Fault-Injection Based Chosen-Plaintext Attacks on Multicycle AES Implementations

Yadi Zhong and Ujjwal Guin
Auburn University, Auburn, Alabama, USA
{yadi,ujjwal.guin}@auburn.edu

ABSTRACT

Hardware implementations of cryptographic algorithms offer significantly higher throughput on both encryption and decryption than their software counterparts. Advanced Encryption Standard (AES) is a widely used symmetric block cipher for data encryption. The most commonly used architecture for AES hardware implementations is the multicycle design, where each round uses the same hardware resource multiple times to increase area efficiency. In this paper, we successfully decouple the interdependency of multiple key bytes from the AES encryption. Thus, we solve each key byte separately with an overall attack complexity in $\boldsymbol{O}(2^8)$. Moreover, we uniquely determine each key byte through a chosen set of three plaintext-ciphertext pairs. We propose two novel chosen-plaintext attacks on multicycle AES implementations. Both attacks can eliminate the key diffusion from the MixColumns and Key Schedule modules. The first attack takes advantage of vulnerable AES implementations where an adversary can observe the output of each round. The second attack is based on fault injection, where a single fault on the completion-indicator register is sufficient to launch the attack. Because no faults are injected in the internal computations of AES, the current fault detection mechanisms are bypassed as no intermediate result has been altered. Lastly, we explore the theoretical aspect for the inherent property of our attacks.

CCS CONCEPTS

• **Security and privacy → Cryptanalysis and other attacks**; **Hardware attacks and countermeasures**.

KEYWORDS

AES, S-box, fault injection attack, chosen-plaintext attack.

ACM Reference Format:
Yadi Zhong and Ujjwal Guin. 2022. Fault-Injection Based Chosen-Plaintext Attacks on Multicycle AES Implementations. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* GLSVLSI, Orange County, CA, USA, 6 pages. https://doi.org/10.1145/3526241.3530826

1 INTRODUCTION

Advanced Encryption Standard (AES) [32] is one of the most common encryption algorithms used in various applications and protocols, e.g., disk encryption, Internet Protocol Security, Transport Layer Security, *etc.*. With the objective to speed up the execution of algorithms, there is an increasing demand for creating dedicated hardware [13, 15] other than optimizing software. This holds for cryptographic algorithms as well [9, 14], and their hardware implementations offer better performance on both encryption and decryption. For example, these low-cost hardware implementations of AES can be well suited for different Internet of Things (IoT) and Cyber-Physical Systems (CPS) for enabling data security, which is practically non-existent for low-cost IoT devices [36]. The hardware implementation of AES is generally multicycle, where each round uses the same hardware resource to provide better area efficiency. Due to the extensive use of AES across diverse sectors, hardware designers can reference the open-source HDL designs which are available in OpenCores [28]. These implementations offer designers greater flexibility by choosing the one that matches their design criteria.

Over the years, different researchers have proposed various types of attacks on AES, and successful countermeasures have also been proposed. AES attacks can be divided into three categories – algebraic attacks [3, 7, 10, 11], differential fault analysis [1, 5, 12, 17, 21], and side-channel attacks [19, 34]. Dunkelman et al. [11] have theoretically explored the effect of excluding the MixColumns transformation. For a reduced-round AES-128 with a single round, they can eliminate the majority of keys by sequentially guessing four bytes in the round key and discard those that failed consistency checks. It needs 2^{16} trial encryptions, and the search complexity is $\boldsymbol{O}(2^{32})$. Bouillaguet et al. [7] require $\boldsymbol{O}(2^{40})$ simulations [6] with one known-plaintext of a full round of encryption for key derivation. They conjecture the time complexity under the same setup with one more known-plaintext; however, the attack may not uniquely determine the AES key. Besides algebraic analysis, differential fault analysis becomes popular where errors or faults are introduced either inside the computation of a particular round or within the Key Schedule algorithm. Blömer et al. [5] proposed an attack on AES by resetting a bit after the XOR of input key and plaintext to zero. Observing the difference in output ciphertext, it helps the attacker decipher one key bit per fault. However, this attack needs to inject faults at the metal wires with extreme timing precision [4]. Moradi et al. [21] and Pogue et al. [29] perform differential fault analysis to extract the secret key, where faults are assumed at the encryption rounds. Ali et al. [1], Giraud et al. [12], and Kim [17] have successfully retrieved the key with differential fault analysis when faults occur at the Key Schedule. Multiple fault detection schemes [2, 16, 22] have been proposed to identify faults through either error detection codes or partial replication of the internal computation of AES. Several fault-resilient AES implementations have also been proposed [18, 20, 33].

