Fault Banana: Comparative Fault Injection Vulnerability in Binaries Across Instruction Set Architectures

Anonymous Author(s)

***Abstract*—Fault injection attacks compromise cyber-physical security by corrupting binary instructions on embedded systems at runtime. This paper presents a comparative study on fault injection vulnerability across three major Instruction Set Architectures (ISAs): x86, ARM, and RISC-V. Our findings indicate that the RISC-based ARM and RISC-V architectures exhibit significantly higher attack success rates (up to 6.12x) compared to the CISC-based x86 architecture.**

The contributions of this paper are two-fold. First, we present an efficient and architecture-agnostic process for detecting fault injection vulnerabilities in binaries. Second, we empirically demonstrate the increased vulnerability of RISC-based architectures compared to CISC-based architectures, providing new insights into architectural fault tolerance.

***Index Terms*—Fault Injection, Security, Embedded Software**

1. Introduction

The most infamous example of an exploitable fault injection vulnerability would be in 2018 when it was discovered that a voltage glitch could be used to load custom firmware on the Nintendo Switch from Boot ROM [1], [2]. One hacker group, Team Xecuter, sold mod chips for this exploit, leading to major lawsuits and arrests [3].

From the perspective of an attacker, fault injection is an exceptionally useful technique for bypassing secure boot, allowing a successful attacker to upload and execute arbitrary code [4]. Outside of game consoles, cyber-physical attacks on embedded systems are becoming increasingly popular [5], [6]. At the same time, the equipment required is becoming increasingly accessible and affordable [7]–[9].

Unlike traditional software attacks, Fault Injection Attacks (FIAs) do not require the presence of an existing software bug. Instead, fault injection dynamically induces software bugs by creating errors at the hardware level that propagate to the software level [10]. In contrast, Instruction Set Architectures (ISAs) are the interface between the hardware level and the software level, defining the set of instructions that a processor can execute [11]. This raises the question: Are certain architecture designs more susceptible to fault injection attacks than others?

There are two main types of ISAs: Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC). Over the years, various CISC and RISC architectures have been used for embedded systems [12].

In this paper, we hypothesize that random fault mutations have a higher attack success rate on RISC architectures than on CISC architectures due to their reduced complexity.

To measure the vulnerabilities of Instruction Set Architectures (ISAs), we build on an existing approach to detect fault injection vulnerabilities in binaries [13]–[16]. This approach simulates faults by rewriting binaries, generating mutant copies of the target binary. This works directly in binary, allowing the detection of vulnerabilities that cannot be detected in source languages or intermediate representations [17]. However, this consumes significant memory storage. Although storing copies of a firmware binary with a single mutated byte may be feasible for smaller programs, in a realistic firmware setting, there can be billions of potential fault mutations [18].

To address the limitation, we store snippets of the mutant binaries instead of full copies. To generalize the process, we utilize only common architecture toolchains. To avoid redundant executions, we catalog if mutations are unique.

We tested our hypothesis on three modern architectures: x86, ARM, and RISC-V. We use a Pin Verification Algorithm for analysis, compiling the C source code into binary for each of the target architectures. We fault every byte position to identify and measure critical instructions. In numbers, we perform 4,104 unique byte mutation experiments. For the same vulnerability, we observe an attack success rate of 16.33% for x86, 100% for ARM, and 77.77% for RISC-V.

In summary, we make the following contributions.

- We present Fault Banana, an open source tool that allows automated, efficient, and architecture-agnostic detection of fault injection vulnerabilities in binaries.
- We evaluate and compare the fault injection vulnerability of three commonly used modern architectures: x86, ARM, and RISC-V.

We made the source code of our tool as well as all of our experimental results open and publicly available [1].

1. Link redacted for blind review

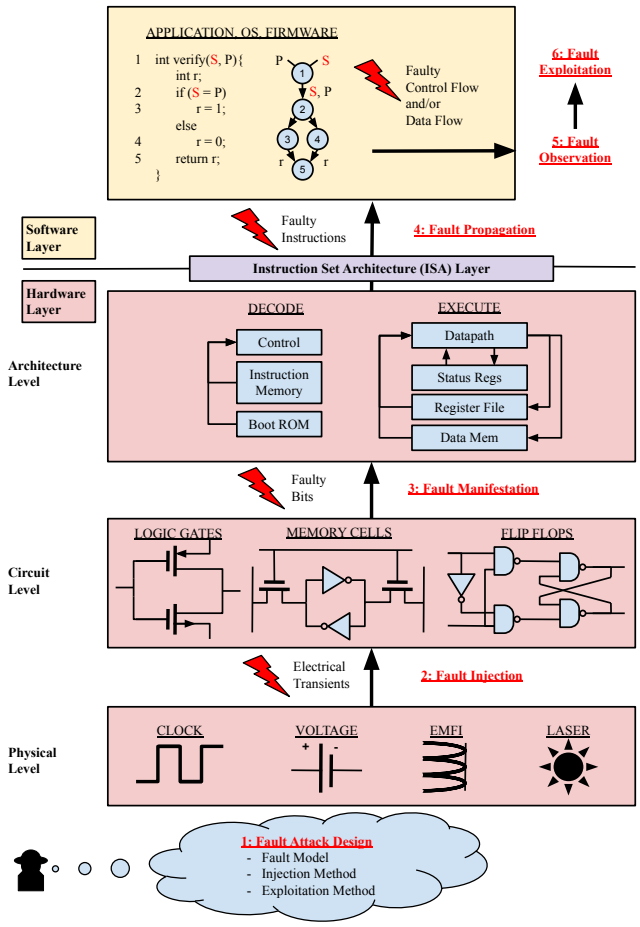

Figure 1. Anatomy of a Fault Injection Attack in [10]

2. Background

This section covers key concepts required for understanding the context of this paper. We cover the threat model of a fault injection attack, how faults are modeled, and the importance of instruction set architectures in this context.

