FaultHunter: Automatically Detecting Vulnerabilities in C against Fault Injection Attacks

Logan Reichling[†], Ikran Warsame[§], Shane Reilly[§], Austen Brownfield[§], Nan Niu[§], Boyang Wang[§]
[†]Ohio Northern University, [§]University of Cincinnati

***Abstract*—Fault injection attacks can completely bypass typical code defenses on embedded systems and lead to severe consequences, such as leaking encryption keys and bypassing secure boot. However, programmers lack awareness of fault injection attacks and there are limited tools to automatically detect these vulnerabilities. In this paper, we conduct an empirical evaluation over 15 C files (5,005 lines of code) selected from GitHub projects designed for embedded systems. We find that 3.72% of lines (i.e., 186 lines) are vulnerable under fault injection attacks. Moreover, we develop a new tool, named FaultHunter, which can automatically detect fault injection vulnerabilities in C code. Our detection method consists of two key building blocks, including parse tree generation and token search. Our experimental results show that FaultHunter can achieve a detection performance with 90.3% recall and 56.4% precision.**

I. Introduction

Fault injection attacks [1], [2] can modify the voltage on a target (e.g. a microcontroller) when it executes programs by changing temperature [3], [4] or electromagnetic radiation [5], [6] nearby. Modified voltage can change data/instructions that a target processes, and therefore, forces the target to misbehave. For instance, changing the voltage from low to high can modify a bit from 0 to 1, which can modify the result of an `if` statement and bypass password verification on a target. Other real-world examples caused by fault injection include compromising AES (Advanced Encryption Standard [7]) encryption keys, bypassing secure boot on crypto wallets [1], and modifying flash memory [6].

One effective approach of *mitigating* fault injection attacks at the *software level* is to write secure programs that are more difficult for attackers to perform the attacks successfully. For instance, rather than checking an `if` statement with a Boolean value, e.g., `if(flag == 1)`, verifying it with a non-trivial numerical value, e.g., `if(flag == 0x3CA5)`, would be more secure. This is because an attacker has to modify multiple bits rather than a single bit to maliciously alter the value of `flag` from one state to another state (changing `0x3CA5` to `0xC35A` v.s. altering `1` to `0`), which is considered much more difficult to achieve in practice. As targets are embedded systems, the mitigations can be integrated in C code. However, compared to other vulnerabilities (e.g., buffer overflow) in C, programmers lack awareness of fault injection and there are limited tools to automatically detect these vulnerabilities.

In light of the limitations above, this paper examines the following **two research questions**:

- *RQ1: To what degree do fault injection vulnerabilities exist in C code in the real world?*
- *RQ2: How could we automatically detect fault injection vulnerabilities in C?*

The **overarching goal** of this research is to raise awareness of fault injection vulnerabilities and improve the security of embedded systems against fault injection attacks. To answer the two research questions, this paper consists of the following **contributions**:

- We conduct an empirical evaluation over real-world C code to understand how common fault injection vulnerabilities exist in practice. Specifically, we examine fault injection vulnerabilities over **15 C files** (**5,005 lines** of code in total) selected from 15 GitHub projects that are designed for embedded systems and include security components (e.g., password verification). Our analyses show that every C file tested has at least one vulnerable line, and **3.72%** (i.e., **186 lines**) of all the lines are vulnerable under fault injection attacks. We examined 3 insecure patterns, including *Branch, ConstantCoding, and DefaultFail*, defined in the Riscure whitepaper [1]. Among the 3 insecure patterns, ConstantCoding is the most common one (53.2% of insecure lines).
- We design an automatic detection tool, named **FaultHunter**, which can automatically detect fault injection vulnerabilities in C code. Specifically, given a C file, our tool first parses the C file and generates a parse tree. Next, our tool performs a token search over the nodes in the parse tree to identify lines with vulnerabilities by following insecure patterns defined in the previously mentioned Riscure whitepaper [1]. We leverage ANTLR [8], a Java-based parse generator, to parse C files and generate parse trees. Our experimental results over the 15 C files show that FaultHunter can achieve a performance of **90.3% recall** and **56.4% precision** in detecting fault injection vulnerabilities.
- The 15 C files (5,005 code lines in total without comments lines or empty lines) we analyzed also form a valuable dataset, referred to as FH dataset, which can be utilized by the community to evaluate the performance of future automatic detection tools. Each line of C code is labeled as positive (insecure) or negative (secure). If a line is labeled with insecure, the specific insecure pattern identified is also provided. To the best of our knowledge, there is no such type of dataset previously available for the research community.