In this paper, we propose two attacks to break the multicycle AES implementations. Both attacks take three plaintext-ciphertext pairs to evaluate one key byte. The first attack is designed to break a vulnerable AES implementation that leaks round operation to the output. An adversary can also access the internal scan chains and observe the round register value. By varying one byte in plaintext, we show that the effect of three other key bytes of the same MixColumns operation, along with key addition, can be eliminated by XORing the same output byte using only two plaintext-ciphertext pairs. However, one more plaintext-ciphertext pair is necessary to remove the redundant solution to uniquely determine the correct key byte. The second attack focuses on breaking a multicycle AES implementation where the content in ciphertext register is updated once all the round computations are complete. The adversary can launch the attack by injecting a fault in the flag register that signal the completion of all computations (e.g., done). Once a fault is injected, the internal states of the round registers are dumped as the ciphertext (see Figure 2). An adversary can perform the same steps mentioned in the first attack to determine the key after observing the first round result from the ciphertext. This attack is also applicable to the encryption round that skips MixColumns. As no faults are injected in the internal computations of AES, the traditional fault detection schemes can not identify this attack. We show both attacks on different AES implementations with 128-bit key size (AES-128) from the OpenCores benchmark site with the key search space complexity of $\boldsymbol{O}(2^8)$. The same attacks can be applied to other AES implementations with 192/256-bit keys.

The contributions of this paper are summarized as follows:

- *Exhaustive key-byte search using three chosen plaintext-ciphertext pairs:* The exhaustive key search attack relies on the availability of the internal round at the output of multicycle AES implementation. We show that an adversary only requires nine plaintext-ciphertext pairs in total to decipher the entire 128-bit key.
- *Fault injection attack:* Fault injection is necessary when an adversary cannot observe the internal state of the round register that holds the output of each round. This fault injection attack focuses on bypassing the entire computation of AES and dumps the internal state as ciphertext. We show that only one fault is necessary to launch the attack. The fault is injected at the completion-indicator register and does not affect the internal computations of AES, which makes traditional error detection schemes ineffective.

The rest of the paper is organized as follows. We briefly introduce the background for AES and describe the threat model in Section 2. Our proposed attacks are presented in Section 3. A theoretical perspective for our proposed attacks is analyzed in Section 4. Finally, we conclude the paper in Section 5.

2 BACKGROUND AND THREAT MODEL

2.1 AES Implementation

The hardware implementations of AES are often multicycle, where area efficiency is assured since all the rounds use the same resources at different clock cycles. Depending on the key size, the ciphertext is produced after 10, 12, or 14 clock cycles. If each encryption takes more than one clock cycle to finish, the ciphertext is generated after an integer multiple of 10, 12, or 14 clock cycles for AES-128, AES-192, and AES-256, respectively. Each encryption round contains SubBytes (SB), ShiftRows (SR), MixColumns (MC), and AddRoundKey ($\bigoplus$) modules, while the last one skips MC, as shown inside the dashed box of Figure 1.

Notations: We use the following notations to maintain uniformity across the entire paper.

