FaultArm: Detecting Fault Injection Vulnerabilities in Arm Assembly

Prateek Kharangate[†], Guillermo Rached[†], Harris Musungu[§], Nan Niu[†], Boyang Wang[†]
[†]University of Cincinnati, [§]Ashland University
{*kharanpv, rachedge*}*@mail.uc.edu, harrismusungu@gmail.com, nan.niu@uc.edu, boyang.wang@uc.edu*

***Abstract*—Fault injection attacks can flip bits by changing voltage, temperature or EM radiation on a target (e.g., a microcontroller), and therefore, modify program execution on the target, such as bypassing secure boot. However, there are limited tools to automatically detect these vulnerabilities in source code at the development stage. In this paper, we develop a new tool, named FaultArm, which can automatically detection four types of vulnerable code under fault injection attacks in Arm assembly. Our approach includes (1) parsing and (2) token matching. Specifically, we design a customized parser for Arm assembly and design specific token matching rules. We create a dataset of 32 Arm assembly files with 8,493 lines across three optimization levels, including O0, O1 and O2. Our evaluation show that our tool is effective and efficient. Specifically, our tool can achieve 100% precision and 98% recall in O0, 98.6% precision and 90.9% recall in O1, and 96.5% precision and 88.2% recall in O2.**

I. Introduction

Fault injection attacks (or glitching attacks) [1], [2] can change the voltage, temperature [3], [4] or EM radiation [5], [6] on a target (e.g. a microcontroller) when it executes programs. As a result, the attack can flip bits (e.g., 0 to 1), and therefore, modify the execution of a program and forces the target to misbehave. There are several real-world examples of fault injection attacks, such as revealing AES (Advanced Encryption Standard [7]) encryption keys, bypassing secure boot on crypto wallets [1], and modifying flash memory [6].

One proactive approach of *mitigating* fault injection attacks at the *development stage* is to produce code that are resilient under fault injection attacks. For instance, two comprehensive values with a greater number of Hamming distance (e.g., 0x3CA5 and 0xC35A) are preferred to represent False and True rather than using two trivial values (e.g., 0 and 1). This is because modifying 0x3CA5 to 0xC35A requires flipping 16 bits, which is much challenging to achieve than flipping 1 bit to change 0 to 1.

A recent study developed a tool, named FaultHunter [8], which can automatically detection vulnerable code under fault injection attacks in C. This tool parses a C file into a parsing tree by leveraging ANTLR, a JAVA-based generator, and then search nodes in the parsing tree to detect vulnerable lines. The tool can detect 3 types of vulnerable patterns, including Branch, ConstantCoding, and DefaultFail, achieve 90.3% precision and 56.4% recall. *However, how to detect vulnerable lines under fault injection attacks at the assembly level remains unknown.*

Our Contributions. In this paper, we design a new tool, referred to as FaultArm, which can automatically detect vulnerable lines under fault injection attacks in Arm assembly. Our design consists of two phases, including parsing and token matching. Specifically, we first design a parser that can parse a given assembly file into a list of tokens, including instructions, registers, addresses, strings, and integers. Next, we design specific token matching rules to detect each vulnerable pattern. Our tool can detect four vulnerable patterns, including Branch, ConstantCoding, DoubleCheck, and LoopCheck [1].

We create a labeled dataset of 32 Arm assembly files (8,493 lines) and evaluate the detection performance of our tool across three optimization levels, including 00, O1, and O2. Our evaluation indicates that our tool is effective and efficient. Specifically, our tool can achieve 100% precision and 98% recall in O0, 98.6% precision and 90.9% recall in O1, and 96.5% precision and 88.2% recall in O2. *Our findings suggest that it is feasible to improve the robustness and resiliency of embedded systems and mitigate fault injection attacks early in the development stage at the assembly level.*

Reproducibility. Our source code and dataset are made publicly available and can be found on GitHub https://github.com/UCdasec/FaultArm.

II. Background

A. Vulnerable Patterns

Riscure whitepaper [1] identifies 11 vulnerable patterns in C under fault injection attacks. Riscure is a security company specialized in analyzing and preventing side-channel attacks and fault injection attacks on embedded systems. For each vulnerable pattern, the whitepaper also provides resilient coding practices that can mitigate the attacks in C. *In essence, resilient coding examples double check critical paths or data and force an attacker flipping multiple bits rather than a single bit.*

In this section, we specifically present 4 vulnerable patterns, including Branch, ConstantCoding, LoopCheck, and Bypass, that we examine in this study and provide concrete examples in both C and Arm assembly. For the detection of remaining vulnerable patterns, we will leave those as future work.