Reproducibility. Our source code and dataset are made pub-

licly available and can be found on GitHub [9].

II. Background

Riscure whitepaper [1] has identified 11 insecure types of patterns/vulnerabilities in C under fault injection attacks. Riscure is a well-known security company specialized in analyzing and preventing side-channel attacks and fault injection attacks on embedded systems. For each insecure pattern, the whitepaper also provides secure coding practices that can mitigate the attacks.

Our Scope. For the scope of our research study, we focus on 3 insecure patterns, including Branch, ConstantCoding, and DefaultFail. We select these 3 patterns as they are less controversial to label. The other 8 patterns could be leveraged to generate labels as well, but the results of the labeling could vary significantly among users. We will leave the other 8 insecure patterns as one of our future work.

Next, we will briefly introduce the 3 insecure patterns covered in this study. More details of these insecure patterns can be found in [1].

Branch. This vulnerability is present when Boolean values are used in if-statement evaluation expressions. A Boolean value (e.g., 0 or 1) in an if-expression can be easily modified by an attacker as shown in List. 1. A more secure pattern is to use non-trivial numerical values in if-statement evaluation expressions as illustrated in List. 2.

```
if(flag == true){
    // Critical Code
}
```

Listing 1: Positive/Insecure Branch Example

```
if(flag == 0x5CA9){
    // Critical Code
}
```

Listing 2: Negative/Secure Branch Example

ConstantCoding. This insecure pattern covers sensitive constants that carry 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 weights between two states are recommended for secure coding against fault injection. This shares a similar concept as Branch. Rather than focusing on if statements in Branch, ConstantCoding focuses on constant variables. Both insecure and secure examples are presented below

```
static final short STATE_INIT = 0;
static final short STATE_LOCKED = 1;
```

Listing 3: Positive/Insecure ConstantCoding Example

```
static final short STATE_INIT = 0x5A3C;
static final short STATE_LOCKED = 0xC3A5;
```

Listing 4: Negative/Secure ConstantCoding Example

DefaultFail. This insecure pattern refers to the vulnerability present in branch statements when code related to a success is located inside the `else` or `default` code block. Fault injection can allow the attacker to easily access these portions of the code with fault injection, as they can simply create a state not handled, and therefore, fail into the success code.

```
if(flag == FAIL_FLAG){
    // fail code
}else{
    // success code
}
```

Listing 5: Positive/Insecure DefaultFail Example

```
if(flag == SECURE_FLAG){
    // success code
}else{
    // fail code
}
```

Listing 6: Negative/Secure DefaultFail Example

III. Dataset

To answer research question RQ1, we examine 15 real-world C files collected from GitHub and label each line in these C files by following the Riscure whitepaper. Specifically, we use the following criteria to select each GitHub project that is suitable for our study.

- The project is intended for use in embedded systems
- The project must contain C code files
- The project code has security-related components, such as cryptography and authentication
- The project is not trivial (i.e., has at least 100 lines of C code)

We select 'C' as the language filter and apply keywords, including "embedded", "iot", "embedded hardware", "smart lock", "embedded security", etc to obtain the initial pool, which has over 300 pages of search results with 10 projects per page. Then, we examine these search results manually to confirm the ones that meet our criteria. Finally, we selected 15 GitHub projects for our current study. It takes around 4 days overall to search and select these 15 projects from GitHub. Two students participate in this process. A detailed list of all the 15 projects and their GitHub links can be found in Table IV.

