Breaking Authenticated Encryption: A Fault Injection Attack on AES-GCM

Final Project Report for EECE7352 - Computer Architecture

Jack Bonfiglio
Northeastern University
bonfiglio.j@northeastern.edu

December 2025

Abstract

AES-GCM (Galois/Counter Mode) is one of the most widely deployed authenticated encryption schemes, protecting everything from web traffic in TLS 1.3 to virtual private networks in IPsec. While cryptographically secure against mathematical attacks, its real-world security depends on correct physical implementation. This project demonstrates how physical fault injection attacks can completely bypass AES-GCM's authentication guarantees, allowing an attacker to forge modified messages that the system incorrectly accepts as authentic.

Using the FaultFinder simulation framework on ARM architecture, we conducted a comprehensive fault injection campaign targeting the GHASH authentication component of AES-GCM. Our experiments injected over 58,000 single-bit faults into registers during Galois field multiplication operations. The results reveal a concerning vulnerability: approximately 7.4% of injected faults produce exploitable authentication corruption, and nearly 1 in 4 faults (24%) enable successful message forgery attacks.

We demonstrate this vulnerability through a realistic banking transaction forgery scenario, where an attacker intercepts an encrypted message authorizing a $100 transfer and modifies it to $999. Without fault injection, the authentication system correctly detects and rejects the tampered message. However, with strategic fault injection during tag verification, the forged transaction passes authentication 24% of the time, resulting in 14,161 successful forgeries out of 58,656 attempts.

This work demonstrates that cryptographic implementations remain vulnerable at the hardware level, even when the underlying algorithms are sound. Our findings emphasize the critical need for fault-resistant hardware designs and countermeasures in systems processing sensitive data, particularly in physically accessible environments where attackers can exploit implementation-level vulnerabilities.

Contents

1 Introduction

AES-GCM has become the standard for authenticated encryption, protecting everything from TLS 1.3 web traffic to cloud storage systems. Major providers like AWS process billions of AES-GCM operations daily [1]. While the algorithm is mathematically secure, its physical implementation introduces vulnerabilities that theory cannot address. Fault injection attacks exploit this gap by inducing transient errors during computation—voltage glitches, clock manipulation, or laser pulses that cause processors to misexecute instructions [2].

This project demonstrates how fault injection can break AES-GCM's authentication. Using the FaultFinder simulation framework [2], we injected over 58,000 single-bit faults into registers during GHASH operations. Our results show that 7.4% of faults corrupt authentication in exploitable ways, and 24% enable successful message forgery. We demonstrate this through a banking transaction attack where an intercepted $100 transfer is modified to $999. With strategic fault injection during tag verification, the forged transaction passes authentication 24% of the time—14,161 successful forgeries out of 58,656 attempts.

This report presents background on AES-GCM and fault injection (Section 2), our experimental methodology (Section 3), fault injection results (Section 4), the practical attack demonstration (Section 5), and security implications (Section 6). All source code, configuration files, and experimental data are available at: `https://github.com/Jbonfigs/AES-GCM-Fault-Injection`

2 Background

2.1 AES-GCM Authenticated Encryption

AES-GCM is an authenticated encryption mode that combines the confidentiality of AES counter mode with the authentication guarantees of the GHASH function [3]. The mode takes as input a plaintext message, an initialization vector (IV), and additional authenticated data (AAD), producing both ciphertext and an authentication tag. The authentication tag provides cryptographic assurance that the message hasn't been tampered with during transmission or storage.

The encryption portion uses AES in counter mode, where successive counter values are encrypted and XORed with plaintext blocks to produce ciphertext. This provides confidentiality but no authentication. The authentication component, GHASH, operates on the ciphertext and AAD to produce a message authentication code. GHASH is essentially a universal hash function that performs polynomial evaluation in the Galois field GF(2^{128}) [3].

The GHASH computation is where our attack focuses. For each block of input data, GHASH performs an XOR with the previous state, followed by multiplication by a hash key H in GF(2^{128}). This can be expressed as:

$$X_i = (X_{i-1} \oplus C_i) \cdot H \tag{1}$$

where X_i is the intermediate authentication state, C_i is the current ciphertext block, and H is the hash key derived from encrypting a zero block with the AES key. After processing all blocks, a final encryption step produces the authentication tag.