2.1. Threat Model

The goal of a Fault Injection Attack is to bypass software security by inducing unintended hardware behavior, breaking the assumption that hardware will guarantee proper execution of software.

FIAs are achieved by changing a value, such as a byte or a bit, stored on the hardware [19]. As shown in Figure 1, faulty binary values are interpreted as faulty instructions that are interpreted as faulty control flow or data in the intended program. In this paper, we focus on attacks against user programs and assume that the hardware is untrustworthy.

2.2. Fault Models

The purpose of a fault model is to represent the nature and scope of induced mutations, describing the effect

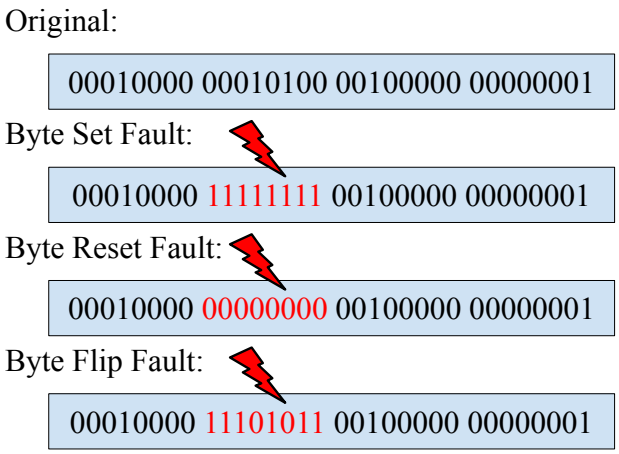

Figure 2. Byte-level Fault Models

of an injected fault. Various fault models have been proposed [20]–[24]. These fault models describe the behavior of a fault as a single bit, byte, or instruction corruption. The three basic fault models are as follows.

Set fault Changes the target bit to a '1'
Reset fault Changes the target bit to a '0'
Bit-flip fault Changes the target bit to its opposite.

In this paper, we focus on the byte-level manifestation of faults. This reduces the scope of our problem space. In addition, a byte-level model is more representative of a malicious attack [25], [26]. A byte-level model is also more consistent with electromagnetic fault injection (EMFI) where an electromagnetic pulse is directed at a local area of a processor [16]. Figure 2 shows the byte-level fault models we use in our tool.

2.3. Instruction Set Architectures

Instruction Set Architectures (ISAs) are important because they determine what strings of binary are valid instructions. All valid machine code is a subset of all binary strings. As shown in Figure 3, instructions are encoded as groups of binary. The relative position of a substring of binary determines if it represents an operation (opcode) or a specific variable (operand). Needless to say, ISAs are important because they determine what part of an instruction is corrupted when a fault is placed and whether or not it is valid. An opcode could be faulted to a value that does not correspond to any operation. An operand could be faulted to a storage location that does not exist.

For the choice between CISC vs. RISC, this is a trade-off between memory usage, execution time, and CPU complexity [12]. Complex instructions do more jobs per instruction, but require more clock cycles to compute.

This raises the question: Assuming an attacker can correctly time when a target instruction is loaded in the processor, what are the chances that a randomly placed byte fault will create a valid instruction?

3. Design

This section covers the process used in this paper for detecting fault injection vulnerabilities of binaries.

Figure 4 shows a simplified overview of the process. First, we rewrite binaries to simulate fault injection. Second, we attempt to run the binaries to observe the results of the faults. Third, we then categorize results accordingly: Correct, Incorrect, Vulnerable, Crash, and Timeout.

Correct The fault has no change on behavior
Incorrect The fault changes the output
Vulnerable The fault breaks a security property defined by an assertion statement, e.g. access with a wrong pin
Crash The fault throws an error, e.g. a segmentation fault, an illegal instruction, etc.
Timeout The fault causes an infinite loop

We perform a random "quick" scan on every byte index to identify vulnerable bytes. We use vulnerable bytes from the "quick" scan to identify the critical instructions they belong to. We then perform a "deep" scan on every bit index of critical instructions to measure the attack success rate of a randomly placed byte fault. As shown in Figure 5, the "quick" scan tests every byte offset while the "deep" scan tests every bit offset.

4. Implementation

This section covers the implementation of the process from Section 3. We utilize the standard GNU toolchain [27], [28] where possible: compiling, linking, and disassembling.

An overview of the implementation, shown in Figure 6, is as follows. We start with the source code written in the C language. We add assertion statements to check for security properties. We compile the source into executable binaries of target architectures by using corresponding GNU C Compilers (GCC) [27]. We extract assembly instructions from the binaries using the corresponding GNU Objdump Disassembler [28]. We simulate faults using custom Python scripts. We attempt to run the binaries using the QEMU Emulator [29]. We then analyze the results to generate vulnerability reports and figures using custom Python scripts.

To simplify the problem space, we focus on only the ".text" code segment section of the binary, which translates into instructions from the source code. To mimic the lack of an OS in a bare-metal embedded system, we choose to link binaries statically. However, this copies a significant amount of generic binary into the ".text" section. To analyze only the source code of interest, we compile without linking, simulate faults, and then link. To improve efficiency, we check if the offset mutation results in a unique binary, avoiding duplicate runs. To help facilitate analysis, we also compile with debug information and optimizations disabled.

The full implementation and automation of the process resulted in 1640 lines of Python code.

