HAL
open science

An automated and scalable formal process for detecting fault injection vulnerabilities in binaries

Thomas Given-Wilson, Annelie Heuser, Nisrine Jafri, Axel Legay

To cite this version:

Thomas Given-Wilson, Annelie Heuser, Nisrine Jafri, Axel Legay. An automated and scalable formal process for detecting fault injection vulnerabilities in binaries. Concurrency and Computation: Practice and Experience, 2018, pp.1-12. 10.1002/cpe.4794 . hal-01960940v2

HAL Id: hal-01960940

https://inria.hal.science/hal-01960940v2

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.

Received 15 October, 2017; | Revised: 00 Month 0000 | Accepted: 00 Month 0000

DOI: xxx/xxxx

An Automated and Scalable Formal Process for Detecting Fault Injection Vulnerabilities in Binaries

Thomas Given-Wilson*[1] | Annelie Heuser*[2] | Nisrine Jafri*[1] | Axel Legay*[1]

[1]Inria, France
[2]CNRS, France

Correspondence
*Corresponding authors. Email: {thomas.given-wilson, nisrine.jafri, jean-louis.lanet@inria.fr, axel.legay}@inria.fr, annelie.heuser@irisa.fr

Present Address
Centre de recherche Inria Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu
35042 Rennes, France

Abstract

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 an extension of a recently published general model checking based process to detect fault injection vulnerabilities in binaries. This new extension makes the general process scalable to real-world implementions which is demonstrated by detecting vulnerabilities in different cryptographic implementations.

KEYWORDS:
Fault Injection, Model Checking, Attack, Encryption, Vulnerability

1 | 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 extension of the automated formal process introduced in (11) for the detection of fault injection vulnerabilities in binaries. This extension overcomes limitations in the scalability of the pure model-based general process. In particular, it addresses cases where the full rigour of model checking is not required, such as when a program crash, infinite loop, or non-functional state can be rapidly approximated. The extension still accounts for many different kinds of fault injections and does not require specialised hardware or equipment. This process is achieved by simulating fault injection attacks upon the executable binary for the given

software, running pre-analysis to rapidly eliminate defective outcomes, and then using model checking on the remaining executable binaries 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 executable binary involves checking various properties using model checking to ensure the executable binary meets its specification. Fault injection attacks are then simulated on the executable binary, producing mutant binaries. The pre-analysis quickly rules out mutant binaries that are not suitable for model checking. The properties are then model checked on the mutant binaries that pass pre-analysis. A difference in the result between validating and checking the properties indicates a vulnerability to the fault injection attack that was simulated.

This extended 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 while being scalable to real-world implementations. 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 (12, 11). 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. The process design, and extension with pre-analysis, allow for easy parallelism and thus scalability of the process in practice.

To demonstrate the efficacy of this process, this paper includes a case study of applying the process with various fault models to two different cryptographic implementations: the PRESENT lightweight encryption algorithm (14, 15), and the recently introduced lightweight encryption algorithm SPECK (20).

The results found 82 vulnerabilities, with 9 in each of PRESENT and SPECK allowing an attacker to bypass the encryption entirely. A further 64 cryptanalytical attack vulnerabilities were found in PRESENT.

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 exploiting existing tools.
- The addition of pre-analysis to the process to significantly reduce model checking cost.
- A fault injection tool for injection various faults into executable binaries.
- A case study demonstrating the efficacy of using the process on PRESENT and SPECK.
- The identification of 73 fault injection vulnerabilities in PRESENT, 9 that bypass encryption entirely.
- The identification of 9 fault injection vulnerabilities in SPECK that bypass encryption entirely.

The structure of the paper is as follows. Section 2 recalls background on fault injection attacks, fault models, PRESENT & SPECK, model checking and properties. Section 3 introduces the extended process of this paper. Section 4 overviews the implementation and tools used to achieve the process. Section 5 applies the process to the PRESENT and SPECK and analyses the results. Section 6 discusses related work. Section 7 concludes.

2 | 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, fault models, cryptographic algorithms, and model checking & properties.

2.1 | 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 hardware effect of a fault injection attack is described through a fault model (see next section) that specifies the nature and scope of the induced modification. Typically such attacks are achieved by changing a value stored in the hardware, e.g. the value of a 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 in software, the rest of this paper only considers software simulations.