The iterative nature of GHASH creates a vulnerability window. Each multiplication operation involves 128 iterations of conditional XOR operations based on individual bit values. During this computation, register values are repeatedly read, modified, and written back. A fault injected during any of these operations can propagate through subsequent iterations, corrupting the final authentication tag in unpredictable ways. Unlike some cryptographic primitives where a single bit

flip might have minimal effect, faults in GHASH can cascade through the polynomial multiplication, potentially producing tags that an attacker can exploit.

2.2 Fault Injection Attacks

Fault injection attacks exploit the gap between cryptographic theory and physical implementation. While AES-GCM is mathematically secure, the hardware executing it operates in the physical world where transient errors can be deliberately induced. Physical techniques include voltage glitching (briefly manipulating power supply), clock glitching (creating timing violations), laser injection, and electromagnetic pulses—all capable of causing processors to misexecute instructions or skip security checks [2].

These attacks are particularly dangerous because they leave no permanent trace. A single transient fault during a critical operation can cause an instruction to execute incorrectly, a comparison to produce the wrong result, or a conditional branch to be incorrectly taken. For authenticated encryption like AES-GCM, attackers don't need to leak the secret key—they can simply corrupt the authentication mechanism to forge messages.

2.3 FaultFinder Simulation Framework

FaultFinder is a high-performance fault injection simulation tool built on the Unicorn Engine CPU emulator that provides architecture-independent firmware security evaluation [2]. The tool addresses a fundamental challenge: the gap between identifying vulnerabilities through manual analysis and validating them through expensive physical experiments. Traditional fault injection requires specialized hardware, careful sample preparation, and time-consuming parameter sweeps that may still miss exploitable faults.

FaultFinder enables systematic fault space exploration through simulation. Users specify fault models (bit flips in registers, instruction skips, memory modifications), and the tool exhaustively tests each fault while recording outcomes. The architecture consists of a Controller managing the campaign, parallel Worker tasks running Unicorn emulator instances with specific faults, and a Logger collecting results [2].

Two key optimizations enable large-scale campaigns. Checkpoints save intermediate execution states, allowing late-execution faults to restore from nearby checkpoints rather than re-executing from the beginning. Equivalence detection identifies when different faults produce identical states, focusing analysis on unique outcomes [2].

We chose FaultFinder over alternatives for several reasons. ARCHIE focuses on system-level faults for safety applications and bootloader security rather than deep cryptographic analysis [4]. FiSim, designed for bootloader training, provides a GUI but has slower performance for large campaigns and targets instruction-level faults primarily. FaultFinder's performance, cryptographic security focus, and detailed register logging made it ideal for analyzing authentication mechanisms like GHASH where understanding fault propagation through Galois field arithmetic is critical.

3 Experimental Methodology

3.1 Implementation

The core implementation centers on two functions that embody the GHASH authentication computation. The `gf128_mul()` function performs binary Galois field multiplication over GF(2^{128}),

implementing the polynomial reduction specified in the GCM standard [3]. This operation translates to a 128-iteration loop where each iteration involves conditional XOR operations based on individual bit values, followed by shift operations and conditional polynomial reduction. At the instruction level, this manifests as repeated sequences of load, test-and-branch, XOR, and shift instructions—each presenting a potential target for fault injection.

The `ghash()` function orchestrates the authentication tag computation by XORing input data blocks with an accumulator, then invoking `gf128_mul()` to perform the field multiplication. This iterative structure creates dependencies between successive operations where faults can propagate through the computation pipeline. From an architectural perspective, the critical registers (R0-R3 in our ARM implementation) hold intermediate multiplication results that, if corrupted, cascade through subsequent iterations to produce exploitable authentication failures.

Our compilation strategy uses the GNU ARM cross-compiler with optimization disabled (`-O0`) to preserve a direct mapping between C source code and generated assembly instructions. The resulting binary executes approximately 10,000 instructions during a complete GHASH computation, with the core Galois field multiplication accounting for roughly 5,000 instructions in the critical path. We configured FaultFinder to inject faults specifically during this multiplication window where register values undergo repeated modification.