Once we obtain the 15 projects, we select 15 C files (one from each project). There are 5,005 lines of code in total from these 15 C files. Given each C file, we label each line as positive (insecure) or negative (secure). If it is insecure, we also specify which insecure pattern it is from among Branch, ConstantCoding, and DefaultFail. Only one insecure pattern is provided for each insecure line. Each student labels the 15 C files independently. After the initial labeling, the two students share their labeling results and discuss to reach a consensus on all the labels.

Overall, 186 lines are labeled as positive (insecure), which is 3.72% of all the 5,005 code lines we examine. Among the

TABLE I: Summary of the 3 fault injection vulnerabilities over our 15 C files

Patterns	No. of Insecure Lines
ConstantCoding	99
Branch	72
DefaultFail	15

3 insecure patterns, ConstantCoding is the most common one as shown in Table I. Every C file has at least 2 lines that are considered vulnerable. The detailed statistical information of the fault injection vulnerabilities over the 15 C files is summarized in Table II.

The 5,005 lines of code along with their labels consist of a new dataset, referred to as *FH dataset*, which can be utilized as a labeled dataset to evaluate the performance of automatic detection tools for identifying fault injection vulnerabilities in C code. To the best of our knowledge, there is no such type of dataset previously available for the research community. Establishing this FH dataset in this study aims to fill this gap.

IV. Automatic Detection

In this section, we describe the details of our automatic detection tool, named FaultHunter. The main idea of our tool can be highlighted in Fig. 1. Given a C file as input, our tool first parses the file and generates a parse tree. After obtaining the parse tree, our tool performs a token search over the nodes in the tree to identify tokens with vulnerabilities. Once the tokens are found, the corresponding lines and specific insecure patterns are provided as the output of our tool.

Fig. 1: Overview of FaultHunter

Our tool utilizes ANTLR (ANother Tool for Language Recognition) [8], a powerful Java-based parse generator, to parse a C file and generate a parse tree. An example of a parse tree generated by ANTLR is presented in Fig. 2. ANTLR offers an open source grammar file to parse C code and can create a number of automatically generated Java functions from the loaded grammar file. We override these automatically generated functions with customized listeners to detect fault injection vulnerabilities in a parse tree. For each insecure pattern, we design multiple customized listeners. Each customized listener is run when its specific grammar segment is parsed by ANTLR.

For instance, to detect Branch, we override 10 listeners related to `if` grammar. When it is inside an `if` statement, our tool examines the child nodes of "expression" and checks if the logical comparison involves a trivial value (e.g., integers with a low hamming distance, boolean values, etc). If a trivial value is found, then the corresponding line is considered as insecure. The line number and also the comment of Branch will be reported. The token search rule for Branch is described in Algo.1.

To detect ConstantCoding, 7 listeners are used. The token search rule for ConstantCoding is described in Algo.2. Two listeners are overridden to detect DefaultFail. The token search rule for DefaultFail is described in Algo.3.

Algorithm 1 (Abstract) Token Search Rule for Branch

Input: a parse tree T and a set $S = \{0, 1, Low\_HW\}$, where *Low_HW* includes values with low hamming weights (≤ 3).
while node $n \in T$ has not been visited **do**
  if n is "selectionStatement" **then**
    Find child node m == "expression";
    while node r is a child node of m **do**
      if $r \in S$ **then**
        report this line as insecure due to Branch
      end if
    end while
  end if
  Mark node n as visited
end while