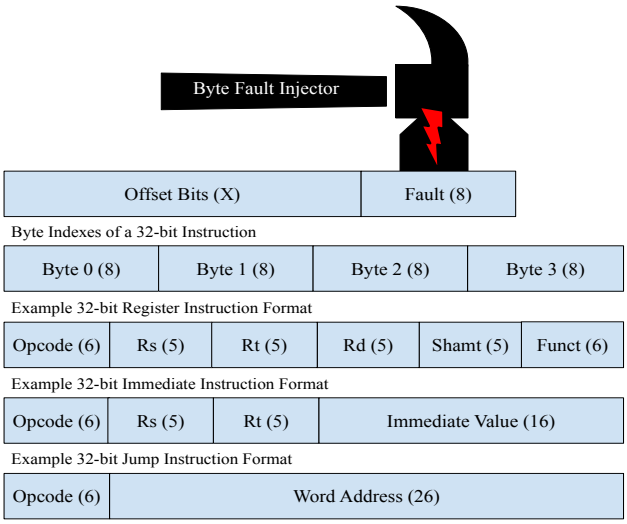

Figure 3. Fault Injector and Bit Position

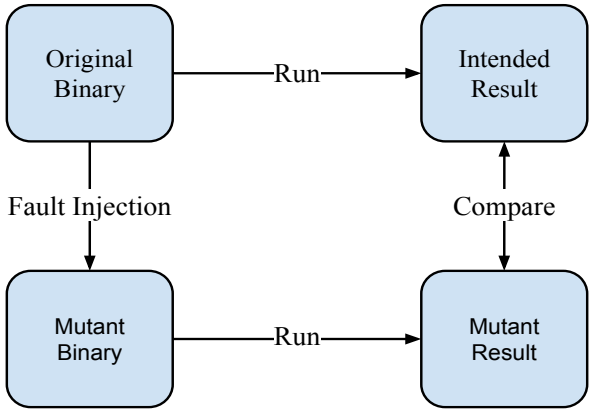

Figure 4. Process Diagram

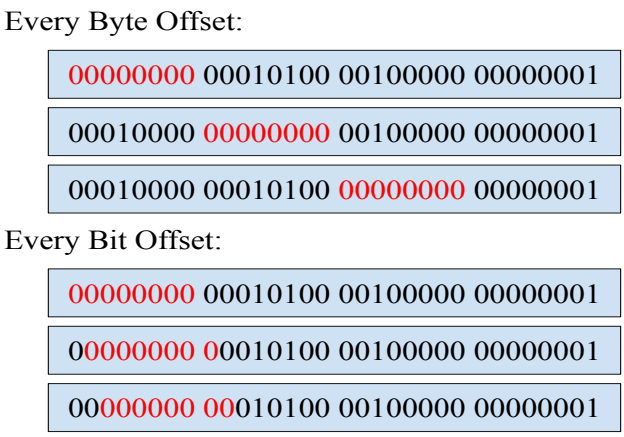

Figure 5. Fault Step Size

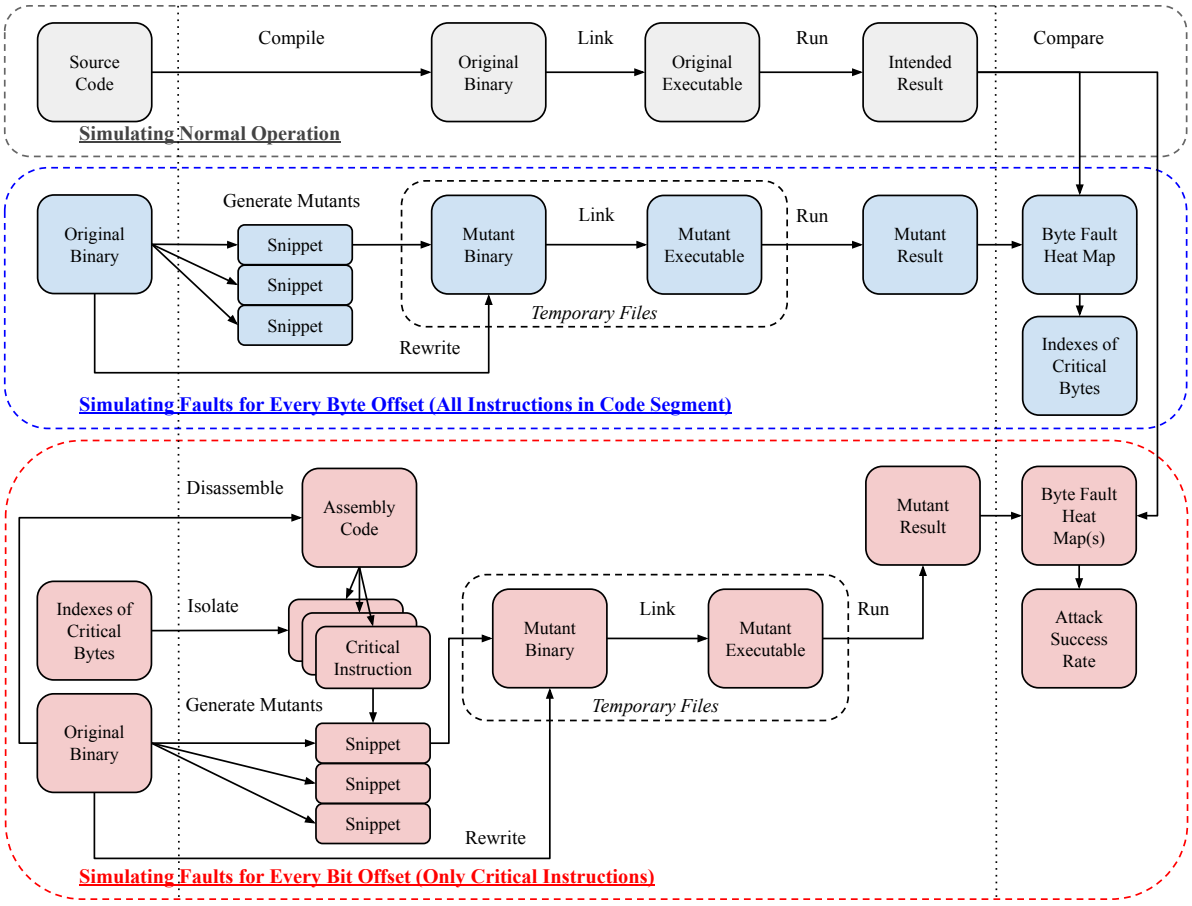

Figure 6. Implementation Diagram

5. Evaluation

This section presents the experiment results from our "quick" scan and "deep" scan, comparing the same program compiled for each target architecture.

We use an existing motivating example as a case study: a Pin Verification Algorithm [14]. The attack goal behind the motivating example is to corrupt the control flow for the opposite result of what was intended. The motivating example has zero software countermeasures. It is intentionally made to be vulnerable to fault injection.

In the Pin Verification Algorithm, a candidate pin array is checked against the true pin array, determining if access is granted. The intended vulnerability is for a fault to zero the initialization of the PINSize variable, skipping the while loop. Figure 7 shows the security property of interest. Figure 8 shows the source code of the case study.

```
assert(!(grantAccess == true && PINCandidate !=
  PINTrue));

```

Figure 7. Assert.c

```
int main() {
  int PINSize = 4;
  int PINCandidate[] = {0,0,0,0};
  int PINTrue[] = {1,2,3,4};
  bool grantAccess = false;
  bool badValue = false;
  int i = 0;
  while (i < PINSize) {
    if (PINCandidate[i] != PINTrue[i]) {
      badValue = true;
    }
    i++;
  }
  if (badValue == false) {
    grantAccess = true;
  }
  if (grantAccess) {
        printf("Access Granted");
    } else {
        printf("Access Denied");
    }
    return 0;
}

```

Figure 8. Pin-Verify.c

5.1. Quick Scan

Table 1 shows an overview of the "quick" scan, simulating faults for every byte offset, for each architecture. x86 resulted in 284 experiments for each fault model. ARM resulted in 260 experiments for each fault model. RISC-V resulted in 256 experiments for each fault model.

TABLE 1. QUICK SCAN EXPERIMENTS

Architecture	Total Bytes	.Text Size	Total Mutants
x86 32bit	3920	284	852
ARM 32bit	3792	260	780
RISC-V 32bit	6976	256	768

5.1.1. x86 Architecture. When testing x86, there were 852 mutation experiments in the "quick" scan. The observed results are shown in Table 2. Detailed results showing which faulted byte yielded which effect can be seen in the byte heat maps in Figure 9.

TABLE 2. X86 FAULT INJECTION RESULTS

	Byte Set	Byte Reset	Byte Flip
Correct	141	152	127
Incorrect	8	5	9