2.2 | Fault Models

This section details the fault models that are used in the experiments. Each fault model describes an attack that can be conducted through a hardware or software fault injection.

The **FLP** fault model simulates the flipping of a single bit, either from 0 to 1 or from 1 to 0. This fault model is highly representative of many kinds of faults that can be induced, ranging from those due to atmospheric radiation, to software effects such as the rowhammer attack (4).

The **Z1B** fault model simulates setting a single byte to zero (regardless of prior value). This fault model reflects a more malicious attack in general, and corresponds to a commonly achievable attack in practice (16, 17).

The **Z1W** fault model represents setting an entire word to zero (again regardless of prior value). This is similar in concept to the **Z1B** fault model and attack, but captures behaviour more related to the hardware model, since it reflects faulting some piece of the hardware that operates on words rather than bits or bytes (such as the bus).

The **NOP** fault model sets the targeted operation to a non-operation instruction for the chosen architecture (in this case 0x90). The concept behind this model is that it simulates skipping an instruction, a common effect of many runtime faults (10). However, due to the inconsistent alignment of instructions in X86 this fault model may also change parts of other instructions or values when the alignment does not match.

Both the **JMP** fault model and the **JBE** fault model simulate faulting the target address of a jump instruction. In practice there are multiple jump instructions, either unconditional jumps **JMP** or conditional jumps **JBE**. In both cases, the fault model simulates any possible change to the jump target address. This fault model corresponds to targeted attacks that attempt to bypass code, or significantly disrupt the control flow (18, 19). Note that this is a superset of other faults that may target the same bits of the target address.

2.3 | Encryption Algorithms

This section gives a brief overview of the the encryption algorithms considered in the experiments. Both implementations are open-source and consider a standard setting (e.g. without additional countermeasures).

PRESENT (14, 15) 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[1] 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) as used in (11).

SPECK (20) is a lightweight Feistel network cipher recently published. The SPECK algorithm is highly software-oriented as it relies only on addition, word rotation, and Xor operations. Each round consists of 2 rotations, addition of the right word to the left word, Xoring the key into the left word, and Xoring the left word to the right word. SPECK supports a variety of block and key sizes with respective number of rounds. The canonical version[2] analysed here is implemented in C, has a key length 128 bits, block size 64, and 21 rounds.

2.4 | Model Checking & Properties

Model checking is a formal method for determining whether properties hold on a model (21). 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.

Model checking must operate on properties that define good or bad behaviour of the model. Here properties are used to define correct behaviour, and also specific vulnerabilities that may be introduced by fault injection. Here three properties were considered that can be applied to the encryption algorithms.

Property 1 is to check whether the encryption was conducted successfully, i.e. the ciphertext at the end of the encryption function corresponded was correct. This allows the detection of any kind of damage to the encryption algorithm by fault injection. However, further properties are considered to examine more specific vulnerabilities.

Property 2 checks whether the ciphertext is equal to the plaintext, i.e. the encryption can be bypassed by fault injection. This is an extreme vulnerability where a fault is able to entirely skip the encryption and render the binary useless (albeit still appearing to execute without error state and produce a "ciphertext").

Property 3 is specific to each encryption algorithm, and results in a key recovery attack when combined with a cryptanalytical attack (CA). The CAs are: for PRESENT skipping the last (31st) round (22, 23), and for SPECK to output the result of implementing only 9 or 10 rounds (24).

A Note on Efficiency

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 (25).

Bounded model checking is a refinement of model checking that alleviates some of the issues with possibly infinite complexity by bounding the checking (26). The key idea in bounded model checking is to put a bound on parts of the model that could be infinite (or at least extremely large). In practice this allows potentially infinite loops to be checked (exited early), at the loss of some precision in the results.

3 | PROCESS

This section discusses the process used to detect fault injection vulnerabilities. In (11) a simpler version of this process was presented, here a pre-analysis step is added that improves efficiency.

[1] Available at http://www.lightweightcrypto.org/implementations.php
[2] Available at https://github.com/inmcm/Simon_Speck_Ciphers

An overview of the (extended) process is as follows. 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. Pre-analysis is used to filter out mutants that fail to execute, or fail to yield an output. Mutants that execute and produce an output are then passed to model checking. A difference in the results of validation and checking the properties indicates a vulnerability to the simulated fault injection. An illustration of the process is given in Figure 1 with the new extensions here shown in red. The rest of this section details and discusses this process.