3.2 Fault Injection Configuration

Table 1: Fault Injection Campaign Configuration

Parameter	**Configuration**
Target Architecture	ARM 32-bit (ARMv7)
Emulation Engine	Unicorn Engine via FaultFinder
Fault Model	Single-bit XOR
Target Registers	R0-R3 (computation registers)
Fault Window	GHASH computation (5000 instructions)
Bit Positions	8 positions per register
Total Fault Attempts	58,656
Checkpoints	5 (for performance)

We configured FaultFinder to systematically inject single-bit faults into the ARM registers during GHASH execution. The emulation runs on Unicorn Engine [5], a lightweight CPU emulator that provides precise control over processor state without the overhead of full system virtualization. Our fault model targets registers R0 through R3, which hold the critical intermediate values during Galois field multiplication. For each register, we inject faults at 8 different bit positions (bits 0, 1, 2, 4, 8, 16, 32, 64), testing how bit flips at various magnitudes affect the final authentication tag.

The fault injection window spans approximately 5,000 instructions covering the core `gf128_mul()` function. We use FaultFinder's checkpoint feature with 5 strategically placed snapshots to avoid re-executing the entire program for each fault. Each of the 58,656 fault attempts injects exactly one bit flip at a specific instruction, records whether the program crashes or completes, and captures the final authentication tag value in register R0.

3.3 Golden Run Baseline

Before injecting any faults, we established a baseline by executing the GHASH implementation without modification. This "golden run" completed successfully after 81,908 instructions, producing an authentication tag of `0x6a94589f` in register R0. The program terminates cleanly when it attempts to fetch from unmapped memory at address `0x0`—an expected behavior for our bare-metal implementation that signals completion by returning to a null pointer.

This golden tag serves as our reference point for evaluating fault injection outcomes. Any fault that produces a different tag value in R0 represents a corruption of the authentication mechanism. We categorize each fault attempt by comparing its output to this baseline: matching tags indicate the fault had no effect, while differing tags suggest exploitable authentication failures.

3.4 Metrics

We classify each fault injection attempt into three categories based on its outcome. Faults that produce the golden tag (`0x6a94589f`) had **no effect**—the bit flip occurred in a location or at a time that didn't impact the final computation. Faults that cause the program to crash or fail to complete are classified as **program crashes**, typically resulting from corrupted memory addresses or invalid operations that terminate execution early.

The most interesting category is **exploitable faults**—those that allow the program to complete successfully but produce a corrupted authentication tag different from the golden value. These represent authentication bypasses where an attacker's modified message would incorrectly pass verification. From a security perspective, exploitable faults are the critical metric, as they directly enable message forgery attacks.

