FaultRISC-V: Detecting Fault Injection Vulnerabilities in RISC-V Assembly

Prateek Kharangate[†], Brayden Sheafer[§], Guillermo Rached[†], Harris Musungu[‡], Boyang Wang[†]
[†]University of Cincinnati, [§]Cedarville University, [‡]Rochester Institute of Technology
{*kharanpv, rachedge*}*@mail.uc.edu, bsheafer@cedarville.edu, harrismusungu@gmail.com, boyang.wang@uc.edu*

***Abstract*—Fault injection attacks can alter bits by manipulating voltage, temperature, or electromagnetic (EM) radiation on a target, such as a microcontroller. Altered bits can potentially lead to changes in program execution, such as bypassing a secure boot. However, tools for automatically detecting these vulnerabilities in assembly code are limited. This paper introduces FaultRISC-V, a static analysis tool designed to automatically detect four types of vulnerable code patterns in RISC-V assembly under fault injection attacks. It involves (1) parsing and (2) token matching, with a customized RISC-V assembly parser and specific token matching rules. We established a dataset of 57 RISC-V assembly files related to secure boot and embedded systems authentication, consisting of 46,111 lines of assembly code in total across three optimization levels (including O0, O1, and O2). Our evaluation results demonstrate the effectiveness and efficiency of our tool over O0, O1, and O2. For instance, our tool achieves 96.4% precision and 97.7% recall over assembly files generated from O0. The average time to parse an assembly file and report the detection results is less than 200 milliseconds.**

I. Introduction

Fault injection attacks [1], [2], can manipulate factors such as voltage, temperature [3], [4], or electromagnetic (EM) radiation [5], [6] on a target device (e.g., a microcontroller) during program execution. These manipulations can cause bit flips (e.g., changing a 0 to a 1) or instruction skips, thereby altering program behavior and causing the target to malfunction. Real-world examples of fault injection attacks include revealing encryption keys [7], bypassing secure boot [1], and altering flash memory [6].

To mitigate fault injection attacks, one approach is to write code that is more resilient to such attacks [8]. For example, using two complex values with a greater Hamming distance (e.g., `0x7A3F` and `0x85C0`) to represent False and True is preferable to using simple values 0 and 1 as more multiple bits need to be flipped rather than a single bit.

A tool named FaultARM [9] was developed to automatically detect vulnerable ARM assembly instructions under fault injection attacks. While the tool is efficient and effective, it does not support detection over RISC-V assembly.

Our Contributions. We present a tool named FaultRISC-V, which can automatically identify vulnerable lines in RISC-V assembly code that are susceptible to fault injection attacks. Similarly to the design methodology of FaultARM, our tool consists of two phases, including parsing and token matching. Specifically, we develop a parser that processes an assembly file and converts it into a list of tokens, which includes elements such as instructions, functions, attributes, global variables, and locations. Our parser can further break down these tokens into registers, integers, strings, memory addresses, and labels. Following this, we implement specific token matching rules to identify each vulnerable pattern. Our tool is capable of detecting four types of vulnerable patterns: Branch, Bypass, Constant Coding, and Loop Check, which are defined in [1]. In addition, our tool can automatically detect the optimization level (either O0, O1, or O2) and customize detection at each given optimization level when needed. *In essence, FaultRISC-V is a static analysis tool, which efficiently identifies vulnerable code to fault injections without executing binaries.*

We established a labeled dataset of 57 RISC-V assembly files (46,111 lines in total) and evaluated the detection performance of our tool across three optimization levels, including O0 (19 files, 17,726 lines of assembly code), O1 (19 files, 12,747 lines of assembly code) and O2 (19 files, 15,638 lines of assembly code). The results show that FaultRISC-V is both effective and efficient. Specifically, it achieves 96.4% precision and 97.7% recall in O0, 98.5% precision and 96.8% recall in O1, and 97.4% precision and 96.1% recall in O2. Our tool can process an assembly file and report vulnerable lines within 200 milliseconds on average. Our findings continue to support the feasibility of enhancing the resilience of embedded systems against fault injection attacks early in the software development stage.

Reproducibility. Source code and dataset can be found at `https://github.com/UCdasec/FaultRISC-V`.