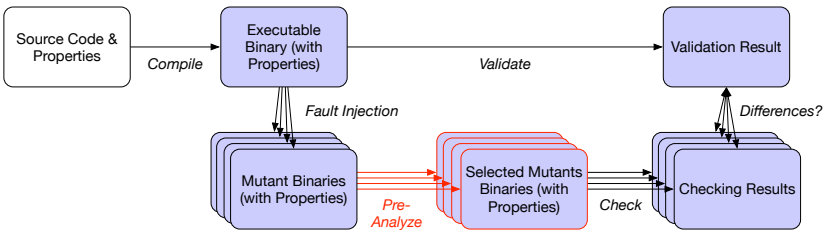

FIGURE 1 Process Diagram

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. For further discussion refer to (11).

Since this paper considers fault injection attacks upon the binary it is necessary to compile the source code into an executable binary. This executable binary represents the software application that would be executed by the system in practice. For the process here the compilation must maintain the properties 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 and thus if there is some other error in the source code, it can be detected before fault injection 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 the actual fault injection attacks and produces mutant executable binaries that represent the executable binary after the attacks have been effected. For simplicity here, the fault injection is chosen to always avoid the annotations of the properties. This is again discussed in more detail in (11).

Pre-analysis is performed on the mutant to quickly eliminate mutants that fail to execute or fail to produce an output. Pre-analysis includes detection of: crashes, infinite loops, and error codes. This saves model checking effort when there is already evidence that the properties will fail to hold.

Mutants that pass pre-analysis are then checked using model checking. Differences in the properties between the validation and the checking indicates that the fault that was injected yields a change in behaviour that violates the properties, and so could be exploited by an attacker.

4 | IMPLEMENTATION

This section presents the implementation of the process from the previous section. 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 (11).

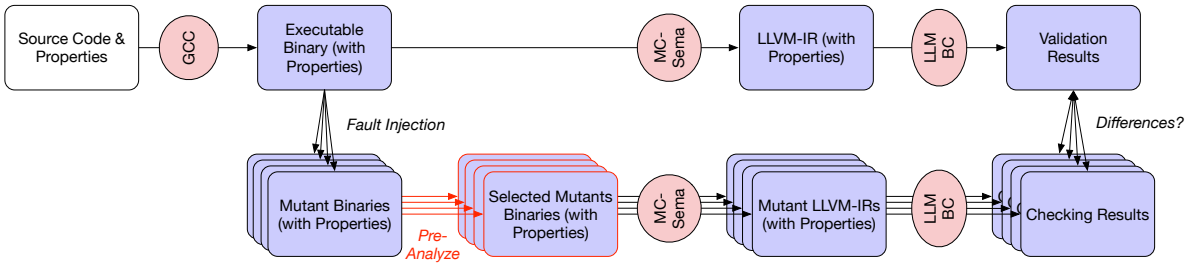

FIGURE 2 Implementation Diagram

An overview of the implementation is as follows, and shown in Figure 2 , again extensions from (11) shown in red. The implementation begins with the source code written in the C language and the properties represented in the source code by tool specific annotations. The source code and properties are compiled to an executable binary by the GNU C Compiler (GCC) (27). 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 (28). The properties are then checked on the intermediate representation using the Low Level Bounded Model Checker (LLBMC) (29, 30). An automated fault injection tool written in python (see below) then produces mutant binaries by injecting faults into the executable binary according to the tool's fault model (**?**). Pre-analysis is then performed that executes the mutant binary and checks for various failures or non-output states. If pre-analysis is passed then the mutant binary is transformed via MC-Sema and LLVM-IR to be checked by LLBMC as above. Finally, the results of model checking the executable binary and each mutant binary are compared for differences by matching the outputs of LLBMC.

Fault Injection Tool

The Fault-Injection-Tool was developed in python and released as part of the development of this paper. This tool allows injection of all fault models used in this paper, and also byte tamper that allows faulting an arbitrary byte in the executable binary to any value.

5 | CASE STUDIES: PRESENT & SPECK

This section presents the results of experiments on PRESENT and SPECK algorithms. This includes the results of both the pre-analysis and the model checking. The experiment design is presented first, then overviews of pre-analysis and model checking, and finally vulnerabilities are discussed in PRESENT and SPECK.