Vulnerable	16	13	10
Crash	118	112	134
Timeout	1	2	4

The "quick" scan identified 39 potential faults that break security. 16 vulnerable bytes were identified for set faults. 13 vulnerable bytes were identified for reset faults. 10 vulnerable bytes were identified for flip faults.

5.1.2. ARM Architecture. When testing ARM, there were 780 mutation experiments in the "quick" scan. The observed results are shown in Table 3. Detailed results showing which faulted byte yield which effect can be seen in the byte heat maps in Figure 10.

TABLE 3. ARM FAULT INJECTION RESULTS

	Byte Set	Byte Reset	Byte Flip
Correct	100	131	105
Incorrect	7	9	13
Vulnerable	20	26	20
Crash	130	91	116
Timeout	3	3	6

The "quick" scan identified 66 potential faults that break security. 20 vulnerable bytes were identified for set faults. 26 vulnerable bytes were identified for reset faults. 20 vulnerable bytes were identified for flip faults.

5.1.3. RISC-V Architecture. When testing RISC-V, there were 768 mutation experiments in the "quick" scan. The observed results are shown in Table 4. Detailed results showing which faulted byte yield which effect can be seen in the byte heat maps in Figure 11.

TABLE 4. RISC-V FAULT INJECTION RESULTS

	Byte Set	Byte Reset	Byte Flip
Correct	111	139	115
Incorrect	3	3	3
Vulnerable	14	17	20
Crash	124	89	114
Timeout	4	8	4

The "quick" scan identified 51 potential faults that break security. 14 vulnerable bytes were identified for set faults. 17 vulnerable bytes were identified for reset faults. 20 vulnerable bytes were identified for flip faults.

5.2. Deep Scan

Table 5 shows the number of critical instructions, identified by vulnerable bytes from the "quick" scan, for each architecture and fault model. Table 6 shows an overview of the "deep" scan, simulating faults for every bit offset in the critical instructions, for each architecture.

TABLE 5. CRITICAL INSTRUCTIONS

Architecture	Total Instructions	Set	Reset	Flip
x86 32bit	80	10	11	8
ARM 32bit	99	19	16	17
RISC-V 32bit	79	10	13	9

TABLE 6. DEEP SCAN EXPERIMENTS

Architecture	Set	Reset	Flip	Unique Mutants
x86 32bit	207	146	234	587
ARM 32bit	192	141	219	552
RISC-V 32bit	166	181	218	565

Fault results from testing every critical instruction were used to generate vulnerability reports, relating the vulnerable critical instructions to vulnerable source code with corresponding attack success rates. It is important to note that a fault on the assertion statement itself is a false positive. These instructions were omitted from the following tables.

