

This combined approach requires significantly more experimental work than is typically conducted. The software based experiments must cover the whole program and a wide variety of (or ideally all known) fault models to be able to reason about and compare the results well. This is in contrast to many prior works where merely checking a specific location, fault model, or counter-measure was the goal. Similarly, the hardware experiments must cover the whole program (and with enough repetitions to be reliable) to ensure that all possible configurations have been tested, and that they can be compared with all the software results. Again this differs from prior work where the hardware approach typically focuses only on very specific (known to be potentially vulnerable) code points and exploits the expert knowledge of the experimenter.

The results of such experiments on a common case study and with coverage of the whole program allows for many interesting questions to be addressed including those below.

– Do the software and hardware based approaches coincide? That is, do the results actually match each other, or are the two unrelated in where they find vulnerabilities and how they explain such vulnerabilities.
– Do the software results have false negatives or positives? In theory using a (correct) formal approach should never yield false negative results since this would imply the formalism is incorrect. However, false positives are a more interesting question since the inability to produce a fault with a hardware technique may be due to a variety of factors. Further, a false positive may merely indicate an extremely rare or difficult to reproduce attack.

– What fault model matches a hardware attack method? For some hardware attacks such as lasering a bit the fault model is very clear. For other hardware methods such as EMP the fault model is considered to correlate with skipping an instruction [22], but with such an imprecise attack the evidence could show otherwise.
– Can combining both software and hardware improve vulnerability detection? Although having access to both approaches (particularly for a variety of hardware attacks) is not feasible, knowing how to make the most efficient use of the available resources could be an advantage gained by knowing the coincidence between software and hardware. That is, software could indicate where to attempt to fault the hardware in a program without a known (or suspected) weakness.

These and other open questions can be addressed by conducting such combined experiments, yielding deeper insight into both the software and hardware aspects of fault injection vulnerability detection.

7 Looking Forward

Recent works have shown significant advances in detecting fault injection vulnerabilities using software, hardware, and combined approaches. However, there are still many opportunities for progress and areas that need significant effort to be able to make fault injection vulnerability detection a reliable and easily applicable part of software development.

One significant challenges for many of the recent and current approaches is the underlying tools that they depend upon. For example, the translation tools used for the results highlighted here rely upon MC-Sema [38] that has various limitations with instruction sets, or failures to correctly translation behaviour. Similarly, in many other works [15, 25] the tools limit the applicability of the technique to some limited scope, limited architecture, limited size, etc. Thus, in many areas the tools used require refinement and maturity, and in other areas the tools simply do not exist and would need to be created. Another example is of the limitations of applying some of the tools in the manner used here, such as LLBMC as used in [14] is not able to produce a counter-example to the property and thus indicate which combination of inputs were vulnerable for a given fault injection vulnerability, further in [13] LLBMC was shown to be inconsistent when producing results. Here an alternative model checker (and likely alternative intermediate language and so translation tools) could yield much more precise results.

Another main area of improvement would be in the automation of fault injection vulnerability detection. Although some recent works highlighted here [13, 14] have begun to address automation, most approaches have not. Being able to automate the the search for vulnerabilities allows fault injection vulnerability detection to be changed from a highly manual process, to another quality or verification process used during software development. Indeed, this would allow fault injection vulnerability to be considered along with other quality checks such as:

bug detection, standards compliance, verification, correctness-by-construction, etc.

Another area to advance in would be in the application of formal methods. Many of the current approaches use highly specific and limited applications of formal methods [14, 25, 27], or a heavy technique that does not exploit domain specific information. For example, the model checking highlighted here does not take into account prior results, or modularity of sub-components. Thus, an incremental approach may yield significant efficiency returns. Similarly, developing and exploiting formal methods that focus on the exact problems considered in vulnerability detection could yield much more precise results than those that are currently state of the art.

Work on strongly connecting the software and hardware based approaches is clearly a goal for future research and development. A strong foundation of understanding of the relations between different kinds of software and hardware based approaches will enrich and improve the results of both. Further, by connecting these results, software based results can be validated to be genuine by reproducing them with hardware experiments. In the other direction, hardware based experiments will demonstrate the efficacy and accuracy of the software based approaches.

Finally, many existing works in the domain of fault injection vulnerabilities and their detection work on examples or programs where a vulnerability is already known to exist. The goal of the work is to (re)produce a known attack (or exploit one that has been intentionally designed in) to demonstrate the efficacy of the approaches used. However, finding vulnerabilities that were not even suspected in advance, or devising approaches that allow the finding of such vulnerabilities in an efficient manner is a clear requirement for practical application in the future.

8 Conclusion

Fault injection represent a serious threat to the robustness and security of many software systems used in daily life. There are two main approaches to detecting fault injection vulnerabilities and testing system robustness; software and hardware based. Both approaches have yielded useful results and can make useful contributions. Software based approaches are good for simulation and being able to cheaply implement, albeit at the cost of the ability to demonstrate a fault injection vulnerability is genuine and can be exploited. Hardware based approaches are good for proving genuine exploitability, but are expensive in time, equipment, and expertise to conduct.

Many recent software based approaches propose the use of formal methods in the process of detecting fault injection vulnerabilities. However, these solutions still have challenges regarding the proposed process as an whole or the tools used in their implementation. Although some of the most recent works attempt to broaden the abilities of software based approaches, in combination with formal methods, there are still challenges ahead for the underlying tools.