Algorithm 2 (Abstract) Token Search Rule for ConstantCoding

Input: a parse tree T and a set $S = \{0, 1, Low\_HW\}$, where *Low_HW* includes values with low hamming weights (≤ 3).
while node $n \in T$ has not been visited **do**
  if n is "labeledStatement" **or** "jumpStatement" **or** "assignmentExpression" **then**
    Find child node m == "expression";
    while node r is a child node of m **do**
      if $r \in S$ **then**
        report this line as insecure due to ConstantCoding
      end if
    end while
  end if
  Mark node n as visited
end while

Algorithm 3 (Abstract) Token Search Rule for DefaultFail

Input: a parse tree T
while node $n \in T$ has not been visited **do**
  if n is "labeledStatement" **or** "selectionStatement" **then**
    if n has child nodes **then**
      report this line as insecure due to DefaultFail
    end if
  end if
  Mark node n as visited
end while

Our tool also offers a GUI to users. The GUI is developed based on JavaFX. A screenshot of the GUI is presented in Fig. 3. Specifically, a user can use the "Load C File" button to select a C file as input and click the "Run" button to obtain the output, which includes the line numbers of vulnerabilities and also the detailed comments and patterns. In addition to the output, a user can also choose to view the parse tree of the C file by checking the "Show Tree" checkbox.

V. Evaluation

In this section, we evaluate the performance of our automatic detection tool by using the FH dataset we established.

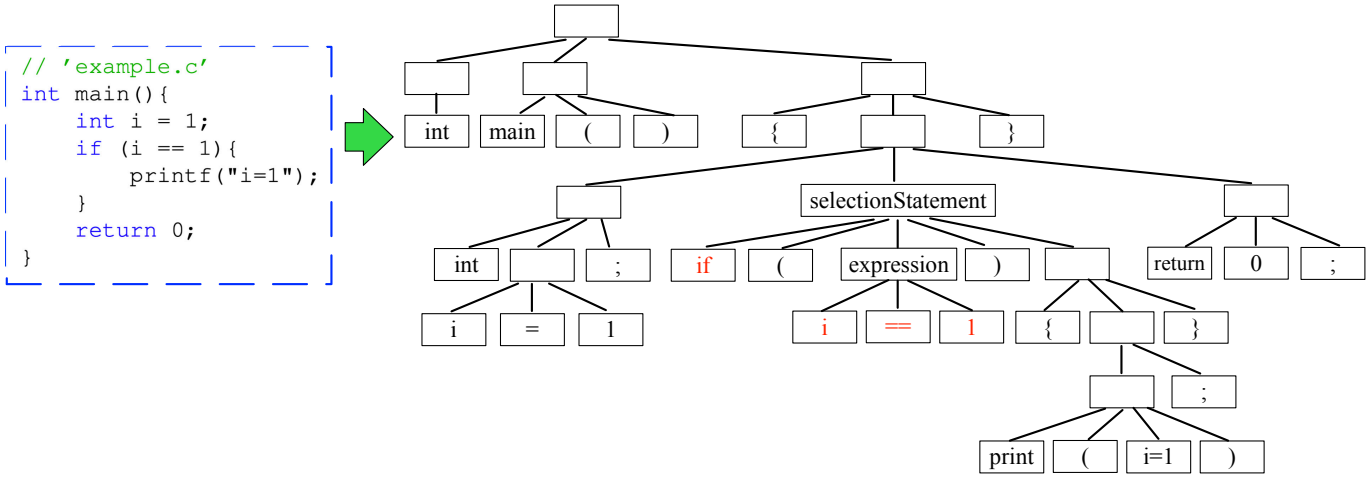

Fig. 2: The (abstract) parse tree of example.c generated by ANTLR.