5.1 | Experiment Design

Each experiment fixes one of the six fault models, one of the two encryption algorithms, and one of the three properties for that encryption algorithm. Then the process is applied, testing all possible mutations that can be produced by the fault model, applied to the binary for the encryption algorithm, and testing for the property chosen. This yields thirty-six different experiments (although results are merged in following sections).

Each experiment was run on a twin Intel Xeon E5-2660 with 2×14 cores (maximum of 56 parallel threads) with 128GB of memory. The operating system is Ubuntu 16.04 (kernel version 4.4.0-21). The experiments are limited to a maximum of 17 parallel instances since LLBMC uses approximately 7GB of memory, and thus this prevents any possibility of memory exhaustion. Multiple experiments were run in parallel on identical hardware instances.

	PRESENT						SPECK					
	FLP	**Z1B**	**Z1W**	**NOP**	**JMP**	**JBE**	**FLP**	**Z1B**	**Z1W**	**NOP**	**JMP**	**JBE**
Model Check	4797	411	3	400	782	613	4199	492	38	316	230	83
Infinite Loop	464	44	0	63	545	69	80	11	1	7	72	0
Illegal Instruction	473	5	0	55	26	63	315	2	1	38	16	8
Aborted	355	23	2	49	187	141	110	6	4	8	14	8
Segmentation Fault	3099	666	1144	589	1263	1639	3525	519	986	659	429	155
Other Errors	4	0	0	0	2552	2830	11	0	0	2	259	766

TABLE 1 Overview of Fault Injection Pre-Analyse Results.

5.2 | Pre-Analysis Results

The result of the pre-analysis can be seen in Table 1 . The effect of the different fault models is fairly consistent on both algorithms.

The **FLP** fault model yields many different behaviours, but the vast majority still produce output that needs to be model checked to determine the effect. Infinite loops, illegal instructions, and aborted results from execution are common, but all minority results. Segmentation faults are highly likely, but not as likely as an output. Other errors are very rare.

Both the **Z1B** and **NOP** fault models were highly likely to cause a segmentation fault. The next most likely outcome was an output that required model checking. Infinite loop, illegal instruction, and aborted were very rare, and other errors extremely rare (only evident in SPECK). The similarity of results for these fault models is not surprising, since they both change the value of a byte in the same manner.

The **Z1W** fault model almost always caused a segmentation fault. All other outcomes were extremely rare or non-evident. In particular, this meant that very few required model checking and so the pre-analysis was highly effective at reducing the cost here.

The **JMP** and **JBE** fault models predominantly produced other error or segmentation fault, although a large number of mutants still required model checking. Many segmentation fault are expected since X86 has variable size instructions and often jumps target the middle of instructions, yielding invalid instructions or arguments.

Observe that the pre-analysis reduced the total number of mutants requiring model checking from 110157 to 37089, thus reducing the model checking effort by 66.33%.

5.3 | Model Checking Overview

An overview of the results of model checking can be seen in Table 2 . Observe that the most interesting result of each property has been highlighted in the table. The rest of this section overviews these results in a broad sense, while each encryption algorithm is discussed in detail in the following sections.

The validity of the model checking results was manually verified by randomly selected samples from the outcomes in Table 2 . Several[3] mutants were checked from each combination of: property, encryption algorithm, and fault model. This resulted in manual verification of the results for 265 mutants to ensure the results were correct.

The evidence from Property 1 suggests that the investigated implementations of both PRESENT and SPECK are somewhat resistant to fault injection attacks, with approximately 25.78% of mutants still yielding correct output. Conversely 74.22% produced some kind of output, usually a faulty output that does not match the vulnerabilities

[3]Between 10 and 15 mutants, or all mutants if the total was less than 10.

		PRESENT						SPECK					
		FLP	**Z1B**	**Z1W**	**NOP**	**JMP**	**JBE**	**FLP**	**Z1B**	**Z1W**	**NOP**	**JMP**	**JBE**
Prop. 1	Good Ciphertext	1080	52	0	40	96	82	1536	205	1	86	10	0
	Bad Ciphertext	2249	116	1	173	532	362	2514	264	31	188	216	83
	Crashed	1471	239	2	186	154	169	149	24	6	42	4	0
Prop. 2	Leak Plaintext	0	0	0	0	1	8	3	0	0	2	2	2
	Not Leak Plaintext	3285	162	1	201	439	445	671	294	17	224	88	2
	Crashed	1472	246	2	190	153	91	3433	175	8	64	4	0
Prop. 3	CA Vulnerable	48	0	0	5	10	1	0	0	0	0	0	0
	Not CA Vulnerable	3247	163	1	199	430	447	2244	401	17	255	90	4
	Crashed	1466	245	2	186	152	90	1860	49	8	65	4	0

TABLE 2 Overview of Fault Injection Model Checking Results.

considered here[4]. Although Property 1 is specified to identify any incorrect output, the correct outputs are those of most significance here due to overlap between "Bad Ciphertext" and the other two properties.

