An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT

Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, Axel Legay

An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries

and Case Study on PRESENT

Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay
Inria
Rennes, France
Email: {thomas.given-wilson, nisrine.jafri, jean-louis.lanet, axel.legay}@inria.fr

***Abstract*—Recently fault injection has increasingly been used both to attack software applications, and to test system robustness. Detecting fault injection vulnerabilities has been approached with a variety of different but limited methods. This paper proposes a general process without these limitations that uses model checking to detect fault injection vulnerabilities in binaries. The efficacy of this process is demonstrated by detecting vulnerabilities in the PRESENT binary.**

I. Introduction

Recently fault injection has been increasingly used both as a method to attack software applications, and to test the robustness of software systems. Many systems are particularly vulnerable to fault injection attacks due to operating in hostile environments, i.e. environments where an attacker may be able to perform physical attacks on the system hardware. Many such attacks have been demonstrated on various systems, showing that different kinds of faults can be injected into various devices [1], [2], [3]. Attacks can also be achieved through software alone and do not require attacking the hardware directly. A recent example of this is row hammer [4] that has been exploited in various attacks [5], [6].

The wide variety of fault injection attacks and possible impacts upon a system make it impossible to prevent software from failing under all possible attacks [3]. Thus, recent work has approached the problem of fault injection by limiting the scope of attacks, or limiting the kinds of vulnerabilities analysed [7], [8], [9], [10], often requiring specialised equipment.

This paper proposes an automated formal process for the detection of fault injection vulnerabilities in binaries. In particular, a process that can account for many different kinds of fault injections and that does not require extensive hardware or specialised equipment. This process is achieved by simulating fault injection attacks upon the executable binary for the given software, and then using model checking to determine whether or not the simulated fault injection attack violates properties the software should maintain.

This paper presents an automated process for detecting vulnerabilities in binaries using model checking. The process begins with the *executable binary* that represents the program to be considered. The validation of the binary involves checking various *properties* using model checking to ensure the binary meets its specification. Fault injection attacks are then simulated on the executable binary, producing *mutant binaries*. The properties are then model checked on the mutant binaries. A difference in the result between validating and checking the properties indicates a vulnerability to the fault injection attack that was simulated.

This process provides a general approach that can support detecting a wide variety of fault injection vulnerabilities in binaries by varying the fault model of the fault injection. The strengths of this approach include the following. By operating directly upon the binary, fault injection vulnerabilities that cannot be detected in source languages or intermediate representations can be detected [11], [12]. Formal methods, here model checking, ensure the rigour of the analysis and so ensure that fault injection vulnerabilities that are detected are real and not false positives. An automated process can be easily iterated over various fault injection models and approaches, and thus allows broad, or even complete, coverage of possible fault injection attacks. Combining automation, broad coverage, and formal methods, allows the process to make strong guarantees about the vulnerability of a system that has been analysed.

To demonstrate the efficacy of this process, this paper includes a case study of applying the process to the PRESENT encryption algorithm [13], [14] with various

fault models[1]. PRESENT is a lightweight encryption algorithm designed to be used on embedded devices, thus making it an ideal choice to consider security critical software in hostile environments. The process proposed here was applied to the PRESENT binary with five different fault models for a total of approximately 5700 experiments. These fault injections yielded a number of infinite loops, and crashes, but also 9 vulnerabilities where the encryption algorithm is completely bypassed.

The key contributions of this paper are as follows.

- Describing a general process that allows automated detection of fault injection vulnerabilities in binaries.
- An implementation of the process that allows easy automation with existing tools.
- A case study demonstrating the efficacy of using the process on the PRESENT algorithm.
- The identification of 9 fault injection vulnerabilities in the PRESENT algorithm that bypass encryption.

The structure of the paper is as follows. Section II recalls background on fault injection attacks, model checking, properties, and PRESENT. Section III introduces the process proposed in this paper. Section IV overviews the implementation and tools used to achieve the process. Section V applies the process to the PRESENT algorithm with five fault models and analyses the results. Section VI discusses related work. Section VII concludes and discusses future work.