II. Background

Riscure (now a part of Keysight), published a whitepaper [1] that outlines 11 vulnerable patterns in C that are susceptible to fault injection attacks. For each vulnerable pattern, the whitepaper also offers resilient coding practices to help mitigate these attacks in C. *Essentially, these resilient coding examples ensure critical paths or data are double-checked, making it necessary for an attacker to flip multiple bits instead of flipping single bits.*

We provide concrete code examples associated with four vulnerable patterns, including Branch, Constant Coding,

Loop Check, and Bypass, which were defined in the Riscure whitepaper [1]. For each pattern, we provide vulnerable and resilient examples in both C and RISC-V assembly. The code examples in C are similar to the one suggested in the Riscure whitepaper. The code examples in RISC-V are compiled by us from the C code using the RISC-V toolchain.

Branch. This vulnerability arises when Boolean values are used in an `if` statement. An attacker can flip a single bit to change a Boolean value from one state to another (e.g., from 1 to 0). Using more complex numerical values to represent different states in an `if` statement is considered more resilient, as it would require flipping multiple bits rather than just one. Given this pattern, a vulnerable and a resilient example in C are shown in List. 1 and List. 2 respectively.

```
if(flag == 1){ // flag is 0 or 1
    // Critical Code, e.g., secure boot
}
```

Listing 1: Vulnerable Example (Branch, C)

```
if(flag == 0x3CA5){ // flag is 0x3CA5 or 0xC35A
    // Critical Code, e.g., secure boot
}
```

Listing 2: Resillient Example (Branch, C)

An `if` statement in C can be associated with any of the conditional branch statements, such as `bne`, `blt`, `beq`, and `bgt` in RISC-V assembly. If we compile the above vulnerable C code into RISC-V assmebly, we can observe that a conditional branch instruction includes a register `a4` with a trivial value `1` (stored in register `a5`) as shown in List. 3. On the other hand, the resilient example compares register `a4` with a non-trivial value `15,525` (stored in register `a5`) in List. 4.

```
li      a5,1
bne     a4,a5,.L2 //L2 jumps to critical code
```

Listing 3: Vulnerable Example (Branch, RISC-V, O0)

```
li      a5,16384
addi    a5,a5,-859  //16384 - 859 = 15525 => 0x3ca5
bne     a4,a5,.L2  //L2 jumps to critical code
```

Listing 4: Resillient Example (Branch, RISC-V, O0)

Constant Coding. This vulnerable pattern pertains to sensitive constants that have a limited range of values or states, such as `0`, `1`, `0xFF`. These constant values can be easily altered from one to another within the set by flipping a single bit. In contrast, non-trivial numerical values that exhibit a greater Hamming distance between two states are considered more resilient against fault injection attacks. This pattern is similar to Branch. However, instead of focusing on `if` statements, Constant Coding targets constant variables (e.g., static variables). We provide a vulnerable example and a resilient example in C below.

```
static short STATE_INIT = 0;  //global
static short STATE_LOCKED = 1; //global
```

Listing 5: Vulnerable Example (Constant Coding, C)

```
static short STATE_INIT = 0x5A3C; //global
static short STATE_LOCKED = 0xC3A5; //global
```

Listing 6: Resilient Example (Constant Coding, C)

The distinction between trivial values and non-trivial values is also evident in RISC-V assembly. The corresponding vulnerable and resilient examples in RISC-V assembly are shown in List. 7 and List. 8.

```
STATE_INIT:
        .half  0
STATE_LOCKED:
        .half  1
```

Listing 7: Vulnerable Example (Constant Coding, RISC-V, O0)

```
STATE_INIT:
        .half 23100
STATE_LOCKED:
        .half  -15451
```

Listing 8: Resilient Example (Constant Coding, RISC-V, O0)

Loop Check. This vulnerability occurs when the execution of a `for` loop is not followed by a verification of completion (usually with an `if` condition) to confirm that the loop has completed with the expected number of iterations. If the completion of a loop is not verified, an attacker could flip bits on the counter, potentially resulting in the last few iterations being skipped and causing data or state to become incorrect.

```
int i = 0, sum = 1;
for (i = 0; i<10; i++) {
    sum++;
}
// missing loop check
foo(sum);
```