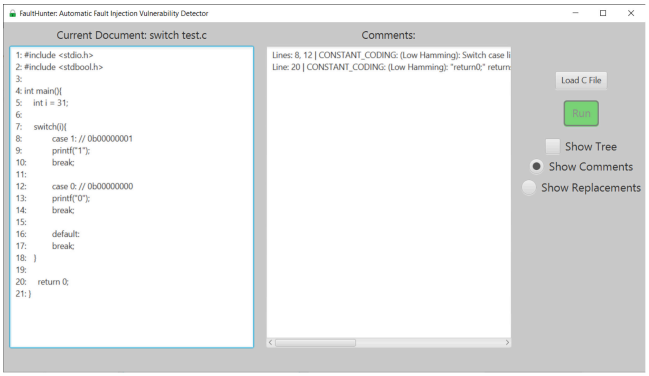

Fig. 3: The GUI of FaultHunter

Evaluation Metric. We use *precision* and *recall* to measure the performance of our tool. We define true positives, false negatives, and false positives as below.

- True Positive (TP): our tool flags a line as positive (insecure); the label of this line in the FH dataset is positive.
- False Negative (FN): our tool flags a line as negative (not included in the output); the label of this line in the FH dataset is positive.
- False Positive (FP): our tool flags a line as positive; the label of this line in the FH dataset is negative.

Precision is computed as

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

A higher precision indicates a better performance. Recall is calculated as

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

A higher recall suggests a better performance.

Between recall and precision, we decide to focus on achieving a higher recall during the design of our tool. This is because being able to detect as many insecure lines as possible is critical in this context. On the other hand, if the precision is not high (i.e., the false positive rate is high), a user can always further manually examine false positives, which although takes some additional time, is still much more effective than manually examining all the lines in a file.

Performance of FaultHunter. Overall, our tool achieves good performance for detecting fault injection vulnerabilities. Specifically, it achieves 90.3% recall and 56.4% precision. Details of the detection over each C file are summarized in Table II, while detection over each pattern are summarized in Table III. Our method takes 44.5 ms on average to examine each file across 15 C files processed sequentially.

VI. Discussion and Future Work

Covering More Insecure Patterns. In this study, we only examine three insecure patterns mentioned in the Riscure whitepaper. There are still eight more insecure patterns that can be further examined in the future. One key step would be generating a much larger labeled dataset with all the 11 insecure patterns.

Automatic Repair. We focus on automatic detection in this study. We believe that automatic repair of fault injection vulnerabilities would also be an interesting problem to research in the future.

VII. Related Work

Typical fault injection resistance involves hardware and/or software-based approaches that protect the underlying code from sudden manipulation due to fault injection. Some standard hardware-based approaches to fault injection resistance include fault detection fuses [2], data redundancy circuits [10], optical sensors, and wire meshes [11], which aim to deflect injection or disable the device if malicious faults occur during normal operation.

Software-based fault injection resistance instead involves restructuring or addition to existing firmware code, including verification of correct cryptography, replacing vulnerable constants, restructuring branch statements, and more [1]. While

TABLE II: The precision and recall of our tool over the FH dataset.

Project/File Name	**No. of Lines**	**No. of Insecure Lines**	TP	FP	FN	Precision	Recall
smart-door-lock/main.c	161	11	11	4	0	73.33%	100%
sef-project/main.c	285	21	19	14	2	57.58%	90.48%
micro-bros-smart-home/main.c	134	9	9	1	0	90.00%	100%
embedded-fingerprint/main.c	174	2	2	2	0	50.00%	100%
TrustFlex/pub_key_rotate.c	229	2	2	9	0	18.18%	100%
microFourQ/schnorrq.c	148	5	5	6	0	45.45%	100%
iotkit-embedded/iotx_http_api.c	650	23	15	7	8	68.18%	65.22%
FMT-Firmware/i2c_core.c	185	3	2	6	1	25.00%	66.67%
mbd-arduino-cashless/mdb.c	827	18	18	17	0	51.43%	100%
SmartLock_HardwareDriver/main.c	382	16	15	7	1	68.18%	93.75%
AX3_Firmware/main.c	668	26	24	37	2	39.34%	92.31%
ESP8266-Firmware/main.c	150	8	8	3	0	72.73%	100%
hardware-bitcoin-wallet/xex.c	718	23	19	15	4	55.88%	82.61%
rauc/crypt.c	139	12	12	0	0	100%	100%
tc-iot-sdk-embedded/main.c	155	7	7	2	0	77.78%	100%
Total	**5,005**	**186**	**168**	**130**	**18**	**56.38%**	**90.32%**

TABLE III: True positives, false positives, and false negatives per pattern per file.