II. Background

This section recalls key concepts and information useful to understanding the rest of the paper. These are divided into four main areas: fault injection, model checking, properties, and the PRESENT algorithm.

A. Fault Injection

Fault injection can be considered as an attack, when an attacker targets the hardware of a system to create an exploitable error at the software level. The goal of such an attack is to cause a specific effect at the hardware level, that in turn creates an exploitable change in the software behaviour. The rest of this section recalls key points regarding the kinds of fault injection attacks considered here.

The hardware effect of a fault injection attack is described through a *fault model* that specifies the nature and scope of the induced modification. Typically such attacks are achieved by changing a value stored in the hardware, such as changing the value of a whole byte [3]. Such hardware effects generally focus on the kind of fault that can be created rather than the effect this has on the software.

Simulating fault injection attacks in an experimental environment can be done in two ways: reproducing the attack on the hardware, or simulating the attack with software. Reproducing the attack using hardware technology is relatively difficult and expensive, since specialised hardware must be used to inject the fault (e.g. using a laser [7], or electromagnetic pulse [8]). Comparatively, simulating with software is easy and cheap because this requires only modification to the executable binary and no specialised hardware. Since the goal in this paper is to develop an efficient process that can be implemented with a software tool chain, the rest of this paper will only consider software simulations.

Software simulation attacks can also be classified into two kinds of fault injection attacks, *run time* and *compile time*. Run time fault injection attacks are those that occur only while the code being attacked is being executed. Compile time fault injection attacks are those that occur at any time starting form compilation of the code, and up until just prior to execution. This paper considers only compile time fault injection attacks since this captures many run time faults as well and also builds towards future work (see Section VII-A).

B. Model checking

Model checking is a formal method for determining whether properties hold on a model [15]. Model checking 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 property holds for a given model.

The *model* is a representation of the program or system being considered. A good model is able to represent all the possible states and transitions that the program can achieve. In this work, the model represents an executable binary program.

The cost of model checking comes in the potential exponential complexity used to consider all the possibly infinite states of a model. However, for limited models, model checking is highly efficient and precise. Further, various approaches have been used to make model checking efficient even for large and complex models (or programs) [16].

Bounded model checking is a refinement of model checking that alleviates some of the issues with possibly infinite complexity by bounding the checking [17]. The key idea in bounded model checking is to put a bound

[1]Other experimental results to illustrate properties and binary only attacks are available in the long version of this paper [12].

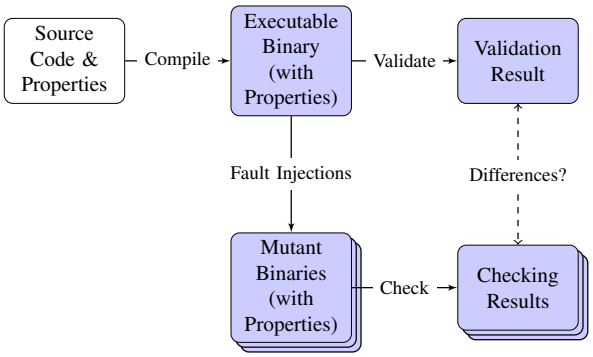

Fig. 1: Process Diagram

on parts of the model that could be infinite (or at least extremely large). For example, checking a program with a loop, going through hundreds or millions of iterations could be very costly for model checking. However, bounded model checking of such an example could limit the number of times to iterate through a loop. Thus, bounded model checking allows limits to be placed upon such potentially unbounded aspects of model checking.

C. Properties

To perform model checking requires specifying the *properties* to be checked upon the model. There are two main kinds of properties that can be checked *safety*, and *liveness* [15]. Safety properties are used to express that certain propositions hold when they are encountered. Liveness properties express that propositions hold over some temporal dimension. This paper only considers safety properties since these are clearer, more intuitive to represent, and sufficient to illustrate the feasibility of the process. Liveness properties can also be checked in a similar manner, although this is not presented in this paper, for further details see the discussion in Section VII-A.

Safety properties can be expressed by simple propositions that can be annotated into the code of the program being considered. Generally such properties support: negation, equality, inequality, conjunction, and disjunction. These can be defined upon values or states within the model (and thus the binary being checked).