Branch. This vulnerability presents when Boolean values are used in an `if` statement. A Boolean value in an `if` statement can be modified from one state to the other (e.g., from 1 to 0) when an attacker flips one bit. On the other hand, using non-trivial numerical values to represent two different states in an `if` statement is considered more resilient (flipping multiple bits v.s. flipping 1 single bit). The vulnerable and resilient examples related to Branch in C are described 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 two instructions in Arm assembly, including a comparison instruction (e.g. `cmp`) and a branch instruction (e.g., `bne`). The associated vulnerable example of Branch in Arm compares register `r3` with a trivial value `1` in List. 3. On the other hand, the resilient example compares register `r3` with a non-trivial value `15,525` in List. 4.

```
cmp r3, #1
bne .L2 // L2 jumps to critical instructions
```

Listing 3: Vulnerable Example (Branch, Arm, compiled from List. 1 with O0

```
cmp r3, #15525  // 0x3CA5
bne .L2 // L2 jumps to critical instructions
```

Listing 4: Resillient Example (Branch, Arm, compiled from List. 2 with O0)

ConstantCoding. This vulnerable pattern covers sensitive constants carrying a limited set of values/states, e.g., {`0`, `1`, `0xFF`}, where these constant values can be easily modified from one to another within the set by modifying a single bit. On the other hand, non-trivial numerical values with greater hamming distance between two states are believed to be more resilient under fault injection attacks. This vulnerable pattern is similar as Branch. Instead of focusing on `if` statements in Branch, ConstantCoding focuses on constant variables (e.g., static variables). A vulnerable example and resilient example in C are presented below.

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

Listing 5: Vulnerable Example (ConstantCoding, C)

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

Listing 6: Resilient Example (ConstantCoding, C)

The difference between trivial values and non-trivial values still presents in Arm assembly. The corresponding vulnerable example and resilient example in Arm assembly are presented in List. 7 and List. 8.

```
STATE_INIT:
        .short  0
STATE_LOCKED:
        .short  1
```

Listing 7: Vulnerable Example (ConstantCoding, Arm, compiled from List. 5 with O0)

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

Listing 8: Resilient Example (ConstantCoding, Arm, compiled from List. 6 with O0)

LoopCheck. This vulnerability exists when an `for` loop is not followed with an `if` condition to verify if the `for` loop completed with the expected number of iterations. Without checking the completion of the loop, an attacker could flip bits such that the last few iterations are skipped and data/state is incorrect/corrupted.

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

Listing 9: Vulnerable Example (LoopCheck, C)

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

Listing 10: Resilient Example (LoopCheck, C)

In the corresponding examples in Arm assembly, the resilient example of LoopCheck leads to the repeat of a load instruction (`ldr`) on register `r3` and a comparison instruction (`cmp`) on register `r3` with the same integer value (e.g., 10). On the other hand, the vulnerable example only performs the comparison instruction on register `r3` with value 10 once.

```
ldr  r3, [r7, #20]
cmp  r3, #10
ble  .L7
ldr  r0, [r7, #16]  // missing loop check
bl   foo
```

Listing 11: Vulnerable Example (LoopCheck, Arm, compiled from List. 9 with O0)

```
ldr  r3, [r7, #20]
cmp  r3, #10
```

```
ble  .L7
ldr  r3, [r7, #20] // checking if loop is completed
cmp  r3, #10
bne  .L8
ldr  r0, [r7, #16]
bl   foo
```

Listing 12: Resilient Example (LoopCheck, Arm, compiled from List. 10 with O0)

Bypass. This vulnerable pattern is present when a condition check does not occur at the same level as protected functionality. For example, a verification function call is made within an `if` statement. This may allow an attacker to modify the return value or execution of a program and run critical code with a single fault. Instead, Riscure whitepaper suggests that faults should be detected at the same level (function) that executes protected functionality, e.g. storing the return values of function calls to variables before being checked in conditions. To make it even more secure, double condition checks with the same logic should be applied to avoid single fault failures. A vulnerable example and resilient example in C are presented 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)