Property 2 identifies the most severe outcome considered here; when the encryption algorithm appears to operate correctly but outputs the plaintext. Observe that there are very few fault injection attacks that can completely bypass the encryption algorithm, only 9 for PRESENT and 9 for SPECK. Detailed discussion follows in later sections.

Property 3 identifies specific CA vulnerabilities that an attacker can use to learn information about the key (23, 32). These tend to be specific to the behaviour of the algorithm considered, e.g. skipping a specific iteration of a loop, yet still significant to security. These turned out to be achievable frequently (64 occurrences) and with most fault models upon PRESENT, but not achievable with any of the fault models considered here on SPECK.

Observe that several crash results occurred during model checking. These are due to various different outcomes, all listed as "Crashed" in Table 2 . In 8.08% of cases these crashes were due to MC-Sema failing to parse the mutant into LLVM-IR. The vast majority, 91.9%, were due to failures of LLBMC, in particular segmentation faults in the SMT solver (STP with MiniSAT). Lastly, in the remaining 0.02% of cases LLBMC was unable to produce reliable output, stating that multiple mutually exclusive properties held[5].

5.4 | PRESENT Vulnerabilities

This section explores the 73 vulnerabilities found in PRESENT. Only a small number of these (9 occurrences) were found to violate Property 2 and so yield the plaintext. The majority (64 occurrences) were CA vulnerabilities that allow the key to be calculated from a number of ciphertexts.

All Property 2 vulnerabilities occurred from modifications to jump instructions in the code, 1 for **JMP** and 8 for **JBE**. In practice these all succeeded by changing the control flow to skip the entire algorithm and appear in only 2 places in the binary labeled with "2" in Figure 3 .

The **JMP** occurs early in the code as a wrapper around the main encryption algorithm, and can be exploited to jump past the encryption. There is only 1 vulnerability here because any earlier target would still execute some part of the encryption (or crash), and any later target would jump beyond the limits of the code.

The **JBE** vulnerabilities are all from the same address, albeit there are multiple targets that the jump can be modified to and still yield successful execution. This jump is from the comparison for the first loop, and can instead be modified to jump past the first and second encryption loops (almost to the end of the execution). There are

[4]This does not preclude these outputs corresponding to a different vulnerabilities such as (31, 24).

[5]This is clearly a limitation of LLBMC and was not attempted to be rectified here. Despite LLBMC being bounded, these results did not indicate the bound was reached. Many of these results were manually verified and found to only hold true for Property 1, i.e. produce a bad ciphertex.