Listing 9: Vulnerable Example (Loop Check, C)

```
int i = 0, sum = 1;
for (i = 0; i<=10; i++) {
    sum++;
}
// double check for loop is completed
if (i==10) {
    foo(sum);
}
```

Listing 10: Resilient Example (Loop Check, C)

In the corresponding examples in RISC-V assembly below, the vulnerable example fails to compare the value of the iterating variable (register `a4`) again after the initial comparison in the `ble` instruction. In contrast, the resilient example performs an additional check by loading the same operands to verify whether the loop has completed as expected.

```
lw      a5,-20(s0)
sext.w  a4,a5
li      a5,9
ble     a4,a5,.L4  // loop ends
li      a5,0 // missing loop check
```

Listing 11: Vulnerable Example (Loop Check, RISC-V, O0)

```
lw      a5,-20(s0)
sext.w  a4,a5
li      a5,9
ble     a4,a5,.L4   // loop ends
lw      a5,-20(s0)
sext.w  a4,a5
li      a5,9
bgt     a4,a5,.L5   // double check if loop ends
```

Listing 12: Resilient Example (Loop Check, RISC-V, O0)

Bypass. This vulnerable pattern arises when a condition check is not performed at the same level as the protected functionality. For instance, if a verification function is called within an `if` statement, an attacker might be able to alter the return value or the execution of a program, enabling critical code to run with a single fault. The Riscure whitepaper recommends that faults be detected at the same level (function) that executes the protected functionality, such as by storing the return values of function calls in variables before they are checked in conditions. To enhance security further, it is advised to implement double condition checks using the same logic to prevent single fault failures. A vulnerable example and a resilient example in C are provided below.

```
if (!test1()) return; // access denied
// critical code
```

Listing 13: Vulnerable Example (Bypass, C)

```
bool r1 = test1();
bool r2 = test1();
if (r1 != r2) faultDetect(); // fault detected
if (!r1 || !r2) return; // access denied
// critical code
```

Listing 14: Resilient Example (Bypass, C)

If we compare the corresponding RISC-V code examples in Listing 15 and Listing 16, we can see that the vulnerable example simply moves the return value to a register (e.g., `a5`) without storing it on the stack and subsequently execute a conditional branch (e.g., `beq`) instruction based on the value in register `a5`. On the other hand, the resilient example stores the return value in the register (register `a5`) with a `sw` instruction after the move instruction. In addition, it subsequently loads the value on stack using a `lw` instruction.

```
bl   test1
mov  a5, a0 // move return value to register a5
beq  a5, zero, .L7
```

Listing 15: Vulnerable Example (Bypass, RISC-V, O0)

```
bl     test1
mov    a5, a0  // move return value to register a5
sw     a5, -24(s0) // store return value to stack
bl     test1   // running the function again
mov    a5, a0  // move return value to register a5
sw     a5, -28(s0) // store return value to stack
lw     a4, -24(s0) // load return value 1st call
lw     a5, -28(s0) // load return value 2nd call
beq    a4, a5, .L8 // compare return values
```

Listing 16: Resilient Example (Bypass, RISC-V, O0)

Problem Formulation. We formulate the problem of detecting vulnerable RISC-V assembly code as follows. Given a sequence of lines of RISC-V assembly code $A = \{a_1, ..., a_m\}$, where m is the total number of lines in an assembly file, a detection method assigns a label l to each line, where $l \in \{\texttt{V}, \texttt{N}\}$. If label $l_i = \texttt{V}$, it indicates that the i-th line with code a_i is vulnerable. Otherwise, it suggests that this line is not vulnerable.

Evaluation Metric. A true positive (TP) indicates that the ground truth of a line is vulnerable and the detection method predicts it as vulnerable. False positive (FP) indicates that a line is not vulnerable but the detection method predicates it as vulnerable. False negative (FN) indicates that a line is vulnerable but the detection labels it as non-vulnerable.