The resilient example of Bypass (in Listing 16) first saves a return value of a function to register `r3` with a `mov` instruction, stores the value to stack with a `strb` instruction, and loads a return value from stack with a `ldrb` instruction. On the other hand, the vulnerable example moves the return value to register `r3` without storing it to stack or performing comparison with a `cmp` instruction based on register `r3`.

```
bl     test1
mov    r3, r0 // save return value to register r3
eor    r3, r3, #1
and    r3, r3, #255
cmp    r3, #0
```

Listing 15: Vulnerable Example (Bypass, Arm, compiled from Listing 13) with O0)

```
bl     test1
mov    r3, r0  // save return value to register r3
strb   r3, [fp, #-5] // store return value to stack
bl     test1
mov    r3, r0
strb   r3, [fp, #-6] // store return value to stack
ldrb   r2, [fp, #-5] // load return value from stack
ldrb   r3, [fp, #-6] // load return value from stack
cmp    r2, r3
beq    .L8
bl     faultDetect
......
```

Listing 16: Resilient Example (Bypass, Arm, compiled from Listing 14 with O0)

B. Problem Formulation

We formulate the problem of identifying Arm assembly code that are vulnerable under fault injection attacks as below. Specifically, given a sequence of lines of Arm assembly code $A = \{a_1, ..., a_m\}$, where m is the total number of lines in an Arm assembly file, a method identifying vulnerable code assigns a label to every line. Let l_i be the label of line a_i, where $1 \leq i \leq m$. Label l_i is either `V` (vulnerable) or `N` (not vulnerable).

C. Evaluation Metric

We leverage *precision* and *recall* to measure the effectiveness of a method identifying vulnerable lines under fault injection attacks. Specifically, a line is considered as a true positive if its ground truth is vulnerable and its predicted label from a method is vulnerable. Precision and recall are defined as below.

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

where TP is true positive, FP is false positive, and FN is false negative.

III. Proposed Automatic Detection

A. Detection Overview

In this section, we describe the details of our automatic detection tool, named FaultArm. The **main idea** of our tool can be highlighted in Fig. 1. Given an assembly file as input, our tool (1) parses the assembly code based on each line, (2) generates tokens based on registers, instructions, integers, and strings, and (3) detects lines with vulnerable patterns based on token matching across multiple lines. The associated lines and specific vulnerable patterns will be included in the output of our tool. We implement the parser, specifically for Arm assembly, and develop customized token matching for each type of vulnerable patterns.

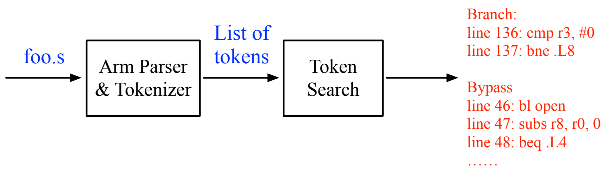
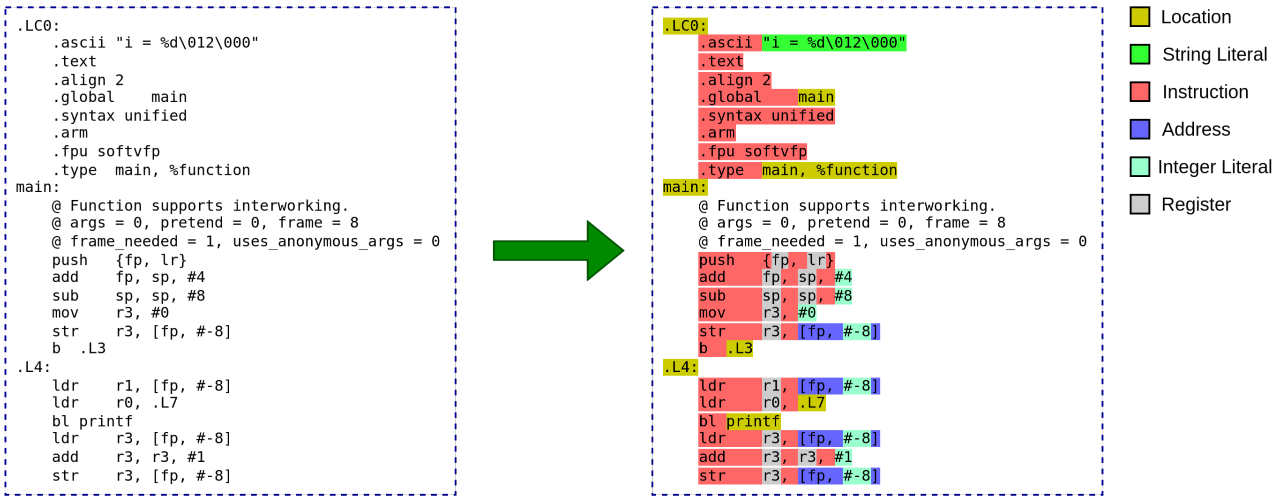