D. The PRESENT Algorithm

PRESENT [13], [14] is a lightweight block cipher designed for use on low power and CPU constrained devices. The PRESENT algorithm consists of 31 rounds of a Substitution-Permutation Network (SPN) with block size of 64 bits. The canonical implementation[2] supports key lengths of 80 or 128 bits. The core encryption algorithm is the same for both 80 and 128 bit keys.

The version of PRESENT analysed here is the canonical version in C for 32 bit architectures (size optimised, 80 bit key) with minor modifications to change loop types (due to limitations in the tools used, see Section VII-A for further discussion of these).

[2]Available at http://www.lightweightcrypto.org/implementations.php

III. Process

This section details the process presented in this paper for detecting fault injection vulnerability in binaries.

An overview of the process is as follows (and shown in Figure 1). Prior to starting the process, the source code, and the properties represented by annotations within the source code, must be defined. The preparation step for the process is then to compile the source code and properties to produce an executable binary in a manner that preserves the properties as annotations. The properties are validated to hold on the executable binary using model checking. The executable binary is then injected with simulated faults to produce mutant binaries. The properties are then checked upon the mutant binaries again using model checking. A difference in the results of validation and checking the properties indicates a vulnerability to the simulated fault injection. The rest of this section details this process.

The choice to start the preparation with the source code and not the binary is made for illustrative clarity and ease of use for the software developer, since defining properties over binaries is more arduous. However, most aspects of the process do not rely upon this choice, and future work is to be able to start directly from the binary (further discussion in Section VII-A)

Since this paper considers fault injection attacks upon the binary it is necessary to compile the source code (and here properties) into an executable binary. This executable binary represents the software application that would be executed by the system in practice. Thus to simulate fault injection attacks on the actual system, the executable binary must be used in the simulation. For the process here the compilation must maintain the properties, and so compilation must maintain the properties as annotations in the executable binary.

The properties are validated to hold upon the executable binary using model checking. This ensures that the executable binary meets the specification of the properties. If there is some other error in the source code or compilation, this can be detected here and not be (incorrectly) attributed to fault injection vulnerability.

Next is the simulation of the fault injection on the executable binary to produce mutant binaries. This simulates

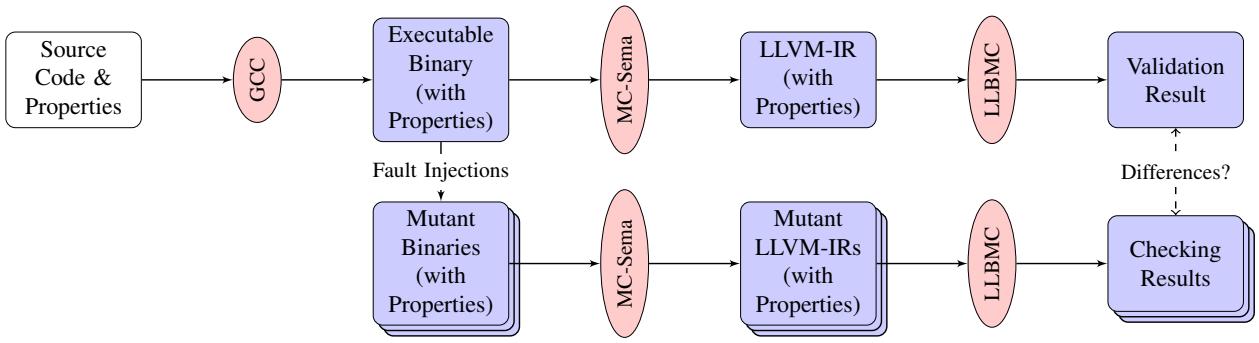

Fig. 2: Implementation Diagram

the actual fault injection attacks and produces mutant executable binaries that represent the executable binary after the attacks have been effected.

Lastly, the validation results from model checking the executable binary are compared with the checking results from model checking each mutant binary. Differences in property checking outputs indicate that the fault that was injected yields a change in behaviour that violates the properties, and so could be exploited by an attacker.