We utilize *precision* and *recall* to assess the effectiveness of a method detecting vulnerable lines to fault injection attacks. Precision and recall are defined as follows:

$$Pecision = \frac{TP}{TP + FP}, \qquad Recall = \frac{TP}{TP + FN}$$

III. Our Proposed Detection

A. Detection Overview

In this section, we outline the details of our static analysis tool, called FaultRISC-V. The **main idea** behind our tool is illustrated in Fig. 2. Specifically, given an assembly file written in RISC-V as input, our tool performs the following steps: (1) it parses the assembly code line by line and generates tokens that include instructions, functions, attributes, global variables, and locations, which can further contain tokens such as registers, integers, strings, memory addresses, and labels, and (2) it detects lines with vulnerable patterns through token matching given each vulnerable pattern.

We implemented a parser specifically for RISC-V assembly and developed customized token matching for each type of vulnerable pattern. Each fault pattern employs a distinct

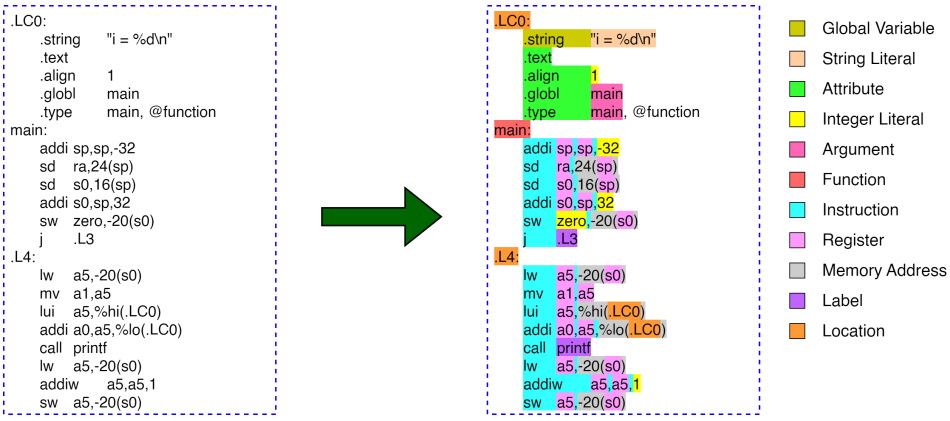

Fig. 1: Example of tokens parsed by our RISC-V parser for file `loop_simple_insecure.s` in O0

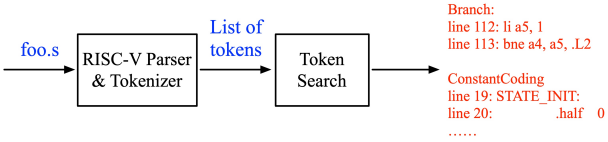

Fig. 2: Overview of FaultRISC-V

detection algorithm. The algorithm is also further tailored for each optimization level, namely O0, O1, and O2, we considered in this study. Certain vulnerable patterns require identifying a specific set of successive lines in the code, while other patterns utilize an algorithm that conducts a bi-directional search across lines that indicate or confirm vulnerability. The relevant lines and specific vulnerable patterns will be included in output of our proposed tool.

B. RISC-V Assembly Parser

We implement a customized parser for RISC-V assembly. Specifically, our parser is a linear parser, which scans assembly code line by line and tokenizes each instruction based on its broader type, including `Instruction`, `Function`, `Attribute`, `Global Variable` and `Location`. We also tokenize each element of an instruction or attribute into `Register`, `IntegerLiteral`, `StringLiteral`, `MemoryAddress`, and `Label`. A list of tokens output by our parser is linearly searched by our detection algorithm for each vulnerable pattern. An example of our parser parsing an assembly file is presented in Fig. 1.

C. Our Proposed Detection

Detection of Branch. Our method classifies a line as Branch vulnerable if (1) the line contains a load integer instruction followed by a conditional branch instruction (i.e., an `li` instruction followed by a `bx` instruction), or (2) the `li` instruction involves an integer argument with a low Hamming weight value. Alternatively, a line can be classified as vulnerable if it is a conditional branch instruction with one of the arguments being of type `IntegerLiteral` and having a low Hamming weight value. In this study, we consider a Hamming weight of 4 or less to be vulnerable by default. On the other hand, this Hamming weight of 4 is an adjustable threshold, which can be tuned for speific applications. A high-level overview of our detection method for Branch vulnerabilities is shown in Fig. 3.