Fig. 1: Overview of FaultArm



```
.LC0:
    .ascii "i = %d\012\000"
    .text
    .align 2
    .global    main
    .syntax unified
    .arm
    .fpu softvfp
    .type  main, %function
main:
    @ Function supports interworking.
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    push    {fp, lr}
    add     fp, sp, #4
    sub     sp, sp, #8
    mov     r3, #0
    str     r3, [fp, #-8]
    b  .L3
.L4:
    ldr     r1, [fp, #-8]
    ldr     r0, .L7
    bl printf
    ldr     r3, [fp, #-8]
    add     r3, r3, #1
    str     r3, [fp, #-8]
```

Fig. 2: Example of tokens parsed by our Arm parser for file `loop_simple_secure.s` in O0

B. *Arm Assembly Parser*

We implement a customized parser for Arm assembly as we could not find an existing parser that satisfies our requirement. Specifically, our parser is a linear parser, which scans assembly code line by line and tokenize each instruction based on its broader type, including `Location`, `Instruction`, and `Address`. We also tokenize each element of an instruction into `Register`, `IntegerLiteral`, and `StringLiteral`. 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. 2.

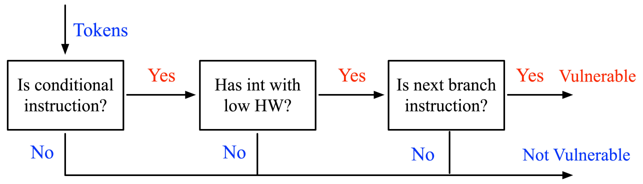


Fig. 3: Our detection method for Branch.

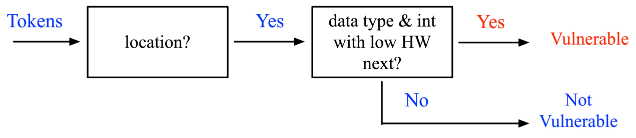


Fig. 4: Our detection method for ConstantCoding.

C. *Our Proposed Detection*

Detection of Branch. Our method decides a line is Branch vulnerable if (1) this line consists of a conditional instruction (e.g. a comparison instruction `cmp`), (2) this conditional instruction consists of an integer augment with low Hamming weight value, and (3) its next instruction is a branch instruction (`bne`, `ble`, or `bx`). In this paper, if the Hamming weight of an integer is lower than 4, we consider it as low. A high-level overview of our detection for Branch is present in Fig. 3.

It is worth to mention that, in some optimizations, such as O1 and O2, instruction `subs` or `rsbs` can be used as conditional instruction rather than `cmp`. Similarly, a branch instruction can be achieved by using an instruction `movx` instead of `bne`, `ble`, or `bx`. Our method incorporates these instructions as well when it performs the detection. This diversity of detection pattern, in essence, is because different combinations of instructions can offer same functionalities.

Detection of ConstantCoding. Our method detects a ConstantCoding vulnerability if (1) the type of a token is Location; (2) its next line consists of data type and an integer with low Hamming weight value. The line with low Hamming weight value will be marked as vulnerable. A high-level overview of our detection for ConstantCoding is present in Fig. 4. In addition to global variables, we also detect local variables with low Hamming weights.

Detection of LoopCheck. Our detection of LoopCheck consists of two phases. The first phase identifies whether there is a for loop in assembly. The second phase examines whether there is an if statement right next to the for loop in assembly. Specifically, for the first phase, our method detects there is a for loop if a combination of load instruction (e.g., `ldr`), a comparison instruction (e.g., `cmp`) and a branch instruction (e,g., `bx`) is repeated and the branch instruction returns to the same address. A high-level overview of this detection is illustrated in Fig. 5.

If the detection of the first phase is positive, our detection moves to the second phase. Specifically, the register r and value v associated with the comparison instruction in phase 1 are recorded. Our method detects there is an if statement for LooCheck if there is a later comparison on the recorded register r with the recorded value v again. If negative, then a LoopCheck vulnerable pattern is detected. The line right after the last branch instruction of the for loop is predicted as vulnerable. On the other hand, if the later comparison instruction is still on register r but with a different value than

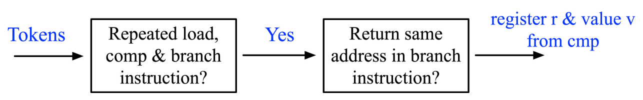

Tokens
Repeated load, comp & branch instruction?
Yes
Return same address in branch instruction?
register r & value v from cmp

Fig. 5: Our detection method for LoopCheck (Phase 1).