- We adopt the following notations, where the superscript index (i) in a variable indicates the current encryption round and the subscript index (j) are for the byte index, from 0 to 15. For example, the j^{th} byte in i^{th} round key is K_j^i and R^i is the result for i^{th} encryption round. We use K, without any superscript, to refer to the input key, which can be either 16, 24, or 32-byte for AES-128, AES-192, or AES-256. To facilitate the derivation in the subsequent sections, we abbreviate the round one result R^1 to R. Aside from the input key, the size for all other variables, e.g., plaintext P, round register R^i, round key K^i, and ciphertext C, are 16 bytes. Note that, in this paper, the ciphertext register C may not contain the actual encrypted data. C, as explained in Section 3, can store the internal round result, before the complete encryption finishes. We use ciphertext and round one result R interchangeably.
- The S-box function is denoted as $s(\cdot)$. Round constant is RC.

2.2 Threat Model

As the hardware implementations of different crypto primitives become prevalent, an adversary can launch the attacks by physically accessing the device. This section describes the adversarial capabilities that help to carry out the attacks. The threat model is summarized as follows:

- The attacker possesses a fully functional chip where the secret key has already been programmed. For example, an electronic device that ensures secure communication can be obtained from an IoT/CPS application. By having the device, an adversary can apply a plaintext and observe its corresponding ciphertext.
- The adversary can obtain the gate-level netlist of the AES implementation. It can be either acquired through IC reverse engineering [30] or from the GDSII files [35]. As the majority of the IC production is offshore, an untrusted foundry can also provide the reverse-engineered netlist to the adversary.
- The adversary can have access to the design-for-testability (DFT) or scan architecture to observe the internal state of the design (e.g., round registers). He/she can launch our proposed first attack to obtain the secret key. Note that the DFT architecture provides the necessary support for manufacturing tests [8]. If not, the attacker can use fault injection equipment to inject a fault in the completion-indicator (CI) register to launch our proposed second attack. Laser fault injection equipment can induce very precise faults and target a single flip-flop [31]. As this equipment is available at universities, we assume that an adversary also has the means to acquire such equipment.

3 PROPOSED ATTACKS ON MULTICYCLE AES IMPLEMENTATIONS

We present two attacks to efficiently break multicycle AES implementations. The first attack exploits the minor issues in the implementations [25–27], where the round registers and ciphertext are updated simultaneously in every round. The second attack, however, requires fault injection for breaking a correct multicycle AES implementation which assigns the round result to the ciphertext output only when it is in the final round [24]. Both attacks work on all three key sizes of AES. For simplicity of discussion, in this section, we will first show all the attacks on AES-128. The same attack methodology can be applied to AES-192 and AES-256 with constant overhead in worst-case search complexity, which we briefly describe how to extend both attacks to key sizes larger than 128-bit at the end.

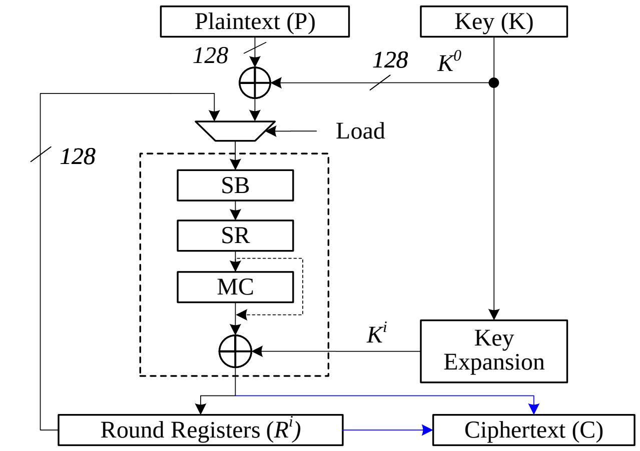


Figure 1: Hardware implementation of multicycle AES-128 [25–27].

3.1 Proposed Exhaustive Key-Byte Search Attack

Our first attack is specific to the multicycle AES implementations, where the output for each encryption round can be observed as ciphertext C. Figure 1 shows an abstract view of the multicycle AES implementation when a designer incorrectly implements AES or an adversary has access to the internal scan chains. For example, one implementation from OpenCores [27] assigns the i^{th} round result, $MC^i \oplus K^i$, straight to the ciphertext output, and other implementations [25, 26] connect the ciphertext to the round registers. Both data-paths are highlighted in blue in Figure 1.