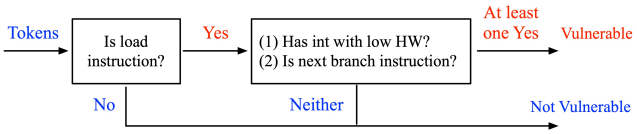

Fig. 3: Our proposed detection for Branch.

In some optimizations, such as O1 and O2, it is possible that branch vulnerabilities can exist without an initial load instruction. In other words, a branch statement exists independently with the integer value directly in the conditional operator of the branch, or a store operation that stores either a 0 or 1 in a register conditionally based on a simple boolean condition (instruction `slti` or `seqz`, for example). Our detection method incorporates these instructions. This diversity of detection pattern, in essence, is because different combinations of instructions can offer same functionalities.

Detection of Constant Coding. Our method detects a Constant Coding vulnerability if (1) a token is `GlobalVariable` or a non-overlapping `Attribute` with an `IntegerLiteral` argument; and (2) some or all of the `IntegerLiteral` values accessible by the global variable have a low Hamming weight. Each value with low Hamming weight value will be marked as a vulnerable line separately even if they belong to the same global variable. An overview of our detection for Constant Coding is described in Fig. 4.

It is worth mentioning that our proposed detection focuses on global variables only. A `GlobalVariable` is unique type of token that expands beyond a single line, i.e., it consists of a location unique name and a list of attributes that store integer literals. Our detection determines whether an Attribute comes under a GlobalVariable if it is within a contiguous block of lines that are identified by the Attribute without emphasis on what type of attribute is stored. Only

IntegerLiteral or StringLiteral attributes are recorded within the global variable for our detection purpose.

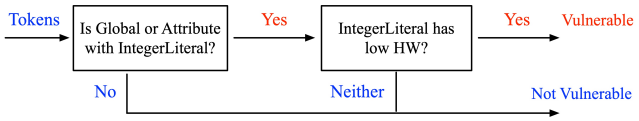

Fig. 4: Our proposed detection for Constant Coding.

Detection of Loop Check. Detecting Loop Check vulnerabilities involves two phases. The first phase identifies whether there is a `for` loop in the assembly code, and the second phase checks whether a condition exists immediately after the loop to verify its completion. In the first phase, our method detects a loop if a branch instruction points backwards, i.e., to an earlier location in the file that has already been visited during the file traversal from top to bottom. Continuity is defined as a sequential set of instructions that are executed in the order in which they appear. An unconditional jump or a return instruction would break this continuity, as they indicate that the instructions before the jump or return are either unrelated or inaccessible from those after it. This backward branching confirms the presence of a loop.

The next step in the first phase is to identify whether the loop is a `for` loop. To achieve this goal, our detection identifies an increment statement, i.e., a statement that increments the value of the iterating variable r, which is stored in one of the operands of the branch instruction. This increment statement must be found between the lines of the branch instruction and location being branched backwards to. If a `for` loop exists, our detection also identifies how many iterations v there potentially are. This can be found by inspecting the other operand of the branch instruction. It can either be loaded as an immediate value, or be a value found in the stack, or otherwise, be assumed to be loaded into the register prior to the beginning of the loop.

If the detection from the first phase is positive, the process continues to the second phase. In this phase, the register r and the value v connected with the branch instruction from phase 1 are stored. Our method then identifies if a conditional branch instruction appears within a defined gap sensitivity, which refers to the number of lines following the branch instruction where we anticipate seeing a check for the Loop Check. If the recorded register r and value v are compared again without either being modified and within the gap sensitivity, we consider it secure. Otherwise, the pattern is flagged as Loop Check vulnerable, and the branch instruction of the loop is labeled vulnerable. An overview of this process is depicted in Fig. 5.