IV. Implementation

This section presents the implementation of the process from the previous section[3]. The implementation here exploits currently available tools where possible, despite some having significant limitations. This choice was made in order to focus upon a simple and feasible implementation of the process. For discussion of the limitations of the tools used in the implementation here, please refer to Section VII-A.

An overview of the implementation is as follows, and shown in Figure 2. The implementation begins with the source code written in the C language and the properties represented in the source code by assert statements. The source code and properties are compiled to an executable binary by the GNU C Compiler (GCC) [18]. The executable binary (including the properties contained within) is transformed into an *intermediate representation* in Low Level Virtual Machine Intermediate Language (LLVM-IR) by the Machine Code Semantics (MC-Sema) tool [19]. The properties are then checked on the intermediate representation using the Low Level Bounded Model Checker (LLBMC) [20], [21]. An automated fault injection tool then produces mutant binaries by injecting faults into the executable binary according to the tool's fault model. The steps to model check the properties on the executable binary are then repeated for each mutant binary. Finally, the results of model checking the executable binary and each mutant binary are compared for differences by matching the outputs of LLBMC.

[3]A detailed description of the implementation designed to allow reproduction is in the long version of this paper [12].

V. Case Study: PRESENT

This section presents a case study of five different fault injection attacks against the PRESENT algorithm.

A. Experimental Design

All the experiments tested a single property to capture the capability of a fault injection attack to bypass the encryption algorithm. The property checked whether the "ciphertext" at the end of the encryption was different to the "plaintext". Thus, violations of this property indicated the encryption algorithm had been effectively bypassed. The result of each fault injected mutant binary were thus classified into one of: *passed* where model checking of all properties succeeded; *infinite loop* when the fault caused an infinite loop in model checking; *crashed* when the fault caused the program to crash; and *vulnerable* when the fault caused the property to be violated.

The five fault models are: *modifying an unconditional jump* (JMP) to jump to a new address; *modifying a conditional jump* (JBE) to jump to a new address; *zero 1 byte* (Z1B) that sets a single byte to zero; *zero 2 bytes* (Z2B) that sets two consecutive bytes to zero; and *NOP'ing an instruction* (NOP) that sets a byte to a non-operation code. Each is detailed below when considering the results for that fault model.

B. Results Overview

An overview of the results for injecting these fault models in all possible locations in the PRESENT binary

can be seen in Table I. All the fault models tested caused crashes, with these being most common with the Z2B and NOP fault models. Infinite loops were also quite common, either through modification of jumps, damaging iterator code, or damaging conditionals. Vulnerabilities were quite rare, which was as expected, with all arising from the jump fault models. The rest of this section considers each of the fault models and the associated experimental results in detail.

Result	Fault Model					Colour
	JMP	JBE	Z1B	Z2B	NOP	
Passed	1632	502	905	855	784	
Infinite Loop	106	0	53	49	93	
Crashed	62	60	172	225	248	
Vulnerable	1	8	0	0	0	

TABLE I: Overview of Fault Injection Results.

C. Unconditional Jump

The fault model for this experiment was to identify unconditional jump instructions and change their target. For simplicity only increasing the value of the target address was considered (i.e. jumping relatively forward, not relatively backwards). Column **JMP** of Table I presents aggregate results. There are 10 unconditional jumps in the PRESENT binary at addresses 0x0120, 0x014B, 0x0155, 0x018C, 0x01D3, 0x0207, 0x02D4, 0x0313, 0x0361, and 0x0447. The only jump that yielded a vulnerability was at 0x014B, and the details are shown in Figure 3 showing the offsets that could be jumped to and the result of checking each mutant (and the blue box indicating the end of the experiment range).

Most of the significant changes here were infinite loops, with a significant number of crashes, and a single vulnerability that skipped the entire encryption algorithm. The infinite loops are largely as expected, since the modified jump can easily skip loop iterator increment code. The crashes are also to be expected, mostly related to jumping to incorrect byte offsets for the instructions, and so yielding invalid instructions (or instructions that crash in other ways such as trying to read invalid memory segments). The single vulnerability was when the jump for the first loop of the encryption algorithm skips over the entire encryption, going straight to the end of the code. Only a single instance was found as most jumps were "short" (single byte offset), meaning they could not bypass significant amounts of code.