4 Fault Injection Results

4.1 Overall Campaign Results

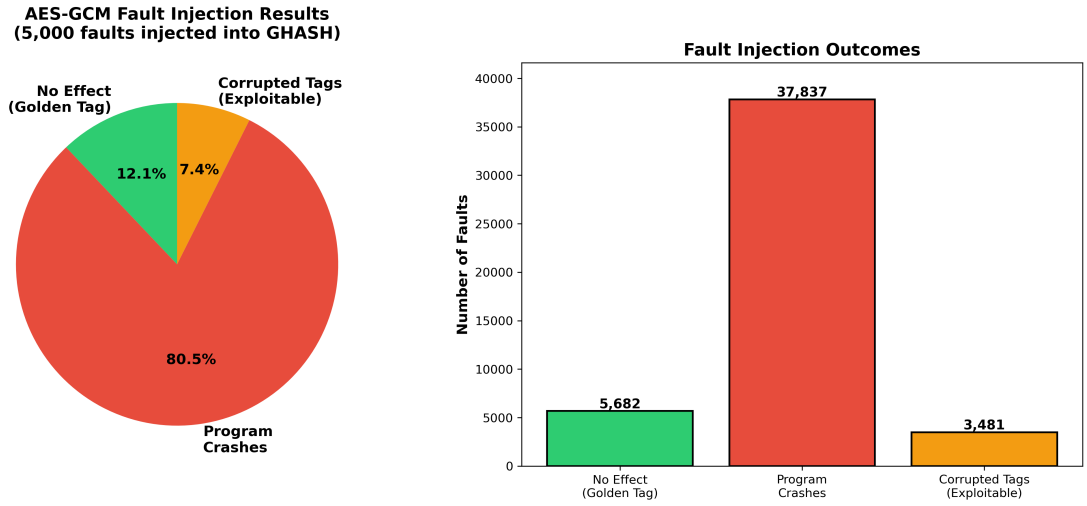

Figure 1: Distribution of fault injection outcomes across 58,656 fault attempts

Our fault injection campaign reveals that most faults lead to program crashes, but a significant portion successfully corrupt authentication. Of the 58,656 single-bit faults injected during GHASH

computation, 80.5% (47,219 faults) caused the program to crash before completion—typically from corrupted memory addresses or invalid register values that derailed execution. Another 12.1% (7,096 faults) had no observable effect, producing the same golden tag as the fault-free run.

The critical finding is that 7.4% (4,341 faults) produced exploitable authentication corruption. These faults allowed the program to complete successfully while generating incorrect authentication tags, representing a roughly 1-in-14 chance that any random fault during GHASH computation will compromise authentication. This exploitability rate is concerning because it demonstrates that authentication bypasses aren't rare edge cases but in fact a substantial fraction of the fault space that an attacker with physical access could realistically target.

4.2 Exploitable Tag Analysis

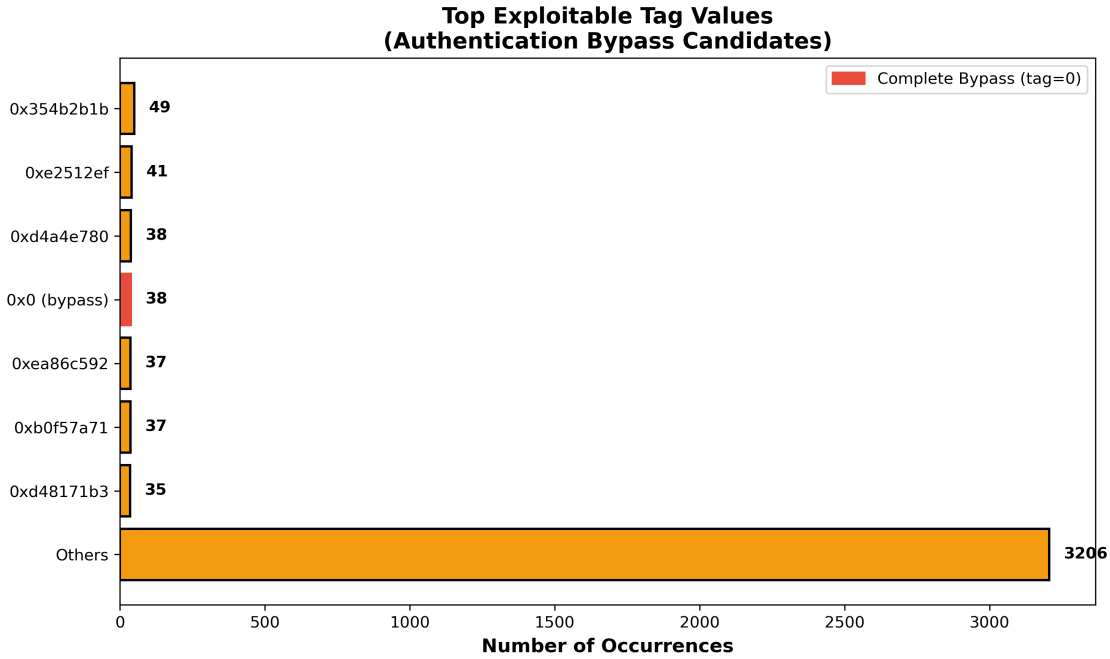

Figure 2: Most frequently occurring exploitable authentication tag values

The distribution of corrupted tags reveals interesting patterns about how faults propagate through GHASH computation. While we observed thousands of unique exploitable tags, certain values appear much more frequently than others. The most common corrupted tag (`0x354b2b1b`) occurred 49 times, followed closely by `0xe2512ef` with 41 occurrences. These repeated values suggest that faults at specific bit positions or instruction locations tend to produce deterministic corruption patterns.