Detection of Bypass. Our detection method for Bypass focuses on determining whether the return value from a function call (stored in register $r0$) is saved to the stack (e.g., using `sx`) or loaded from the stack (e.g., `ldrb`) after the function call (i.e., `call`) but before a conditional branch

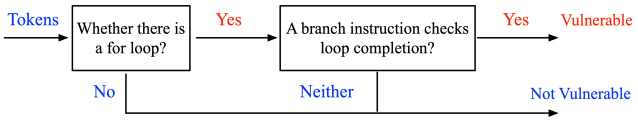

Fig. 5: Our proposed detection for Loop Check.

instruction (i.e., `bx`) that would use the return value. If this is the case, the code is considered not vulnerable. Otherwise, the `bx` instruction is flagged as vulnerable. It is important to note that these instructions do not necessarily have to appear consecutively, as unrelated instructions may be present in between. An overview of this process is presented in Fig. 6.

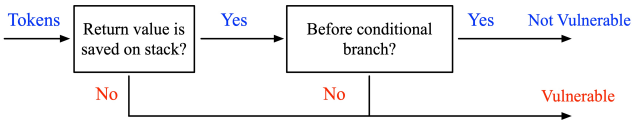

Fig. 6: Our proposed detection for Bypass.

IV. Evaluation

Dataset. We create a dataset of RISC-V assembly files across multiple optimization levels, including O0, O1, and O2. Specifically, we search relevant GitHub repositories with keywords, including embedded systems authentication and/or secure boot. We identified 50 repositories that match our keywords. Next, we compile each of the C files from these repositories running the cross-compiler toolchain `riscv64-unknown-elf-gcc` on a Linux machine. Unfortunately, many of them are not compilable for a variety of reasons, such as missing external header files. In all, we are able to compile 19 C files into RISC-V assembly from 13 repositories. Given each C file, we compile it with O0, O1, and O2 optimization, respectively. Overall, we obtain 19 assembly files at each optimization level with 17,726 lines of assembly in O0, 12,747 lines of assembly in O1, and 15,638 lines of assembly in O2.

Ground Truth. We manually label each line in these assembly files given each vulnerable patterns we consider. This involves examining instructions in each file. Each assembly file was labeled by two undergraduate researchers, and then discrepancies of labels were resolved. It is worth mentioning that a vulnerability can span multiple lines, or one line can have multiple vulnerabilities. We aggregate the number of vulnerable lines across all the assembly files we generated. The total number of the vulnerable lines we manually identified is listed in Table I. Constant Coding is the most popular vulnerable pattern from our dataset.

Results. Given the dataset and ground truth we obtain, we test the effectiveness of our tool, FaultRISC-V, by passing each of the assembly file to it and recording the overall results in terms of precision and recall. We summarize our results in Table I. We have two major observations.

First, our detection proves to be effective across all three optimization levels for all the four vulnerable patterns we

TABLE I: Precision and Recall for FaultRISC-V

	Fault Pattern	# Detected	TP	FP	FN	# Ground Truth	Precision	Recall
O0	Branch	378	357	21	12	369	94.4%	96.7%
	ConstantCoding	475	467	8	0	467	98.3%	100.0%
	LoopCheck	85	78	7	3	81	91.8%	96.3%
	Bypass	87	86	1	8	94	98.8%	91.5%
	Total	**1025**	**988**	**37**	**23**	**1011**	**96.4%**	**97.7%**
O1	Branch	342	335	7	23	358	97.9%	93.6%
	ConstantCoding	578	573	5	0	573	99.1%	100.0%
	LoopCheck	63	62	1	3	65	98.4%	95.4%
	Bypass	107	104	3	9	113	97.2%	92.0%
	Total	**1090**	**1074**	**16**	**35**	**1109**	**98.5%**	**96.8%**
O2	Branch	448	426	22	43	469	95.1%	90.8%
	ConstantCoding	578	571	7	2	573	98.8%	99.7%
	LoopCheck	201	197	4	6	203	98.0%	97.0%
	Bypass	56	56	0	0	56	100.0%	100.0%
	Total	**1283**	**1250**	**33**	**51**	**1301**	**97.4%**	**96.1%**

considered in this study. For instance, given all the assembly files in O0, our tool can achieve 97.4% precision and 98.1% recall. Moreover, the findings are relatively consistent across all the three optimizations. Our detection is robust across different optimizations. Second, our detection is efficient and it takes less than 200 milliseconds to complete the analysis on each file on average.