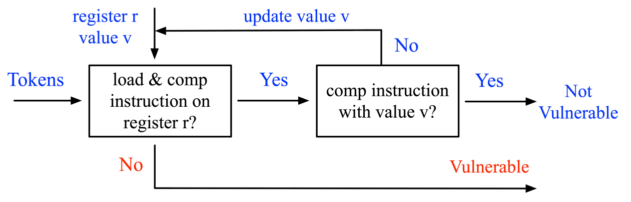


Fig. 6: Our detection method for LoopCheck (Phase 2).

v, then it indicates the for loop is still on-going and value v is updated accordingly. A high-level overview of this detection is illustrated in Fig. 6.

Detection of Bypass. Our detection on Bypass is characterized by identifying whether the value of a register is stored to the stack (e.g., `strb`) or loaded from the stack (e.g., `ldrb`) after a branch to a function (e.g., `bx` and `mov`) and prior to a comparison instruction (e.g., `cmp`). If positive, then it is considered not vulnerable. Otherwise, the line right after the `mov` instruction is considered as vulnerable.

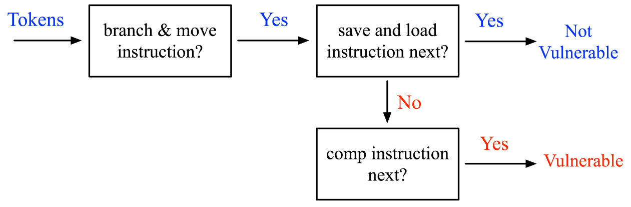


Fig. 7: Overview of our detection method for Bypass.

IV. Dataset and Evaluation

A. Our Dataset

To measure the detection performance of our design, we create a dataset of 96 Arm assembly files across three optimization levels (`O0`, `O1`, and `O2`). Specifically, we first create 24 C files manually. In addition, we leverage ChatGPT (version 3.5) to create 8 C files that are associated with security functions in embedded systems. The description of each C file generated by ChatGPT is presented below.

- ***caesarCipher.c*** – A program that performs the Caesar Cipher encryption method.
- ***calibration.c*** – A program that calibrates the position of an embedded component.
- ***data_encryption_xor.c*** – A program that encrypts a string message with an XOR operation.
- ***data_integrity_checksum.c*** – A program that generates a checksum given a string message.
- ***file_searcher.c*** – A program that returns information of a specific file or folder given its path.
- ***rate_limiting_brute_force.c*** – A program that reads a password and blocks after 3 failed attempts.
- ***rpm_plot.c*** – A program that creates data for a plot for the rpm (revolutions per minute) of a hypothetical motor.
- ***secure_data_wipe.c*** – A program that deletes a data in a program securely.
- ***simple_password_check.c*** – A program that just reads and compares the password in a single attempt.

Given these 32 C files, we compile each one of them with cross-compiler `arm-none-eabi-gcc` with multiple optimizations, including `O1`, `O2`, and `O3`. We obtain 32 Arm assembly files for each optimization.

Ground Truth. To generate the ground truth labels, we first label each line in each C file by following the vulnerable patterns defined in Riscure whitepaper. Only four vulnerable patterns, including Branch, ConstantCoding, LoopCheck, and Bypass, are considered through this labeling process. Next, we find one corresponding line in assembly based on every vulnerable line labeled in the C file. The labelling was performed by two students independently and then cross-referenced to minimize disagreements. It is worth to mention that a single line in C can map to multiple lines in assembly. Among these multiple lines, we choose the assembly line that is the most associated with each vulnerable pattern. Although each line is labeled as vulnerable or non-vulnerable, we would like to emphasize that a line is not labeled independently but based on a few lines before or/and after it. We clarify which line is labeled as vulnerable in our assembly dataset by presenting the examples below from the four vulnerable patterns.

```
cmp r3, #1 // labeled as vulnerable
bne .L2
```

Listing 17: Groud Truth Labeling (Branch, Arm, compiled from List. 1 with O0.

```
STATE_INIT:
        .short  0 // labeled as vulnerable
STATE_LOCKED:
        .short  1 // labeled as vulnerable
```