5.2.1. x86 Architecture. When testing x86, there were 587 mutation experiments in the "deep" scan. Using these experiments, we calculated the attack success rates of each critical instruction (Table 7). The "deep" scan identified 14 unique critical instructions: 10 for Set, 11 for reset, and 8 for Flip. 3 of these instructions were false positives.

5.2.2. ARM Architecture. When testing ARM, there were 552 mutation experiments in the "deep" scan. Using these experiments, we calculated the attack success rates of each critical instruction (Table 8). The "deep" scan identified 23 unique critical instructions: 19 for Set, 16 for Reset, and 17 for Flip. 4 of these instructions were false positives.

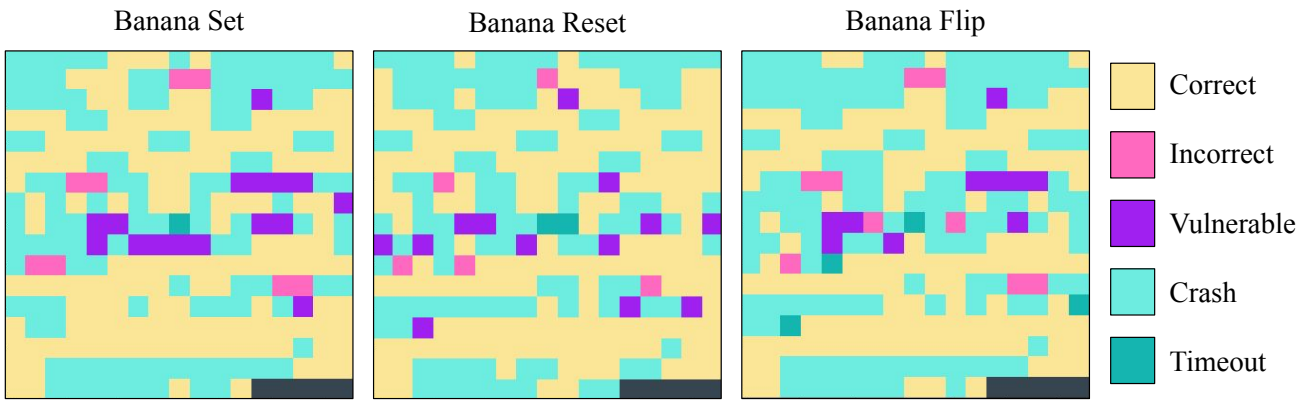

Figure 9. x86 Byte Heat Map

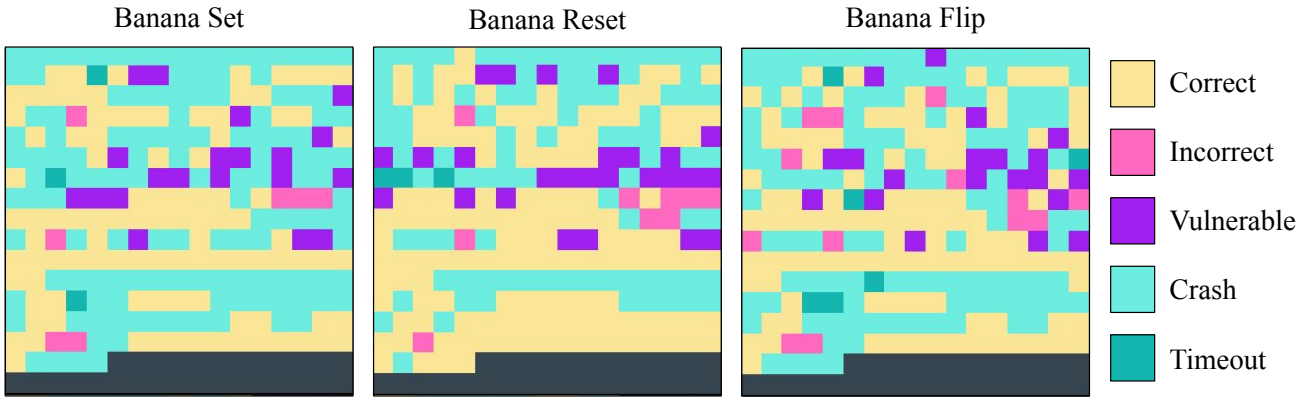

Figure 10. ARM Byte Heat Map

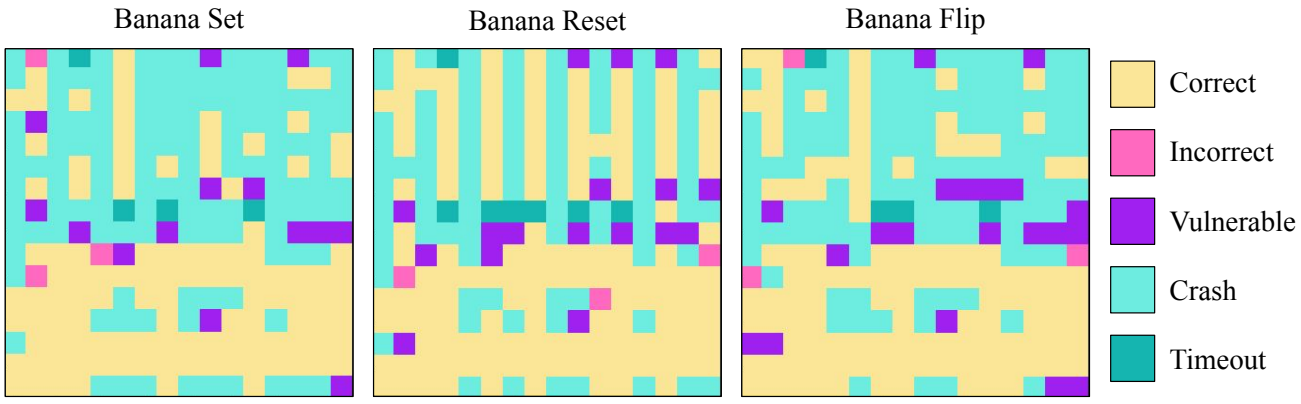

Figure 11. RISC-V Byte Heat Map

5.2.3. RISC-V Architecture. When testing RISC-V, there were 565 mutation experiments in the "deep" scan. Using these experiments, we calculated the attack success rates of each critical instruction (Table 9). The "deep" scan identified 15 unique critical instructions: 10 for Set, 13 for Reset, and 9 for Flip. 3 of these instructions were false positives.

TABLE 7. ATTACK SUCCESS RATES FOR X86

Offset	Instruction	Set	Reset	Flip
0x64	movl $0x4,-0x30(%ebp)	18.4	16.3	20.4
0xab	movl $0x0,-0x34(%ebp)	46.9	12.2	59.2
0xc2	cmp %eax,%edx	11.1		
0xc6	movb $0x1,-0x35(%ebp)	32.0	40.0	40.0
0xce	mov -0x34(%ebp),%eax	11.8		11.8
0xd1	cmp -0x30(%ebp),%eax	5.9	11.8	29.4
0xd4	jl 78 <main+0x78>		66.7	
0xd6	movzbl -0x35(%ebp), %eax	24.0	12.0	8.0
0xda	xor $0x1,%eax	58.8	17.6	52.9
0xdd	test %al,%al	33.3		
0xdf	je a9 <main+0xa9>		55.6	

TABLE 8. ATTACK SUCCESS RATES FOR ARM

Offset	Instruction	Set	Reset	Flip
0x4a	movs r3, #4	33.3	100.0	22.2
0x4c	str r3, [r7, #8]	11.1	77.8	66.7
0x4e	add.w r3, r7, #12		24.0	
0x66	stmia.w r4, {r0, r1, r2, r3}	8.0		
0x72	movs r3, #0	22.2		22.2
0x86	lsls r3, r3, #2	11.1		11.1
0x88	adds r3, #48 ; 0x30		100.0	
0x8a	add r3, r7		11.1	
0x8c	ldr.w r3, [r3, #-20]	8.0	4.0	8.0
0x92	beq.n 64 <main+0x64>	33.3		
0x94	movs r3, #1	22.2	100.0	55.6
0x96	strb r3, [r7, #3]	11.1	77.8	77.8
0xa0	ldr r3, [r7, #8]	33.3		77.8
0xa2	cmp r2, r3	11.1	100.0	33.3
0xa4	blt.n 44 <main+0x44>	33.3	100.0	55.6