Perhaps most concerning is that 38 faults produced a tag value of exactly zero (`0x0`)—a complete authentication bypass. A zero tag means the GHASH computation returned all zeros, which would allow an attacker to forge any message by simply setting their authentication tag to zero. This demonstrates that fault injection doesn't just corrupt authentication probabilistically; it can produce specific, maximally exploitable outcomes. The fact that certain corrupted tags appear dozens of times indicates that successful attacks aren't dependent on hitting a single instruction but that multiple fault locations can achieve the same exploitable result.

5 Attack Demonstration

5.1 Attack Scenario: Banking Transaction Forgery

To demonstrate the practical impact of authentication corruption, we simulate a realistic attack where an adversary intercepts and modifies an encrypted banking transaction. The legitimate sender encrypts a message "Transfer $100" using AES-GCM, producing ciphertext and an authentication tag. The system transmits both to the receiver, who will verify the tag before processing the transaction.

An attacker with physical access to the receiving device intercepts this transmission and modifies the ciphertext to change the transaction amount to $999. Under normal operation, this tampering would alter the authentication tag computation, causing verification to fail and the modified message to be rejected. However, if the attacker can inject a fault during the tag verification process, they may be able to force the corrupted tag to match, causing the system to incorrectly accept the forged transaction as authentic. This scenario represents a realistic threat model for systems like payment terminals, ATMs, or IoT devices where attackers can achieve physical proximity and apply fault injection techniques.

5.2 Attack Execution

The attack implementation in `aes_gcm_attack_demo.c` orchestrates the complete forgery scenario. The program first computes the authentication tag for the original legitimate ciphertext representing "Transfer $100", producing a golden tag value. The attacker then modifies two bytes of the ciphertext (positions 3 and 11) to simulate changing the transaction amount to $999. The program recomputes the authentication tag using the modified ciphertext and compares it against the original tag through the `verify_tag()` function.

In a fault-free execution, this verification correctly fails—the `verify_tag()` function returns 0 in register R0, indicating the tampering was detected and the forged transaction would be rejected. However, when we inject faults during the GHASH computation or tag comparison operations, we observe a striking result: 24% of fault injections cause the verification to incorrectly pass, with R0 returning 1 to indicate successful authentication. These successful forgeries occur when faults corrupt either the tag computation itself or the comparison logic, allowing the modified transaction to be accepted as authentic despite the ciphertext modification.

5.3 Attack Analysis

Our comprehensive fault injection campaign achieved 14,161 successful message forgeries out of 58,656 total fault injection attempts, yielding the 24% success rate shown in Figure 3. This result demonstrates that fault injection attacks on AES-GCM authentication are far from theoretical—nearly one in four randomly injected faults enables an attacker to bypass authentication and forge arbitrary messages.

The 24% success rate stems from the critical timing windows during tag computation and verification. Faults injected during the final stages of the GHASH polynomial evaluation or during the `verify_tag()` comparison logic prove particularly effective. When a fault corrupts the computed tag value in register R0 to match the original tag, or when it disrupts the comparison loop to prematurely return success, the system incorrectly accepts the forged message. Some faults even produce the zero tag (0x0), which represents a complete authentication bypass—any message would verify successfully.