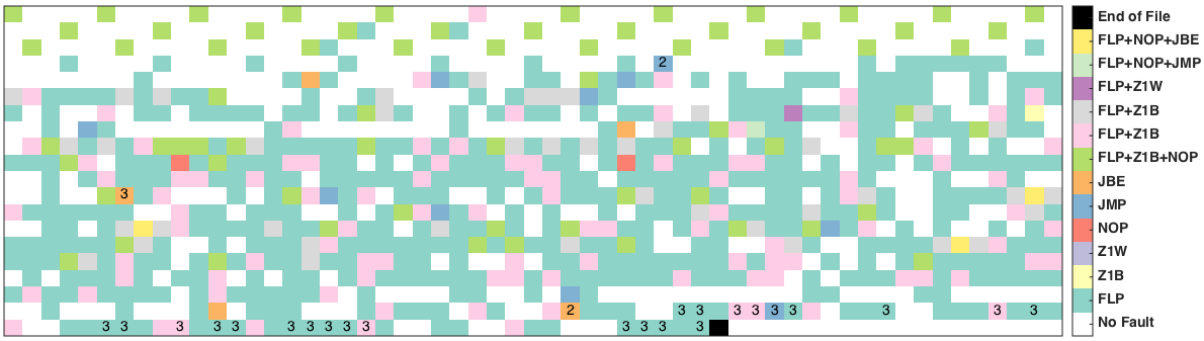

FIGURE 3 Model Checking Results for PRESENT

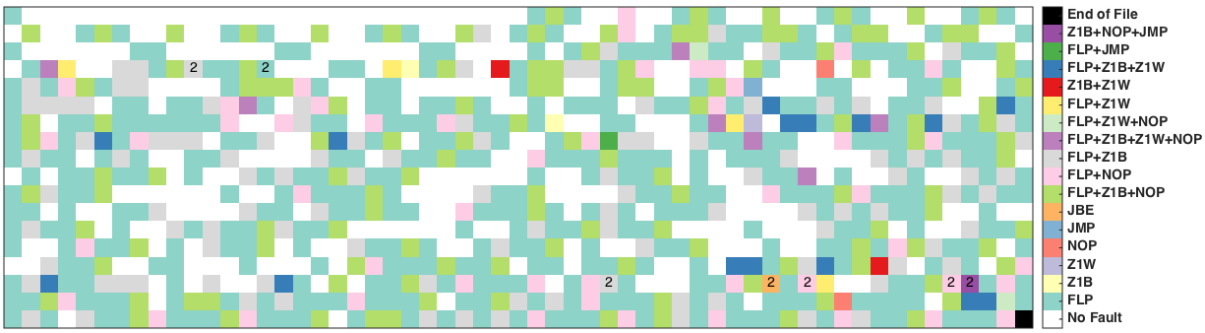

FIGURE 4 Model Checking Results for SPECK

multiple targets here since there are several places in the last loop of the algorithm that do not change the output (and the conditional for exiting the loop holds due to incorrect stack pointer offsets). There are also targets beyond the end of the loop, although these can lead to program crashes depending on the target address.

Property 3 vulnerabilities, labeled "3" in Figure 3 , are more common and can be achieved with a larger variety of fault models. Like the **JMP** of Property 2, the **JBE** here has a single address that can be jumped to to yield a CA vulnerability. There are however multiple **JMP** vulnerabilities, all derived from changes to the exit jump of the main encryption loop being re-targeted to skip over the last round of encryption.

The **FLP** and **NOP** vulnerabilities to Property 3 are all located towards the end of the binary, since this is where the last round of encryption is executed. The majority of these are minor changes to various instructions that prevent the last round being properly executed. These include: modifying the loop index in the last round (e.g. from 0 to 32), changing registers (e.g. loading to/from `%eax` instead of `%ecx`), changing values (e.g. changing while loop bound from > 7 to > -7), changing the comparison instruction to something else (e.g. `cmp` to `sub`), etc.

5.5 | SPECK Vulnerabilities

This section explores the 9 vulnerabilities found in SPECK. All of these were found to violate Property 2 and so yield the plaintext. The lack of Property 3 vulnerabilities is discussed below.

The Property 2 violations were obtained with 4 fault models, **FLP**, **NOP**, **JBE**, and **JMP**. The **FLP** vulnerabilities occurred due to: 1 damaging the stack pointer and so avoiding loops and skipping the whole algorithm, the other 2 changed the round key to prevent the encryption acting effectively.

The **NOP** vulnerabilities occurred indirectly by inserting the value for a **NOP** instruction (0x90) into another instruction. The first, in the jump instruction for the key generation loop and so skipping past the key generation. The second, in the encryption loop and so initialised the loop iterator to 0x90, immediately skipping the encryption.

The **JBE** and **JMP** both skip the encryption loop by jumping forward to the same 2 locations after the encryption has completed. The **JBE** is at the end of key generation, and the **JMP** at the beginning of the encryption.

Property 3 vulnerabilities were not found with any of the fault models tested here. This is unfortunate but not surprising since the implementation does not have simple code path that can be exploited to yield this particular vulnerability. (Unlike PRESENT where the last round is executed in a separate loop.)

The most likely candidate fault model would have been **FLP** where some simple flip of a variable or value could have altered the number of encryption rounds executed. Unfortunately, since 21 rounds are executed and CA vulnerabilities have been published for 9 or 10 rounds, the numbers 21 and 9 or 10 are not a single but flip apart. However, to validate the experiments, a manual fault injection of setting the loop bound value to 10 was tested and found to be CA vulnerable with the process.