Listing 18: Groud Truth Labeling (ConstantCoding, Arm, compiled from List. 5 with O0)

```
ldr  r3, [r7, #20]
cmp  r3, #10
ble  .L7
ldr  r0, [r7, #16]  // labeled as vulnerable
bl   foo
```

Listing 19: Groud Truth Labeling (LoopCheck, Arm, compiled from List. 9 with O0)

```
bl    test1
mov   r3, r0
eor   r3, r3, #1 // labeled as vulnerable
and   r3, r3, #255
cmp   r3, #0
```

Listing 20: Ground Truth Labeling (Bypass, Arm, compiled from Listing 13) with O0)

TABLE I: Precision and Recall for FaultArm

	Fault Pattern	# Detected	TP	FP	FN	# Ground Truth	Precision	Recall
O0	Branch	53	53	0	0	53	100.0%	100.0%
	ConstantCoding	57	57	0	2	59	100.0%	96.6%
	LoopCheck	6	6	0	1	7	100.0%	85.7%
	Bypass	15	15	0	0	15	100.0%	100.0%
	Total	**131**	**131**	**0**	**2**	**133**	**100.0%**	**97.8%**
O1	Branch	37	37	0	0	37	100.0%	100.0%
	ConstantCoding	13	13	0	2	15	100.0%	86.7%
	LoopCheck	3	3	0	5	8	100.0%	37.5%
	Bypass	18	17	1	0	17	94.4%	100.0%
	Total	**71**	**70**	**1**	**7**	**77**	**98.6%**	**90.9%**
O2	Branch	48	48	0	0	48	100.0%	100.0%
	ConstantCoding	10	10	0	4	14	100.0%	71.4%
	LoopCheck	9	9	0	7	16	100.0%	56.3%
	Bypass	18	15	3	0	15	83.3%	100.0%
	Total	**85**	**82**	**3**	**11**	**93**	**96.5%**	**88.2%**

TABLE II: Summary of the number of vulnerable lines over our C and assembly files

	C	O0	O1	O2
Branch	24	53	37	48
ConstantCoding	58	59	15	14
LoopCheck	3	7	8	16
Bypass	14	15	17	15
Total No of Vulnerable Lines	99	133	77	93
Total No. of Lines	705	3,594	2,396	2,503

Note that the reason we first create C files and then produce assembly files is mainly because it is feasible to manually label vulnerable lines in C and then accurately map these labels to assembly code. It provides more accurate and reliable ground truth than labeling assembly code directly. While there are C files (related to security functions for embedded systems) available on GitHub, these files cannot always be compiled directly due to missing libraries or files. This is why we create in-house C files for evaluation. Besides, creating in-house C files also allows us to compile with different optimization levels and examine the changes of the number of vulnerable lines across different optimizations.

Overall, there are 133 lines, 77 lines, and 93 lines that are labeled as vulnerable in O0, O1, and O2 Arm assembly respectively. A summary of our labeled dataset is described in Table II.

B. Evaluation

We measure the detection performance of our methods in precision and recall and report the results in Table I. Overall, we have two main findings. First, our detection is effective across the 3 optimization levels. Second, our detection performance decreases slightly when it gets to a higher level of optimization. This is expected as a higher level of optimization leads to more optimized instructions, which increases the difficulty of our detection. Specifically, our detection achieves 100% precision and 98% recall in O0. It is also worth mentioning that our detection runs in almost real time. The analysis of each file takes within 2 seconds on average.

V. Discussion and Future Work

For our future work, we plan to extend our current methods to detect more vulnerable patterns mentioned in Riscure whitepaper [1]. In addition, we would like to further extend our dataset and investigate assembly written in other instruction sets, such as RISC-V assembly.

VI. Conclusion

We design a tool that can automatically detect vulnerable lines under fault injection attacks in Arm assembly. Our evaluation shows our method is effective and efficient. We also make our source code and dataset publicly available for the research community to reproduce and expand the findings.

Acknowledgements

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, pp. 43–48.

[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, pp. 48–55.

[5] J. Breier and X. Hou, "How practical are fault injection attacks, really?" Cryptology ePrint Archive, Paper 2022/301, 2022. [Online]. Available: https://eprint.iacr.org/2022/301

[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, pp. 1–4.

[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, pp. 1–5.

[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 Symposium for Undergraduate Research in Data Science, Systems, and Security (REU Symposium 2022)*, 2022.