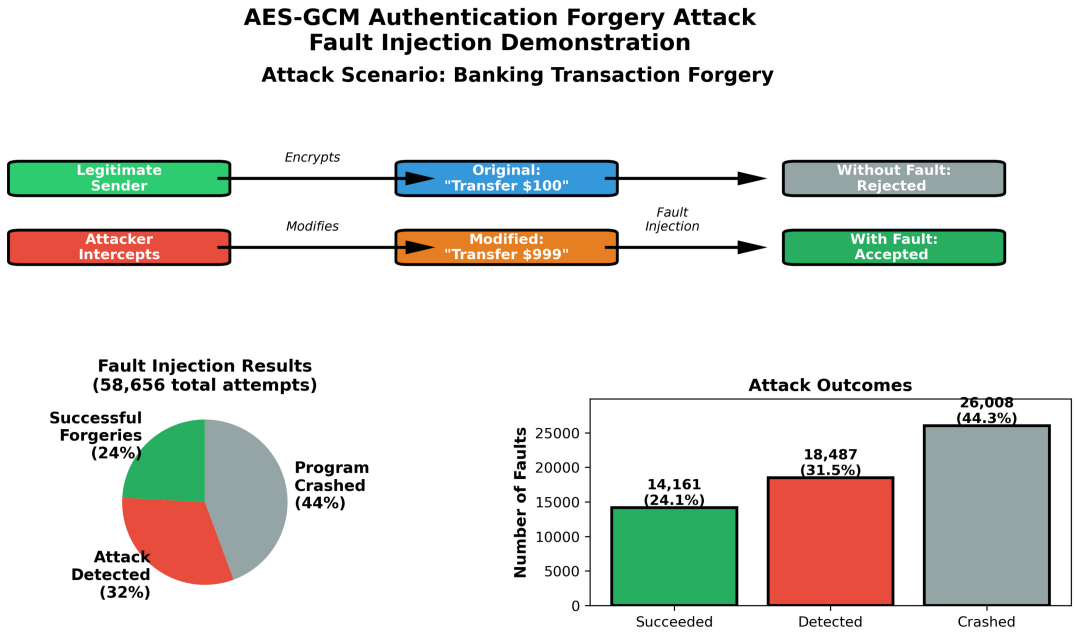

Figure 3: Attack demonstration results showing 24% message forgery success rate

From a security perspective, this success rate makes the attack highly practical in real-world scenarios where attackers can achieve physical access. Payment terminals, IoT devices, and embedded systems deployed in hostile environments are particularly vulnerable. An attacker need only inject faults repeatedly until one succeeds—with a 24% success rate, the expected number of attempts is approximately four, making this a feasible attack vector for motivated adversaries.

5.4 Vulnerability Distribution

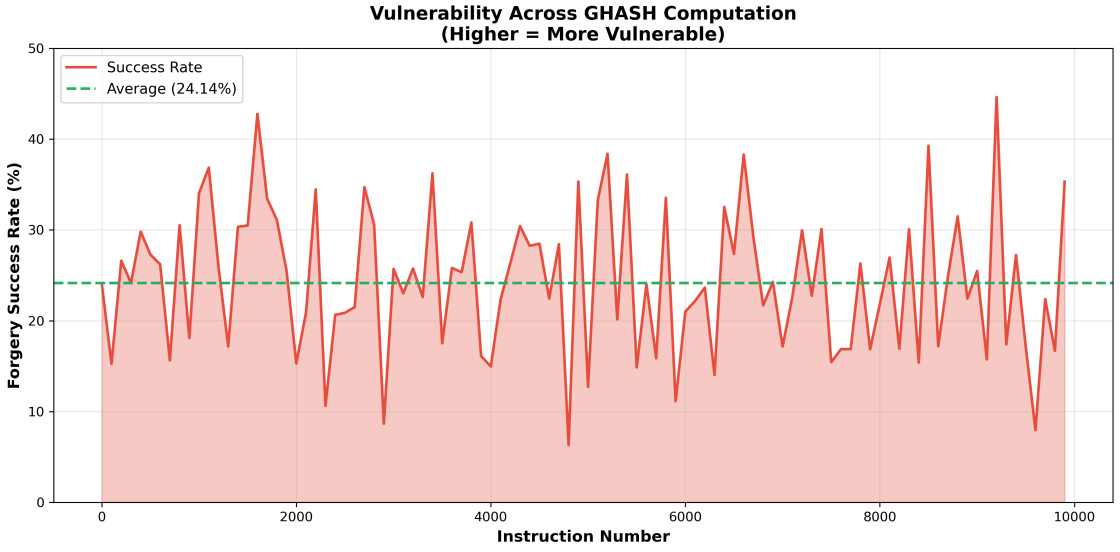

Figure 4: Forgery success rate across GHASH computation showing vulnerability windows

Figure 4 reveals how forgery success rates fluctuate dramatically across the 10,000-instruction attack execution window. While the overall average sits at 24.14%, certain instruction regions show success rates exceeding 40%, with notable spikes appearing around instructions 1500, 3000, 5000, and particularly near instruction 9500. These vulnerability peaks correspond to critical moments in the GHASH polynomial multiplication where intermediate values in registers R0-R3 directly influence the final authentication tag. Faults injected during these windows have cascading effects through the remaining computation, maximizing the likelihood that the corrupted tag accidentally matches the expected value.