The traditional AES implementation mixes different key bytes in such a way that an adversary cannot remove the interdependency among the key bytes in the ciphertext. As a result, AES remains secure no matter how many plaintext-ciphertext pairs one can observe. However, if an adversary can observe the round outputs stored in the round registers of a multicycle AES implementation, it is possible to remove the dependency across the key bytes. We show that one key byte can be determined without knowing the other key bytes in this attack. Thereby, we determine a key byte through simulating all 2^8 key combinations and compare the result with one byte round register value from the working chip under attack to derive the correct key. In the following, we present the detailed steps to obtain the first key byte, K_0. Similar analysis can be performed to reveal the other key bytes.

- Step-1: The adversary chooses two plaintexts, P and P', and observes the two corresponding round outputs R and R' after the first clock cycle from the chip.
- Step-2: The first byte (R_0) of round output R is computed using the following equation:

$$\begin{aligned} R_0 &= \{MC(SR(SB(P \oplus K^0))) \oplus K^1\}_0 \\ &= \{MC(SR(SB(P \oplus K^0)))\}_0 \oplus K^1_0 \\ &= [02 \otimes s(P_0 \oplus \boldsymbol{K_0}) \oplus 03 \otimes s(P_5 \oplus \boldsymbol{K_5}) \oplus (P_{10} \oplus \boldsymbol{K_{10}}) \oplus \\ &\quad s(P_{15} \oplus \boldsymbol{K_{15}})] \oplus [K_0 \oplus s(K_{13}) \oplus RC]. \end{aligned} \tag{1}$$

From Equation 1, we can observe that the value of R_0 depends on K_0, K_5, K_{10}, K_{13}, and K_{15}. Key bytes which can be derived from the equation are highlighted as boldface letters. At this point, it is sub-optimal to brute force all five key bytes as they cannot be uniquely determined. Multiple collisions occur for the 2^{40} key combinations that lead to the same 8-bit R_0 value. Therefore, we choose another plaintext P' with the following properties:

$$P_0 \neq P'_0, \text{ and } P_i = P'_i,\ i = 5, 10, 15$$

The R'_0 can be computed using the following equation:

$$\begin{aligned} R'_0 &= \{MC(SR(SB(P' \oplus K^0))) \oplus K^1\}_0 \\ &= \{MC(SR(SB(P' \oplus K^0)))\}_0 \oplus K^1_0 \\ &= [02 \otimes s(P'_0 \oplus \boldsymbol{K_0}) \oplus 03 \otimes s(P_5 \oplus \boldsymbol{K_5}) \oplus (P_{10} \oplus \boldsymbol{K_{10}}) \oplus \\ &\quad s(P_{15} \oplus \boldsymbol{K_{15}})] \oplus [K_0 \oplus s(K_{13}) \oplus RC]. \end{aligned} \tag{2}$$

The computation details can be found in [32].

- Step-3: R_0 and R'_0 are XORed to remove the dependency for other key bytes.

$$\begin{aligned} R_0 \oplus R'_0 &= [02 \otimes s(P_0 \oplus \boldsymbol{K_0}) \oplus 03 \otimes s(P_5 \oplus \boldsymbol{K_5}) \oplus (P_{10} \oplus \boldsymbol{K_{10}}) \\ &\quad \oplus s(P_{15} \oplus \boldsymbol{K_{15}})] \oplus [K_0 \oplus s(K_{13}) \oplus RC] \oplus [02 \\ &\quad \otimes s(P'_0 \oplus \boldsymbol{K_0}) \oplus 03 \otimes s(P_5 \oplus \boldsymbol{K_5}) \oplus (P_{10} \oplus \boldsymbol{K_{10}}) \oplus \\ &\quad s(P_{15} \oplus \boldsymbol{K_{15}})] \oplus [K_0 \oplus s(K_{13}) \oplus RC] \\ &= 02 \otimes s(P_0 \oplus \boldsymbol{K_0}) \oplus 02 \otimes s(P'_0 \oplus \boldsymbol{K_0}) \end{aligned} \tag{3}$$

We can rewrite Equation 3 as:

$$s(P_0 \oplus \boldsymbol{K_0}) \oplus s(P'_0 \oplus \boldsymbol{K_0}) = M_0 \tag{4}$$

where, M_0 is a constant.