Project/File Name	**ConstantCoding**			**Branch**			**DefaultFail**		
	TP	FP	FN	TP	FP	FN	TP	FP	FN
smart-door-lock/main.c	7	1	0	3	0	0	1	3	0
sef-project/main.c	6	8	1	12	4	1	1	2	0
micro-bros-smart-home/main.c	6	0	0	2	1	0	1	0	0
embedded-fingerprint/main.c	0	1	0	2	0	0	0	1	0
TrustFlex/pub_key_rotate.c	0	4	0	2	0	0	0	5	0
microFourQ/schnorrq.c	1	1	0	4	5	0	0	0	0
iotkit-embedded/iotx_http_api.c	12	3	4	3	0	4	0	4	0
FMT-Firmware/i2c_core.c	1	5	0	1	0	1	0	1	0
mbd-arduino-cashless/mdb.c	8	17	0	10	0	0	0	0	0
SmartLock_HardwareDriver/main.c	7	2	0	8	3	1	0	2	0
AX3_Firmware/main.c	12	11	0	10	11	2	2	15	0
ESP8266-Firmware/main.c	5	0	0	1	1	0	2	2	0
hardware-bitcoin-wallet/xex.c	7	5	4	4	2	0	8	8	0
rauc/crypt.c	11	0	0	1	0	0	0	0	0
tc-iot-sdk-embedded/main.c	7	1	0	0	0	0	0	1	0
Total:	**90**	**59**	**9**	**63**	**27**	**9**	**15**	**44**	**0**

both software and hardware-based resistance are complementary when used together, hardware-based approaches have the distinct disadvantage of requiring a redesign of the circuitry or silicon, in contrast to software-based resistance. In the age of IoT and other embedded devices, additional time and money constraints can prove untenable to embedded system manufacturers while software-based resistance can be implemented quickly and cheaply through updated firmware on vulnerable devices. However, depending on the size of the codebase, a complete review of fault injection robustness can take a considerable amount of time.

A sequence of studies have leveraged machine learning to detect traditional vulnerabilities in C code with a variety of abstractions including the use flow-sensitive ASTs (Abstract Syntax Trees) [12], and the combination of ASTs, DFGs (Data Flow Graphs), and CFGs (Control Flow Graphs) [13]. While these techniques are effective for detecting traditional vulnerabilities, they are not effective for detecting fault injection vulnerabilities, which need to examine the ease of *illogical* execution flow, i.e. skipping nodes in control flow and data flow graphs. Tradition techniques rely on the logical flow of code as it was written, which fault injection attacks disrupt. Lexing techniques were employed alongside machine learning in [14], although they decide to remove functions with duplicate CFGs from their dataset. In the case of fault injection, two code snippets may have the same CFG, yet have different vulnerabilities to fault attacks based on the constants evaluated, as in List.1 vs. List.2. Such subtle differences need to be included in a fault injection vulnerability dataset. A survey on the effectiveness of various detection techniques using machine learning was presented in [15]. Concluding their review, they emphasize the necessity of a standardized dataset to evaluate detection tools and warn that the black-box nature of ML models may call into question the trustworthiness of their results. Such concerns will be evaluated as we further develop our automatic tool.

VIII. Conclusion

In this paper, we examine to what degree fault injection vulnerabilities exist and how to perform automatic detection over these vulnerabilities. Our results show that a significant number of lines in real-world C code are vulnerable to fault injection vulnerabilities. In addition, static analysis, more specifically, parsing C files into trees and running a token search, can

TABLE IV: The links of the 15 C GitHub projects.