0xa6	ldrb r3, [r7, #3]	11.1	77.8	77.8
0xa8	eor.w r3, r3, #1	16.0	68.0	36.0
0xae	cmp r3, #0	66.7	44.4	11.1
0xb0	beq.n 82 <main+0x82>	11.1	77.8	88.9

TABLE 9. ATTACK SUCCESS RATES FOR RISC-V

Offset	Instruction	Set	Reset	Flip
0x3c	li a5,4	22.2	77.8	77.8
0x3e	sw a5,-28(s0)	16.0	36.0	16.0
0x62	sw a2,-60(s0)	4.0		
0x9c	beq a4,a5,72 <.L3>	20.0	20.0	40.0
0xa0	li a5,1		33.3	
0xa2	sb a5,-18(s0)	12.0	36.0	12.0
0xb4	lw a5,-28(s0)	4.0		
0xb8	blt a4,a5,4c <.L4>	16.0	32.0	40.0
0xbc	lbu a5,-18(s0)		36.0	
0xc0	xori a5,a5,1	80.0	64.0	64.0
0xc4	zext.b a5,a5		28.0	8.0
0xc8	beqz a5,9c <.L5>	22.2	77.8	77.8

5.3. Technical Details

The experiments were conducted on a Windows laptop with an i7 13th Gen Intel Core Processor and Ubuntu 22.04.

Prior to memory optimizations, the motivating example experiments for x86 consumed 1.05GB of memory. With binary snippet optimizations, the experiments consumed 449MB of memory, resulting in a 57% reduction.

6. Discussion

This section analyzes "quick" scan and "deep" scan results from Section 5.

6.1. Quick Analysis

The two most common results of a random byte fault were "Correct", the fault was benign and had no impact on the behavior of the program, and "Crash", the fault caused the program to throw an error. "Vulnerable" byte faults were rare, which was expected. "Incorrect" included corrupted console outputs that were not "Vulnerable". These were more rare. During emulation, experiments both timed out and threw an error when the laptop overheated. These results were categorized as "Crash". "Timeout" was the rarest, reserved for results that timed out without error, indicating an infinite loop.

The order of Instruction Set Architectures from the highest to the lowest random vulnerable byte fault total is as follows: ARM, RISC-V, and x86.

6.2. Deep Analysis

For critical instructions, at least one byte fault position is categorized as "Vulnerable". The attack success rate of a critical instruction was calculated by counting the bit offset byte faults with a "Vulnerable" result and dividing it by the total possible bit offset positions. This represents the chances that a randomly placed byte fault will create a "Vulnerable" mutation, assuming that an attacker can correctly time when the target instruction is loaded in the processor.