D. Conditional Jump

The conditional jump fault model changes targets similar to the unconditional jump fault model. Column

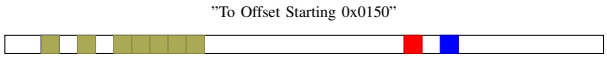

Fig. 3: Unconditional Jumps from Jump at 0x014B

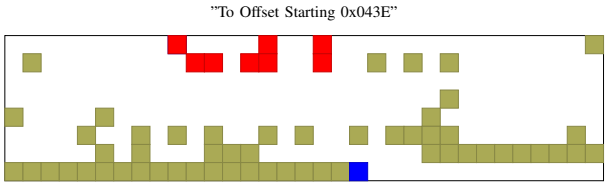

Fig. 4: Conditional Jumps from Jump at 0x043C

JBE of Table I presents the summaries for the two conditional jumps at addresses 0x02C9 and 0x043C. Again vulnerabilities were only found in one at 0x043C and these are detailed in Figure 4.

Here no infinite loops were detected likely due to the conditions always being triggered at least once, instead only crashes where the unconditional jumps were instead targeting bad locations in the code leading to incorrect "instructions". More interesting are the vulnerabilities that fall into two groups. The first group (the first three in the map) jumped to later assignment instructions (including incorrectly offset locations) that ended up bypassing the correct loop controls (by changing values used for later loop control flow), and eventually skipping the encryption algorithm. The second group (the remaining five) simply jumped to the end of the encryption code, merely bypassing the encryption algorithm.

E. Zero 1 Byte

Another fault model to test automating the process over a larger number of mutants was to set a single byte to zero. There are 1130 bytes in the PRESENT executable binary, and each was set to zero in a different mutant, yielding the results shown in the **Z1B** column of Table I. Detailed results showing which faulted bytes yield which effect can be seen in the map in Figure 5.

No fault injection vulnerabilities to this fault model were detected, although many crashes and infinite loops were introduced. This is not a surprising result, since the PRESENT source code has two top-level loops that both perform some part of the encryption. Thus, although setting one byte to zero could skip either one of these, it would require two (non-consecutive) zero one byte fault injection attacks to be "vulnerable" here.

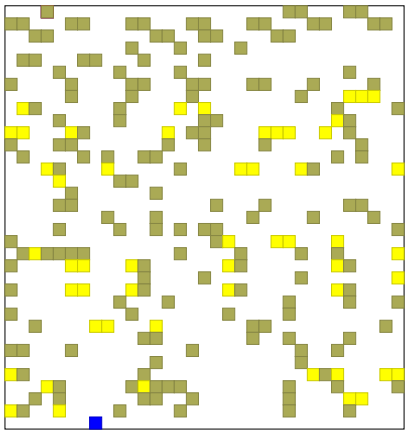

Fig. 5: Zero 1 Byte

Fig. 6: Zero 2 Bytes

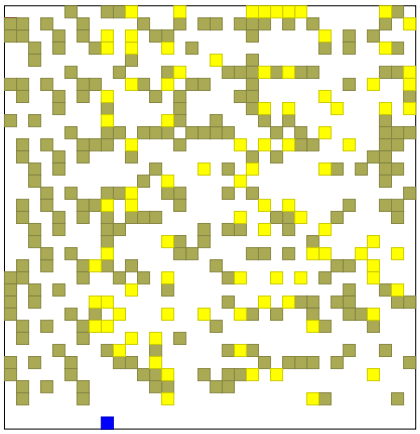

Fig. 7: NOP 1 Byte

F. Zero 2 Bytes

A similar test of automation over many mutants was the fault model that sets two consecutive bytes to zero. There are 1129 possible mutant binaries under this fault model. Their results shown in the **Z2B** column of Table I, and the map showing the starting index of the two bytes is shown in Figure 6.