Recently proposed and evaluated software approaches have shown their efficiency in detecting potential fault injection vulnerabilities. These software approaches were applied to a variety of systems, working on different architectures, embedding different types of programs. Further, these software approaches have demonstrated scalability in being able to be applied to many fault models in many locations on non-trivial real-world programs with previously unknown vulnerabilities.

Despite these software approach's good results, they can not guarantee that the detected vulnerabilities correspond to real vulnerabilities in practice. The fact that the fault injection are simulated gives no guarantee that in a real physical fault injection attack on the system will have the same effect. Thus the challenge of combining both software and hardware based approaches on a single case study to explore how the two approaches connect. Such experiments should improve our understanding of how to interpret software based approaches: do they produce false positives, false negatives, how reproducible the claimed fault injection vulnerabilities are, and other questions. Similarly, such experiments will allow the fault models of hardware based approaches to be more accurately determined. Further, combining both approaches may yield vastly more effective techniques to find vulnerabilities by exploiting the strengths of each approach.

There are many challenges open in the domain of fault injection vulnerability detection. In addition to those explicitly mentioned above, the broader goal can be to have fault injection vulnerability detection reach the maturity and confidence of other software quality approaches. Developing tools that can integrate into development environments or build processes to automatically detect (potential) fault injection vulnerabilities in the near future is a desirable goal.

References

1. A. Ademaj, P. Grillinger, P. Herout, and J. Hlavicka. Fault tolerance evaluation using two software based fault injection methods. In *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*, pages 21–25. IEEE, 2002.
2. S. Anceau, P. Bleuet, J. Clédière, L. Maingault, J.-l. Rainard, and R. Tucoulou. Nanofocused x-ray beam to reprogram secure circuits. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 175–188. Springer, 2017.
3. 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*, pages 105–114. IEEE, 2011.
4. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *IACR Cryptology ePrint Archive*, 2004:100, 2004.
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*, 86(7):1864–1878, 2013.
6. M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant. Idea: embedded fault injection simulator on smartcard. In *International Symposium on Engineering Secure Software and Systems*, pages 222–229. Springer, 2014.

7. J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu. Practical fault attack on deep neural networks. *arXiv preprint arXiv:1806.05859*, 2018.
8. 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*, 10:245–266, 1998.
9. M. Christofi, B. Chetali, and L. Goubin. Formal verification of an implementation of CRT-RSA vigilant's algorithm. In *PROOFS Workshop: Pre-proceedings*, page 28, 2013.
10. 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*, 2012:123, 2012.
11. 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*, pages 107–124. Springer, 2015.
12. R. Ecoffet. In-flight anomalies on electronic devices. In *Radiation Effects on Embedded Systems*, pages 31–68. Springer, 2007.
13. 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.
14. 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*, pages 293–300. IEEE, 2017.
15. A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner. QEMU-based fault injection for a system-level analysis of software countermeasures against fault attacks. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 530–533. IEEE, 2015.
16. M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, Apr. 1997.
17. A. Johansson. Software implemented fault injection used for software evaluation. *Building Reliable Component-Based Systems*, 2002.
18. M. Kooli and G. Di Natale. 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*, pages 1–6. IEEE, 2014.
19. H. M. Le, V. Herdt, D. Große, and R. Drechsler. Resilience evaluation via symbolic fault injection on intermediate code. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 845–850. IEEE, 2018.
20. P. D. Marinescu and G. Candea. Lfi: A practical and general library-level fault injector. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 379–388. IEEE, 2009.
21. T. C. May and M. H. Woods. A new physical mechanism for soft errors in dynamic memories. In *Reliability Physics Symposium, 1978. 16th Annual*, pages 33–40. IEEE, 1978.
22. N. Moro. *Sécurisation de programmes assembleur face aux attaques visant les processeurs embarqués*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2014.
23. 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*, pages 77–88. IEEE, 2013.

24. N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.
25. 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)*, pages 472–481. IEEE, 2008.
26. R. Piscitelli, S. Bhasin, and F. Regazzoni. Fault attacks, injection techniques and tools for simulation. In *Hardware Security and Trust*, pages 27–47. Springer, 2017.
27. 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*, pages 213–222. IEEE, 2014.
28. C. Price. MIPS iv instruction set, 1995.
29. R. Qiao and M. Seaborn. A new approach for rowhammer attacks. In *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*, pages 161–166. IEEE, 2016.
30. J.-J. Quisquater. Eddy current for magnetic analysis with active sensor. *Proceedings of Esmart, 2002*, pages 185–194, 2002.
31. L. Rivière, J. Bringer, T.-H. Le, and H. Chabanne. A novel simulation approach for fault injection resistance evaluation on smart cards. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–8. IEEE, 2015.
32. L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys. 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*, pages 92–111. Springer, 2014.
33. 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*, pages 119–124. IEEE, 2013.
34. J.-M. Schmidt and M. Hutter. *Optical and em fault-attacks on crt-based rsa: Concrete results.* na, 2007.
35. M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 2015.
36. S. Skorobogatov. Optically enhanced position-locked power analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 61–75. Springer, 2006.
37. S. Skorobogatov. Optical fault masking attacks. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 23–29. IEEE, 2010.
38. Trail of bits. Mc-semantics, 2016. https://github.com/trailofbits/mcsema.
39. 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*, pages 3–8. IEEE, 2011.
40. G. Wang and S. Wang. Differential fault analysis on present key schedule. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 362–366. IEEE, 2010.
41. K. S. Yim. The rowhammer attack injection methodology. In *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*, pages 1–10. IEEE, 2016.

42. B. Yuce, P. Schaumont, and M. Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *Journal of Hardware and Systems Security*, pages 1–20, 2018.