The order of Instruction Set Architectures from the highest to the lowest vulnerable critical instruction total is as follows: ARM, RISC-V, and x86.

The order of Instruction Set Architectures from the highest to the lowest attack success rates is as follows: ARM, RISC-V, and x86. Although RISC-V and ARM are similar, only ARM had attack success rates of 100%.

6.2.1. Vulnerability. As motivated in Section 5, the intended vulnerability of the case study is for a fault to zero the initialization of the PINSize variable. The generated vulnerability report for this line of code in x86, ARM, and RISC-V is shown in Figure 12, Figure 13, and Figure 14 respectively. For the RISC architectures, ARM and RISC-V, the initialization was separated into two assembly instructions for loading and storing. x86, being a CISC architecture, used only one complex assembly instruction instead.

For the same vulnerable instruction, moving the value 4 to a register, the deep scan showed significant differences in the attack success rate for the architectures (Figure 15). x86 was 16.33%. ARM was 100%. RISC-V was 77.77%. In comparison to x86, ARM was 6.12 times more vulnerable while RISC-V was 4.76 times.

For x86, only a small range of bit offsets resulted in "Vulnerable". This suggests that the vulnerability's root cause was the bits of the operand holding a 4 being zeroed.

Source Code					
int PINSize = 4;					
Offset	Bytecode	Assembly Instruction	Set Fault	Reset Fault	Flip Fault
0x64	c7 45 d0 04 00 00 00	movl $0x4,-0x30(%ebp)	18.37	16.33	20.41

Figure 12. x86 Report Segment

Source Code					
int PINSize = 4;					
Offset	Bytecode	Assembly Instruction	Set Fault	Reset Fault	Flip Fault
0x4a	2304	movs r3, #4	33.33	100.00	22.22
0x4c	60bb	str r3, [r7, #8]	11.11	77.78	66.67

Figure 13. ARM Report Segment

Source Code					
int PINSize = 4;					
Offset	Bytecode	Assembly Instruction	Set Fault	Reset Fault	Flip Fault
0x3c	4791	li a5,4	22.22	77.78	77.78
0x3e	fef42223	sw a5,-28(s0)	16.00	36.00	16.00

Figure 14. RISC-V Report Segment

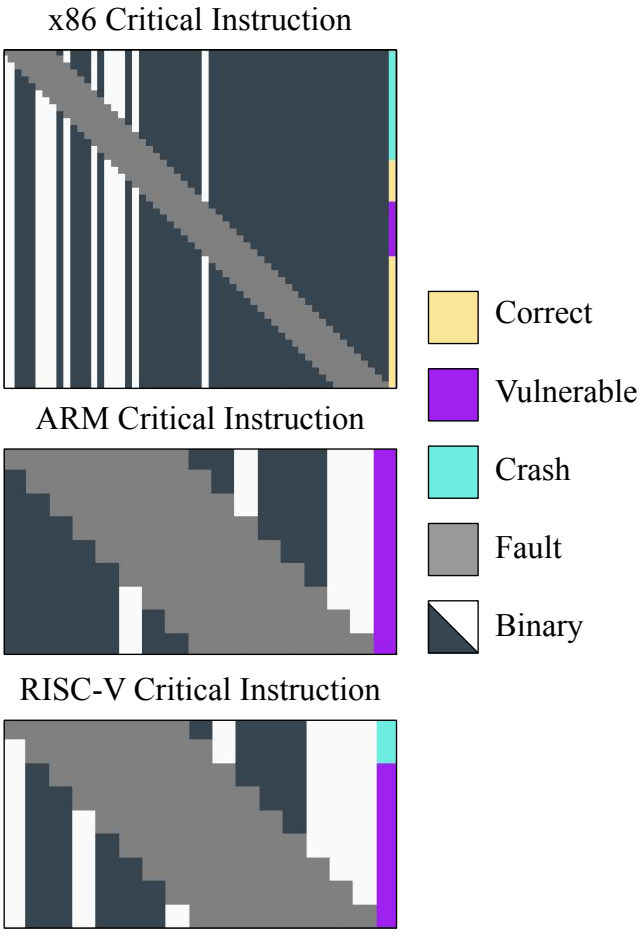

Figure 15. Comparing Deep Scans

For ARM, every bit offset resulted in "Vulnerable". This suggests that the vulnerability's root cause was the mutant instruction having an equivalency collision with a benign instruction, allowing the register to keep a previous value. With the x86 instruction being 56 bits and the ARM instruction being 16 bits, it is unsurprising that ARM is more likely to have a collision with a valid instruction. RISC-V is also 16 bits, but it resulted in "Crash" more in this instance.

7. Related Work

This section discusses how the process used in this paper compares to other existing approaches for detecting fault injection vulnerabilities in binaries.

Initially, our goal was to replicate the approach proposed by Given-Wilson et al. [13]–[16]. This involves generating mutant binaries and subjecting them to model and property checks. However, we found this approach inefficient, requiring a complicated system to check an assertion statement when a standard assertion statement could suffice. This approach also required raising the binary to an intermediate language to perform model checking, which lacks support for many architectures. Therefore, we decided to build on and simplify this approach, making it more efficient and architecture-agnostic.

An alternative approach involves making internal modifications to a system emulator [18], [30]. Space RadSim [18] uses binary-agnostic emulation in QEMU. This approach abstracts away the binary, relying on intermediate translations. FaultFinder [30] is another QEMU-based tool. However, FaultFinder optimizes the simulation process by identifying equivalent states. This is similar to our process where we checked if a mutation results in a unique binary to avoid duplicate executions.