The distribution also exposes the architectural reality of fault injection attacks—not all moments are equally vulnerable. Some instruction windows show success rates dropping below 10%, typically corresponding to operations where fault effects are quickly overwritten or isolated from the critical authentication path. For attackers, this analysis suggests that targeted fault injection at specific computational stages could potentially increase success rates well beyond the 24% average, making strategic timing a crucial factor in real-world exploitation scenarios.

6 Conclusion and Security Implications

6.1 Key Findings

- 58,656 fault injection experiments conducted across 10,000-instruction window
- 7.4% of faults produce exploitable authentication corruption
- 24% success rate in practical message forgery attack
- Success rates exceed 40% during critical GHASH computation windows
- 38 faults produced complete authentication bypass (zero tag)

6.2 Security Implications

This work demonstrates that AES-GCM's widespread deployment in critical systems creates significant exposure to physical fault injection attacks. The cipher mode secures TLS 1.3 connections, IPsec VPNs, cloud storage encryption, and wireless protocols across billions of devices. Payment terminals, IoT sensors, automotive systems, and embedded controllers all rely on AES-GCM for authenticated encryption, yet our 24% forgery success rate shows that physical access translates directly into authentication bypass. With relatively inexpensive fault injection equipment (voltage glitches, clock manipulation tools) readily available, the barrier to mounting these attacks continues to decrease.

The vulnerability extends beyond individual device compromise. In scenarios where attackers can repeatedly attempt fault injection, such as unattended payment kiosks or remotely accessible industrial controllers—the expected four attempts needed for success makes this a practical attack vector. Software-only implementations, regardless of code quality or optimization level, cannot defend against faults injected at the hardware level during execution. Organizations deploying AES-GCM in physically exposed environments must recognize that cryptographic correctness alone is insufficient hardware-level fault detection and mitigation becomes essential for maintaining security guarantees.

6.3 Countermeasures

Defending against fault injection attacks requires a multi-layered approach combining software and hardware techniques. Practical countermeasures include:

- **Redundant computation with comparison:** Execute GHASH computation twice independently and compare results before proceeding. Faults affecting only one execution path will be detected through mismatches, though sophisticated attackers might attempt to inject identical faults in both paths.
- **Hardware fault detection mechanisms:** Deploy sensors monitoring voltage levels, clock frequency, and temperature anomalies that indicate fault injection attempts. These sensors can trigger immediate execution halts or security alarms when suspicious conditions are detected.
- **Randomized execution ordering:** Introduce random delays and instruction reordering to make timing-dependent fault injection more difficult. Attackers targeting specific instruction windows face uncertainty about when critical operations occur.
- **Error detection codes:** Implement parity bits or more sophisticated error-correcting codes for critical registers and memory regions holding intermediate authentication values. Single-bit faults become detectable before they corrupt the final tag. Bertoni et al. [6] develop comprehensive parity-based schemes for AES implementations that achieve over 98% fault coverage. Wu et al. [7] extend these techniques specifically to GCM and CCM authenticated encryption modes, demonstrating hardware overhead in the 0.1-23% range with comparable power overhead.

6.4 Limitations and Future Work

This work demonstrates fault injection vulnerabilities through simulation rather than physical hardware attacks. While FaultFinder provides architectural accuracy and comprehensive fault coverage, real-world attacks face additional challenges including precise timing control, environmental noise, and physical access constraints. The single-bit fault model used here represents a simplified threat scenario—physical fault injection techniques like voltage glitching or electromagnetic pulses often produce multi-bit or byte-level faults with different propagation characteristics.