6 | RELATED WORK

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

Several recent works have considered formal approaches and fault injection vulnerabilities (9, 33, 19, 12, 10). Both (9) and (10) use formal methods to prove the efficacy of countermeasures, in (9) for one specific implementation, and in (10) 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 (33, 19, 12) all have similar proposals to the process here, albeit in more limited manners. The SymPLFIED framework (33) 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 (19) operates only on LLVM-IR, and only considers fault injections upon control flow of the program. In (12) this is combined with the Embedded Fault Simulator (EFS) (34) to also support binary level faults on the hardware, but this extension only supports NOP'ing instructions.

Less closely related works include: (35) that tests for robustness using both software and hardware fault injection (validating software with hardware); and (36) that starts from the hardware model and use this to simulate faults on the assembly code of a program.

7 | CONCLUSIONS

Fault injection has recently been increasingly used to attack software applications, and test system robustness. This paper presents an extended scalable 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).

Overall the process is scalable via parallelism, although model checking is still expensive. The addition of pre-analysis to the process here allowed many fault injections to be easily ignored as yielding failed program states, thus improving the efficacy of the process as a whole. Pre-analysis reduced the number of mutants requiring model checking by 66.33% here.

The addition of a fault injection tool (**?**) here allows other experiments to easily automate fault injection attacks. This was demonstrated in the experiments here.

The results of applying this process yielded 82 vulnerabilities, 73 in PRESENT and 9 in SPECK. Both PRESENT (9 vulnerabilities) and SPECK (4 vulnerabilities) were vulnerable to faulting jump instructions that allowed encryption to be entirely bypassed. For SPECK a further 5 vulnerabilities exist, 2 by inserting non-operation byte values, and 3 by flipping individual bits. PRESENT was also found to be vulnerable to 64 different CA vulnerabilities, mostly through bit flips, but also through nopping instructions and jump target modifications. These were all found towards the end of the binary, where the last round of encryption occurs. SPECK was not found to be vulnerable to the CA vulnerabilities tested here. This is mostly due to the number of rounds (21) to achieve encryption and that 21 is not one bit flip away from either 9 or 10. These results indicate that some kinds of attacks may be more or less achievable depending on the structure of the code used, and so some care should be taken when choosing how to implement a fault resistant binary.

References

[1] Bar-El Hagai, Choukri Hamid, Naccache David, Tunstall Michael, Whelan Claire. The Sorcerer's Apprentice Guide to Fault Attacks.. *IACR Cryptology ePrint Archive.* 2004;2004:100.

[2] Guilley Sylvain, Sauvage Laurent, Danger Jean-Luc, Selmane Nidhal. Fault injection resilience. In: :51–65IEEE; 2010.

[3] Verbauwhede Ingrid, Karaklajic Dusko, Schmidt Jorn-Marc. The fault attack jungle-a classification model to guide you. In: :3–8IEEE; 2011.

[4] Kim Yoongu, Daly Ross, Kim Jeremie, et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: :361–372IEEE Press; 2014.

[5] Seaborn Mark, Dullien Thomas. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat.* 2015;.

[6] Yim Keun Soo. The Rowhammer Attack Injection Methodology. In: :1–10IEEE; 2016.

[7] Barenghi Alessandro, Breveglieri Luca, Koren Israel, Naccache David. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE.* 2012;100(11):3056–3076.

[8] Moro Nicolas, Dehbaoui Amine, Heydemann Karine, Robisson Bruno, Encrenaz Emmanuelle. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In: :77–88IEEE; 2013.

[9] Christofi Maria, Chetali Boutheina, Goubin Louis. Formal verification of an implementation of CRT-RSA Vigilant's algorithm. In: :28; 2013.

[10] Moro Nicolas, Heydemann Karine, Encrenaz Emmanuelle, Robisson Bruno. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering.* 2014;4(3):145–156.

[11] Given-Wilson Thomas, Jafri Nisrine, Lanet Jean-Louis, Legay Axel. An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT. In: :293–300IEEE; 2017.

[12] Rivière Lionel, Potet Marie-Laure, Le Thanh-Ha, Bringer Julien, Chabanne Hervé, Puys Maxime. Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks. In: :92–111Springer; 2014.

[13] Given-Wilson Thomas, Jafri Nisrine, Lanet Jean-Louis, Legay Axel. An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT – Extended Version working paper or preprint2017.

[14] Bogdanov Andrey, Knudsen Lars R, Leander Gregor, et al. PRESENT: An ultra-lightweight block cipher. In: :450–466Springer; 2007.