- Step-4: Brute-force attack is performed using Equation 4.

The only unknown in Equation 4 is K_0, allowing the attacker to enumerate all 256 combinations of K_0 to find the one that satisfies it. However, there exists more than one solution due to the nonlinearity introduced by the S-box.

Claim-1. There exist two solutions for Equation 5.

$$s(p \oplus \boldsymbol{k}) \oplus s(p' \oplus \boldsymbol{k}) = m \tag{5}$$

where, p, p', k, and m are of one byte, and $p \neq p'$.

Observation. There are 128 unique values for byte m for all 256 combinations of k under any fixed p, p' and $p \neq p'$. One can verify the above observation using code available in GitHub [23]. We denote the 128 unique values of m as m_i, $i \in 0, 1, ..., 127$ and $m_i \neq m_j$ when $i \neq j$.

Proof. For each valid m_i, there is at least one unique key k^I_i, $i \in 0, 1, ..., 127$, that satisfies:

$$s(p \oplus k^I_i) \oplus s(p' \oplus k^I_i) = m_i \tag{6}$$

Due to $m_i \neq m_j$, when $i \neq j$, and the bijective property of the S-box, $k^I_i \neq k^I_j$, when $i \neq j$, with fixed p, p', $p \neq p'$. We denote the set of these 128 solutions as Group I. The proof for the existence of another solution for each m_i is sufficient for validating the claim.

Let us consider another key byte k^{II}_i with the form $k^{II}_i = k^I_i \oplus p \oplus p'$. Clearly, $k^{II}_i \neq k^I_i$ since $p \neq p'$.

Applying the value of k^{II}_i in Equation 6, we compute:

$$\begin{aligned} & s(p \oplus k^{II}_i) \oplus s(p' \oplus k^{II}_i) \\ &= s(p \oplus k^I_i \oplus p \oplus p') \oplus s(p' \oplus k^I_i \oplus p \oplus p') \\ &= s(k^I_i \oplus p') \oplus s(k^I_i \oplus p) = m_i \end{aligned} \tag{7}$$

Next, we show that all these 128 k^{II}_i's are unique as well. Now consider any two solutions k^{II}_i and k^{II}_j, $\forall i \neq j$. As $k^I_i \neq k^I_j$, then $(k^I_i \oplus p \oplus p') \neq (k^I_j \oplus p \oplus p')$. This results $k^{II}_i \neq k^{II}_j$. This proves that no two $k_i^{II}, k_j^{II}, i \neq j$ are the same, and we denote the set of these 128 solutions as Group *II*.

Finally, the proof will be complete, if we show that there is no overlap between the solutions in Group *I* (k_i^I's) and Group *II* (k_i^{II}'s), where each group contains 128 unique key within. Let us assume that k_i^{II} belongs to Group *I* and denote with index j, where $i \neq j$, $k_i^{II} = k_j^I$. Substitute k_j^I in Equation 6, and we get

$$s(p \oplus k_j^I) \oplus s(p' \oplus k_j^I) = m_j. \tag{8}$$

Combining Equation 8 and Equation 7 for k_i^I, we have $m_i = m_j$, which contradict the uniqueness of 128 m_i's for $i \neq j$. Thus, there are no common solutions between Group *I* and Group *II* and the 256 solutions from the joint groups make up all the possible key-byte combinations. Therefore, we proved that there exist only two solutions for each m_i that satisfies Equation 5. □