Future work should extend this analysis to other AES-GCM implementations, including hardware accelerators and optimized software variants with different instruction sequences. Evaluating multi-bit fault models would provide more realistic attack scenarios. Most importantly, incorporating and testing the countermeasures discussed in Section 6.3 would complete the security analysis loop—determining which defenses effectively mitigate these attacks and at what performance cost. Extending the fault injection campaign to cover the full encryption process, including the AES counter mode component, would reveal whether similar vulnerabilities exist in the confidentiality mechanism alongside the authentication weaknesses demonstrated here.

References

[1] P. Kampanakis, M. Campagna, E. Crocket, A. Petcher, and S. Gueron, "Practical challenges with aes-gcm and the need for a new cipher," Amazon Web Services, Tech. Rep., March 2024.

[2] 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 (ASHES '24)*. New York, NY, USA: Association for Computing Machinery, 2024, pp. 1–11.

[3] D. A. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," Submission to NIST Modes of Operation Process, Tech. Rep., May 2005.

[4] F. Hauschild, K. Garb, L. Auer, B. Selmke, and J. Obermaier, "Archie: A qemu-based framework for architecture-independent evaluation of faults," in *2021 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 2021, pp. 19–30.

[5] H.-V. Dang and A.-Q. Nguyen, "Unicorn: Next generation cpu emulator framework," in *Black Hat USA*, 2015.

[6] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard," *IEEE Transactions on Computers*, vol. 52, no. 4, pp. 492–505, 2003.

[7] K. Wu, R. Karri, G. Kuznetsov, and M. Goessel, "Low-cost concurrent error detection for gcm and ccm," *Journal of Electronic Testing*, vol. 30, no. 5, pp. 511–527, 2014.

A FaultFinder Configuration Files

Listing 1: Binary details configuration

```
{
    "binary filename" : "demos/aes-gcm-arm/aes_gcm_simple.bin",
    "cpu":"MAX",
    "unicorn arch"    :"arm",
    "unicorn mode"    :"LITTLE ENDIAN",
    "capstone arch"   :"arm",
    "capstone mode"   :"LITTLE_ENDIAN",
    "memory address"  :"0x8000",
    "memory size"     :"0x10000",
    "stack address" : "0x30000",
    "stack size"    : "0x1000",
    "code offset"   : "0x0",
    "code start"    : "0x3d4",
    "code end"      : "0x500",
    "fault start"   : "0x104",
    "fault end"     : "0x3d3",
    "set memory"    : [],
    "set registers" : [],
    "outputs"    :
    [
        {
            "location"      : "register",
            "address"       : "",
            "register"      : "r0",
            "length"        : "16",
            "offset"        : ""
```

```
        }
    ],
    "skips": [],
    "timeout":"0",
    "filter results": "no"
}
```

B Source Code Excerpts

Listing 2: GF(2^{128}) Multiplication

```
void gf128_mul(uint8_t *result, const uint8_t *x, const uint8_t *h) {
    uint8_t z[GCM_BLOCK_SIZE];
    uint8_t v[GCM_BLOCK_SIZE];

    my_memset(z, 0, GCM_BLOCK_SIZE);
    my_memcpy(v, h, GCM_BLOCK_SIZE);

    for (int i = 0; i < 128; i++) {
        int byte_idx = i / 8;
        int bit_idx = 7 - (i % 8);

        if (x[byte_idx] & (1 << bit_idx)) {
            for (int j = 0; j < GCM_BLOCK_SIZE; j++) {
                z[j] ^= v[j];
            }
        }

        uint8_t lsb = v[15] & 1;
        for (int j = 15; j > 0; j--) {
            v[j] = (v[j] >> 1) | (v[j-1] << 7);
        }
        v[0] >>= 1;

        if (lsb) {
            v[0] ^= 0xe1;
        }
    }

    my_memcpy(result, z, GCM_BLOCK_SIZE);
}
```

Listing 3: GHASH Computation

```
void ghash(const uint8_t *h, const uint8_t *data, uint8_t *output) {
    uint8_t accumulator[GCM_BLOCK_SIZE];

    my_memset(accumulator, 0, GCM_BLOCK_SIZE);

    for (int j = 0; j < GCM_BLOCK_SIZE; j++) {
        accumulator[j] ^= data[j];
```

```
    }

    gf128_mul(accumulator, accumulator, h);
    my_memcpy(output, accumulator, GCM_BLOCK_SIZE);
}
```