Project/File Name	Project Link	Last Modified
smart-door-lock/main.c	https://github.com/wywfalcon/smart-door-lock/tree/master/sdl_embedded	03.2016
sef-project/main.c	https://github.com/mario11596/sef-project/tree/master/sef-project	05.2022
micro-bros-smart-home/main.c	https://github.com/Alifathysalama/Micro_Bros-Smart-Home	05.2022
embedded-fingerprint/main.c	https://github.com/padma510/M2-Embedded_Fingerprint_Based_Security_System	12.2021
TrustFlex/pub_key_rotate.c	https://github.com/MicrochipTech/ cryptoauth_trustplatform_designsuite	03.2022
microFourQ/schnorrq.c	https://github.com/geovandro/microFourQ-AVR/tree/master/src	09.2017
iotkit-embedded/iotx_http_api.c	https://github.com/aliyun/iotkit-embedded/tree/v3.0.1/src/http	04.2021
FMT-Firmware/i2c_core.c	https://github.com/Firmament-Autopilot/FMT-Firmware/tree/master/src/hal/i2c	07.2022
mbd-arduino-cashless/mdb.c	https://github.com/LanguidSmartass/mdb-arduino-cashless	03.2019
SmartLock_HardwareDriver/main.c	https://github.com/kiraYuukiAsuna/SmartLock_HardwareDriver_Code/tree/master/USER	03.2021
AX3_Firmware/main.c	https://github.com/digitalinteraction/openmovement/tree/master/Firmware/AX3/Firmware/src	12.2017
ESP8266-Firmware/main.c	https://github.com/devicehive/esp8266-firmware/tree/develop/firmware-src/sources	03.2019
hardware-bitcoin-wallet/xex.c	https://github.com/someone42/hardware-bitcoin-wallet	04.2015
rauc/crypt.c	https://github.com/rauc/rauc/tree/master/src	08.2022
tc-iot-sdk-embedded/main.c	https://github.com/TencentCloud/tc-iot-sdk-embedded-for-esp8266/tree/master/main	05.2022

be an effective way to automatically detect fault injection vulnerabilities.

ACKNOWLEDGEMENTS

Logan Reichling was supported by National Science Foundation (CNS-2150086, NSF REU Site: Research Experiences for Undergraduates in Hardware and Embedded Systems Security and Trust, RHEST), Ikran Warsame was supported by NSF Louis Stokes Alliances for Minority Participation (LSAMP) Program, Shane Reilly and Austen Brownfield were supported by National Science Foundation (CNS-1947913, REU supplements).

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] T. Parr, "Antlr, another tool for language recognition." [Online]. Available: https://www.antlr.org/

[9] L. Reichling, I. Warsame, S. Reilly, A. Brownfield, N. Niu, and B. Wang, "Faulthunter." [Online]. Available: https://github.com/UCdasec/FaultHunter

[10] H. Mestiri, N. Benhadjyoussef, and M. Machhout, "Fault attacks resistant aes hardware implementation," in *2019 IEEE International Conference on Design and Test of Integrated Micro and Nano-Systems (DTS)*, 2019, pp. 1–6.

[11] J. Boone and S. Q. Khan, "Alternative approaches for fault injection countermeasures," NCCGroup, Tech. Rep., July 2021. [Online]. Available: https://research.nccgroup.com/2021/07/09/alternative-approaches-for-fault-injection-countermeasures-part-3-3/

[12] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2021.

[13] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, "Software vulnerability detection via deep learning over disaggregated code graph representation," *CoRR*, vol. abs/2109.03341, 2021. [Online]. Available: https://arxiv.org/abs/2109.03341

[14] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," *CoRR*, vol. abs/1807.04320, 2018. [Online]. Available: http://arxiv.org/abs/1807.04320

[15] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: A survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.