8. Conclusion

In conclusion, we empirically demonstrate that RISC-based architectures, such as ARM and RISC-V, are more vulnerable than CISC-based architectures, such as x86, to Fault Injection Attacks. Fault Injection is an increasing threat to embedded systems, which often use RISC architectures. To help mitigate this threat, we present Fault Banana, an open source tool which allows automated, efficient, and architecture-agnostic detection of fault injection vulnerabilities in binaries.

8.1. Future Work

Although this study contributes new insights on architectural fault tolerance, several areas warrant further exploration. One avenue for future research is to conduct a larger survey of Instruction Set Architectures. Another avenue is to expand the case study used for comparison to be more representative of a real-world binaries, such as secure bootloaders used in modern embedded systems.

References

[1] K. Temkin. (2018, Jun.) Vulnerability disclosure: Fusée gelée. [Online]. Available: https://switch.hacks.guide/files/extras/fusee_gelee_nvidia.pdf

[2] A. Galauner. (2018, Jun.) Glitching the switch. [Online]. Available: https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch

[3] D. Giannakis. (2021, Nov.) Team xecuter - the story of the infamous nintendo switch modding group — mvg. [Online]. Available: https://youtu.be/5sNIE5anpik?si=fQTni9B7uOVvMFVm

[4] N. Timmers and A. Spruyt. (2016, Nov.) Bypassing secure boot using fault injection. [Online]. Available: https://raelize.com/upload/research/2016/2016_BlackHat-EU_Bypassing-Secure-Boot-Using-Fault-Injection_NT-AS.pdf

[5] T. Roth. (2019, Dec.) Trustzone-m(eh): Breaking armv8-m's security. [Online]. Available: https://media.ccc.de/v/36c3-10859-trustzone-m_eh_breaking_armv8-m_s_security

[6] I. Zhuravlev and J. Boone. (2020, Oct.) There's a hole in your soc: Glitching the mediatek bootrom. [Online]. Available: https://www.nccgroup.com/us/research-blog/there-s-a-hole-in-your-soc-glitching-the-mediatek-bootrom/

[7] Chipwhisperer. [Online]. Available: https://github.com/newaetech/chipwhisperer

[8] Chipwhisperer. [Online]. Available: https://github.com/newaetech/chipwhisperer

[9] Voltpillager. [Online]. Available: https://zt-chen.github.io/voltpillager/

[10] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 111–130, 2018.

[11] What is an instruction set? [Online]. Available: https://www.lenovo.com/us/en/glossary/what-is-instruction-set/

[12] C. Walls. (2016, Apr.) Cisc and risc. [Online]. Available: https://blogs.sw.siemens.com/embedded-software/2016/04/18/cisc-and-risc/

[13] T. Given-Wilson, N. Jafri, J.-L. Lanet, and A. Legay, "An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present," in *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 293–300.

[14] T. Given-Wilson *et al.*, "An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present – extended version," 2017.

[15] T. Given-Wilson, A. Heuser, N. Jafri, and A. Legay, "An automated and scalable formal process for detecting fault injection vulnerabilities in binaries," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 23, p. e4794, 2019.

[16] T. Given-Wilson, N. Jafri, and A. Legay, "Combined software and hardware fault injection vulnerability detection," *Innovations in Systems and Software Engineering*, vol. 16, no. 2, pp. 101–120, 2020.

[17] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys, "Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks," in *Foundations and Practice of Security*, F. Cuppens, J. Garcia-Alfaro, N. Zincir Heywood, and P. W. L. Fong, Eds. Springer International Publishing, 2015, pp. 92–111.

[18] J. Willbold, T. Cloosters, S. Wörner, F. Buchmann, M. Schloegel, L. Davi, and T. Holz, "Space radsim: Binary-agnostic fault injection to evaluate cosmic radiation impact on exploit mitigation techniques in space," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 1047–1063.

[19] I. Verbauwhede, D. Karaklajic, and J.-M. Schmidt, "The fault attack jungle - a classification model to guide you," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2011, pp. 3–8.

[20] J. Balasch, B. Gierlichs, and I. Verbauwhede, "An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus," in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2011, pp. 105–114.

[21] R. Nabhan, J.-M. Dutertre, J.-B. Rigaud, J.-L. Danger, and L. Sauvage, "A tale of two models: discussing the timing and sampling em fault injection models," in *2023 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. IEEE, 2023, pp. 1–12.

[22] O. Hériveaux, "Triple exploit chain with laser fault injection on a secure element," in *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2022, pp. 9–17.

[23] T. Trouchkine, G. Bouffard, and J. Clédière, "Em fault model characterization on socs: from different architectures to the same fault model," in *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. IEEE, 2021, pp. 31–38.

[24] V. Khuat, J.-L. Danger, and J.-M. Dutertre, "Laser fault injection in a 32-bit microcontroller: from the flash interface to the execution pipeline," in *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. IEEE, 2021, pp. 74–85.

[25] C. Roscian, J.-M. Dutertre, and A. Tria, "Frontside laser fault injection on cryptosystems - application to the aes' last round -," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 119–124.

[26] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*. Springer Berlin Heidelberg, 2011, pp. 224–233.

[27] Gcc, the gnu compiler collection. [Online]. Available: https://gcc.gnu.org/

[28] Gnu binutils. [Online]. Available: https://www.gnu.org/software/binutils/

[29] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USENIX Association, 2005, p. 41.

[30] K. Murdock, M. Thompson, and D. Oswald, "Faultfinder: Lightning-fast, multi-architectural fault injection simulation," in *Proceedings of the 2024 Workshop on Attacks and Solutions in Hardware Security*. Association for Computing Machinery, 2024, p. 78–88.