Similar to the zero 1 byte fault injection model, no vulnerabilities were detected. Even more crashes were introduced, although a few less infinite loops. Generally this is due to instructions being damaged to yield failure, either by being simply incomprehensible, or by pushing memory access outside acceptable bounds.

G. NOP Code 1 Byte

The last fault model for this experiment was to set each byte to the instruction code for a non-operation (NOP). This fault injection attack was applied to each of the 1130 bytes to ensure complete coverage (and so in some cases had effects other than NOP'ing an instruction). Column **NOP** of Table I summarises these results, with the detailed map in Figure 7.

This approach turned out to be even more destructive than either of the zero byte fault models. Although more crashes were introduced, the almost doubling of infinite loops was an unexpected result that could be investigated further in future. No vulnerabilities were detected here which aligns with the prior results that binaries are fairly resistant to these kinds of byte attacks.

H. A Note on Scalability

The experiments were conducted on a variety of devices with different hardware and configurations (all were virtual machines running Ubuntu X64). The distribution was due to different experiments being run at different times, however this makes it impossible to provide consistent runtime information for the experiments.

That said, in general the model checking (either validation or checking) was by far the most expensive in terms of runtime. No attempt was made to optimise or modify the settings of LLBMC to improve runtime, despite some results taking many minutes. This is due to the process being trivially parallisable, since each mutant can be checked independently.

VI. Related Work

There are various related works that consider fault injection vulnerability and either formal methods or broad testing. This section discusses some key differences with respect to the process and approach used here.

Several recent works have considered formal approaches for handling fault injection vulnerabilities [9], [22], [23], [11], [10]. Both [9] and [10] use formal methods to prove the efficacy of countermeasures, the former for one specific implementation, and the latter as applied to extremely small assembly code fragments. However, they consider only a single fault model, and very limited capabilities of fault injection, the latter not even on the program as a whole. The works [22], [23], [11] all have similar proposals to the process here, albeit in more limited manners. The SymPLFIED framework [22] that exploits symbolic execution and model checking to detect fault vulnerabilities in MIPS assembly code. However, the SymPLFIED approach cannot operate on binaries or give concrete answers about the error, instead tracking "error" states as in taint analysis. Lazart presented in [23] operates only on LLVM-IR, and only considers fault injections upon control flow of the program. In [11] this is combined with the Embedded Fault Simulator (EFS) [24] to also

support binary level faults on the hardware, but this extension only supports NOP'ing instructions.

Less closely related works include: [25] that tests for robustness using both software and hardware fault injection, the latter to validate the former; and [26] that start from the hardware model and use this to simulate faults on the assembly code of a program.

VII. Conclusions and Future Work

Fault injection has recently been increasingly used to attack software applications, and test system robustness. This paper presents a formal process that uses model checking to detect fault injection vulnerabilities in binaries. This process supports the detection of many varieties of fault injection vulnerabilities, and does not rely on any particular system architecture, fault model, or other restricted choices (as are common in the literature).

Experimental results demonstrate the efficacy of the process by testing a variety of fault models on the PRESENT binary. For naive fault models the (Z1B, Z2B, and NOP) this yielded many infinite loops and program crashes, but no attacks to skip the encryption algorithm. However, when (both unconditional and conditional) jump instructions were targeted, 9 fault injection vulnerabilities were found that allow complete bypassing of the PRESENT algorithm.

A. Future Work

There are several limitations with the implementation chosen here that provide opportunities for future research. Indeed the implementation used was merely the easiest to combine effectively to implement the process.

The choice to use MC-Sema was to be able to work with LLVM-IR. The choice of LLVM-IR is due to this being a widely used intermediate representation that is supported by many tools. However, there are limitations with MC-Sema that may limit future work. MC-Sema supports only (some of) the instructions of X86 architecture [19] and so future work is to replace MC-Sema and/or expand to implement the process for other architectures.

The LLBMC model checker is sufficient for the safety properties but does not support liveness properties. Thus although LLBMC was sufficient for the proof of concept here, future work will exploit a non-bounded model checker that can also accept liveness properties. In particular a model checker that can produce traces (LLBMC can, but not combined with MC-Sema) would aide in understanding vulnerabilities and analysing results.