V. Related Work

Some recent studies [10], [11], [12], [13], [14] focus on addressing the detection of vulnerable code through dynamic analysis. For instance, Lacombe et al. [10] present a hybrid approach that combines static analysis with dynamic symbolic execution. Their methodology employs static analysis to pinpoint potential fault injection points within C source code, subsequently utilizing dynamic symbolic execution to validate the feasibility of these vulnerabilities. Lancia et al. [11] introduce a methodology that leverages symbolic execution to identify fault injection vulnerabilities directly within binaries. By simulating fault scenarios at the binary level, the approach enables the detection of potential vulnerabilities without requiring access to source code. Grycel and Schaumont [12] introduce SimpliFI, a simulation framework designed to evaluate the impact of fault injection attacks on embedded software through hardware-level simulation. Unlike traditional high-level fault models that may overlook microarchitectural nuances, SimpliFI leverages detailed hardware simulations to capture the effects of faults.

VI. Conclusion

We develop a tool automatically detecting lines vulnerable to fault injection attacks in RISC-V assembly. Our evaluation demonstrates that the method is both effective and efficient. For future research, we will focus on developing methods with automatic repair and comparisons between static analysis and dynamic analysis for fault injection detection.

Acknowledgments. This work was partially supported by National Science Foundation (CNS-2150086, DGE-2043106, and CNS-1916722).

References

[1] M. Witteman, "Secure application programming in the presence of side channel attacks," Riscure, Tech. Rep., Aug 2017. [Online]. Available: https://www.riscure.com/uploads/2017/08/Riscure_Whitepaper_Side_Channel_Patterns.pdf

[2] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM Using Fault Injection," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2016, pp. 25–35.

[3] R. Kumar, P. Jovanovic, and I. Polian, "Precise Fault-Injections using Voltage and Temperature Manipulation for Differential Cryptanalysis," in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, 2014.

[4] M. M. Alam, S. Tajik, F. Ganji, M. Tehranipoor, and D. Forte, "Ram-jam: Remote temperature and voltage fault attack on fpgas using memory collisions," in *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2019.

[5] J. Breier and X. Hou, "How Practical are Fault Injection Attacks, Really?" *IEEE Access*, 2022.

[6] R. Viera, J.-M. Dutertre, M. Dumont, and P.-A. Moëllic, "Permanent Laser Fault Injection into the Flash Memory of a Microcontroller," in *2021 19th IEEE International New Circuits and Systems Conference (NEWCAS)*, 2021.

[7] F. Khelil, M. Hamdi, S. Guilley, J. L. Danger, and N. Selmane, "Fault Analysis Attack on an FPGA AES Implementation," in *2008 New Technologies, Mobility and Security*, 2008.

[8] L. Reichling, I. Warsame, S. Reilly, A. Brownfield, N. Niu, and B. Wang, "FaultHunter: Automatically Detecting Vulnerabilities in C against Fault Injection Attacks," in *2022 IEEE/ACM International Conference on Big Data Computing, Applications and Technologies (BDCAT'22)*, 2022.

[9] P. Kharangate, G. Rached, H. Musungu, N. Niu, and B. Wang, "FaultArm: Detecting Fault Injection Vulnerabilities in Arm Assembly," in *NAECON 2024 - IEEE National Aerospace and Electronics Conference*, 2024, pp. 285–290.

[10] G. Lacombe, D. Feliot, E. Boespflug, and M.-L. Potet, "Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to Detect Fault Injection Vulnerabilities," *Journal of Cryptographic Engineering*, vol. 14, no. 1, 2023.

[11] J. Lancia, "Detecting Fault Injection Vulnerabilities in Binaries with Symbolic Execution," in *14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 2022.

[12] J. Grycel and P. Schaumont, "SimpliFI: Hardware Simulation of Embedded Software Fault Attacks," *Cryptography*, 2021.

[13] K. Murdock, M. Thompson, and D. Oswald, "FaultFinder: Lightning-fast, Multi-architectural Fault Injection Simulation," in *2024 Workshop on Attacks and Solutions in Hardware Security (ASHES'24)*, 2024.

[14] A. Adhikary, G. T. Petrucci, P. Tanguy, V. Lapôtre, and I. Buhan, "SoK: The Apprentice Guide to Automated Fault Injection Simulation for Security Evaluation," https://eprint.iacr.org/2024/1944.