[15] Knudsen Lars R, Leander Gregor. PRESENT–Block Cipher. In: Springer 2011 (pp. 953–955).

[16] Tunstall Michael, Mukhopadhyay Debdeep, Ali Subidh. Differential Fault Analysis of the Advanced Encryption Standard Using a Single Fault.. *WISTP.* 2011;6633:224–233.

[17] Roscian Cyril, Dutertre Jean-Max, Tria Assia. Frontside laser fault injection on cryptosystems-Application to the AES'last round. In: :119–124IEEE; 2013.

[18] Dureuil Louis, Petiot Guillaume, Potet Marie-Laure, Le Thanh-Ha, Crohen Aude, Choudens Philippe. FISSC: A Fault Injection and Simulation Secure Collection. In: :3–11Springer; 2016.

[19] Potet Marie-Laure, Mounier Laurent, Puys Maxime, Dureuil Louis. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In: :213–222IEEE; 2014.

[20] Beaulieu Ray, Shors Douglas, Smith Jason, Treatman-Clark Stefan, Weeks Bryan, Wingers Louis. The SIMON and SPECK Families of Lightweight Block Ciphers Cryptology ePrint Archive, Report 2013/404http://eprint.iacr.org/2013/404; 2013.

[21] Baier Christel, Katoen Joost-Pieter, Larsen Kim Guldstrand. *Principles of model checking*. MIT press; 2008.

[22] Ghalaty Nahid Farhady, Yuce Bilgiday, Schaumont Patrick. Differential fault intensity analysis on PRESENT and LED block ciphers. In: :174–188Springer; 2015.

[23] Wang Gaoli, Wang Shaohui. Differential fault analysis on PRESENT key schedule. In: :362–366IEEE; 2010.

[24] Dinur Itai. Improved Differential Cryptanalysis of Round-Reduced Speck. In: Joux Antoine, Youssef Amr M., eds. *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, Lecture Notes in Computer Science, vol. 8781: :147–164Springer; 2014.

[25] Kinder Johannes, Katzenbeisser Stefan, Schallhart Christian, Veith Helmut. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing.* 2010;7(4):424–438.

[26] Biere Armin, Cimatti Alessandro, Clarke Edmund M, Strichman Ofer, Zhu Yunshan. Bounded model checking. *Advances in computers.* 2003;58:117–148.

[27] Gough Brian. *GNU scientific library reference manual*. Network Theory Ltd.; 2009.

[28] Trail of bits . *Mc-Semantics.* https://github.com/trailofbits/mcsema; 2016.

[29] Merz Florian, Falke Stephan, Sinz Carsten. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: VSTTE'12:146–161Springer-Verlag; 2012; Berlin, Heidelberg.

[30] Sinz Carsten, Merz Florian, Falke Stephan. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation - (Competition Contribution). In: :542–544; 2012.

[31] Özen Onur, Varici Kerem, Tezcan Cihangir, Kocair Çelebi. Lightweight Block Ciphers Revisited: Cryptanalysis of Reduced Round PRESENT and HIGHT. In: Boyd Colin, Nieto Juan Manuel González, eds. *Information Security and Privacy, 14th Australasian Conference, ACISP 2009, Brisbane, Australia, July 1-3, 2009, Proceedings*, Lecture Notes in Computer Science, vol. 5594: :90–107Springer; 2009.

[32] Biryukov Alex, Roy Arnab, Velichkov Vesselin. Differential analysis of block ciphers SIMON and SPECK. In: :546–570Springer; 2014.

[33] Pattabiraman Karthik, Nakka Nithin, Kalbarczyk Zbigniew, Iyer Ravishankar. SymPLFIED: Symbolic program-level fault injection and error detection framework. In: :472–481IEEE; 2008.

[34] Berthier Maël, Bringer Julien, Chabanne Hervé, Le Thanh-Ha, Rivière Lionel, Servant Victor. Idea: embedded fault injection simulator on smartcard. In: :222–229Springer; 2014.

[35] Ademaj Astrit, Grillinger Petr, Herout Pavel, Hlavicka Jan. Fault tolerance evaluation using two software based fault injection methods. In: :21–25IEEE; 2002.

[36] Dureuil Louis, Potet Marie-Laure, Choudens Philippe, Dumas Cécile, Clédière Jessy. From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference. In: :107–124Springer; 2015.