Fault injection was implemented with custom tools for this work, although there already exist several tools to simulate fault injection attacks on software [27], [28]. However, these tools are limited by various choices that make unsuitable for the process here (hence their lack of use in the implementation). Several are only able to inject faults into intermediate representations, and not into executable binaries [29], [23], [30], thus being unable to simulate faults that appear only at the executable binary level. Others have different limitations, such as: specific hardware platforms [22], [31], specific source code languages [32], [9], [30], or requiring simulating drivers [33]. Despite these limitations, many include useful techniques or developments that could be incorporated into future development of a general fault injection tool for executable binaries.

Complementary research is to explore ways to inject faults intelligently. This could exploit knowledge of the property to inject faults that would lead to property violations, yielding improved efficiency of experiments.

Regarding properties, another area of future work is to consider how to extract properties automatically from the binary (or source code). There is some existing work in this area [34], [35] although they focus upon high level behaviour rather than binary code.

Currently the process identifies vulnerabilities, but does not suggest fixes or countermeasures. Automatically generating countermeasures is non-trivial, although if countermeasures to particular faults are known future work could suggest or implement them automatically. Perhaps more significantly, these countermeasures could be checked immediately using the process here and so their effectiveness verified immediately.

There are also several directions related to case studies. The analysis of PRESENT here was proof-of-concept to demonstrate the process, and indeed some of the "passed" results here correspond to known differential attacks against PRESENT [36], [37]. Applying the process with more properties to consider, and a better suite of fault models would yield more complete results on the vulnerability of the PRESENT binary.

Future case studies could consider other security critical software, e.g. encryption algorithms, mission critical software, embedded device kernels, and also software that has implemented countermeasures.

References

[1] 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.

[2] S. Guilley, L. Sauvage, J.-L. Danger, and N. Selmane, "Fault injection resilience," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE, 2010, pp. 51–65.

[3] 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.

[4] 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*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

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

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

[7] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[8] 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.

[9] 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.

[10] 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.

[11] 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*. Springer, 2014, pp. 92–111.

[12] T. Given-Wilson, N. Jafri, J.-L. Lanet, and A. Legay, "An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT – Extended Version," Apr. 2017, working paper or preprint.

[13] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007, pp. 450–466.

[14] L. R. Knudsen and G. Leander, "Present–block cipher," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 953–955.

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

[16] 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.

[17] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[18] B. Gough, *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[19] Trail of bits, "Mc-semantics," 2016, https://github.com/trailofbits/mcsema.

[20] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded model checking of C and C++ programs using a compiler IR," in *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, ser. VSTTE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 146–161.

[21] C. Sinz, F. Merz, and S. Falke, "LLBMC: A bounded model checker for LLVM's intermediate representation - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, 2012, pp. 542–544.

[22] 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.

[23] 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.

[24] 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*. Springer, 2014, pp. 222–229.

[25] 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*. IEEE, 2002, pp. 21–25.

[26] 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.

[27] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[28] A. Johansson, "Software implemented fault injection used for software evaluation," *Building Reliable Component-Based Systems*, 2002.

[29] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[30] S. K. Vishal Chandra Sharma, Ganesh Gopalakrishnan, "Towards reseiliency evaluation of vector programs," in *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2016.

[31] 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.

[32] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny, "SmartCM a smart card fault injection simulator," in *2011 IEEE International Workshop on Information Forensics and Security*. IEEE, 2011, pp. 1–6.

[33] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 361–372.

[34] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Advanced verification by automatic property generation," *IET computers & digital techniques*, vol. 3, no. 4, pp. 338–353, 2009.

[35] Y. Zhu and H. Gao, "A novel approach to generate the property for web service verification from threat-driven model," *Appl. Math*, vol. 8, no. 2, pp. 657–664, 2014.

[36] 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.

[37] N. Bagheri, R. Ebrahimpour, and N. Ghaedi, "New differential fault analysis on present," *EURASIP Journal on Advances in Signal Processing*, vol. 2013, no. 1, p. 145, 2013.