- Step-5: The double-solution is removed by selecting another plaintext (P'') with P_0'' differs from both P_0 and P_0' (i.e., $P_0'' \neq P_0 \neq P_0'$), and keeping $P_5'' = P_5$, $P_{10}'' = P_{10}$, and $P_{15}'' = P_{15}$ unchanged. Using Step-2 and Step-3, we obtain the following equation:

$$s(P_0 \oplus \boldsymbol{K_0}) \oplus s(P_0'' \oplus \boldsymbol{K_0}) = N_0, \tag{9}$$

where, N_0 is a constant. Equation 9 is applied on the two previously obtained solutions (i.e., K_0^I and K_0^{II}) to determine the correct key byte.

Claim-2. Both solutions, K_0^I and K_0^{II}, cannot be valid under both Equations 4 and 9.

Proof. Let us assume that both solutions, K_0^I and K_0^{II}, are valid and satisfy Equation 9. As a result, we can write,

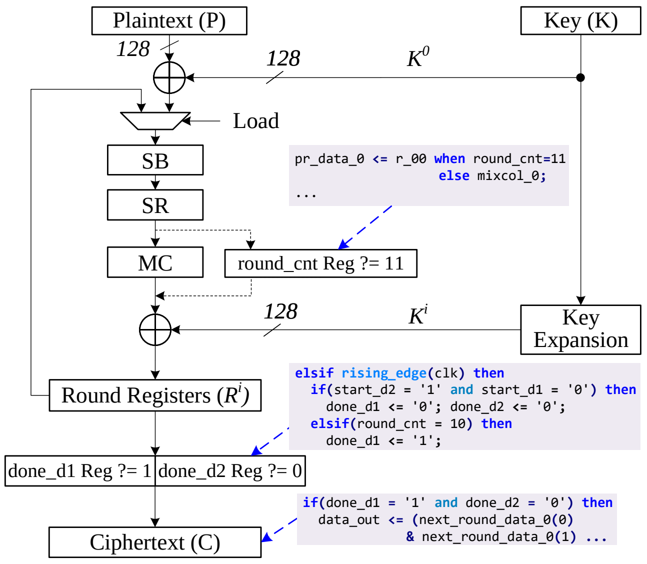
$$\begin{aligned} s(P_0 \oplus K_0^I) \oplus s(P_0'' \oplus K_0^I) &= N_0 \\ &\text{and} \\ s(P_0 \oplus K_0^{II}) \oplus s(P_0'' \oplus K_0^{II}) &= N_0 \end{aligned}$$

Using Claim-1, we can write $s(P_0 \oplus K_0^I) = s(P_0'' \oplus K_0^{II})$. Also, with Claim-1 and Equation 4, we can write $s(P_0 \oplus K_0^I) = s(P_0' \oplus K_0^{II})$. This results, $s(P_0' \oplus K_0^{II}) = s(P_0'' \oplus K_0^{II})$. This can't be true as $P' \neq P''$. □

In the same manner, key bytes K_5, K_{10}, and K_{15} are determined through either of the first four-bytes from round output, R_0, R_1, R_2, R_3, by varying the corresponding plaintext byte, P_5, P_{10}, or P_{15} and constraining the other three plaintext bytes to remain unchanged.

To find the remaining key bytes, we need to consider three bytes of the round register, one from $R_4 - R_7$, $R_8 - R_{11}$, and $R_{12} - R_{15}$ each. These three bytes are sufficient to find the remaining key bytes as their MixColumns transformation incorporate all 12 key bytes. For example, we consider R_4, R_8, and R_{12}, as shown below:

$$\begin{aligned} R_4 &= [02 \otimes s(P_4 \oplus \boldsymbol{K_4}) \oplus 03 \otimes s(P_9 \oplus \boldsymbol{K_9}) \oplus s(P_{14} \oplus \boldsymbol{K_{14}}) \oplus \\ &\quad s(P_3 \oplus \boldsymbol{K_3})] \oplus [K_4 \oplus K_0 \oplus s(K_{13}) \oplus RC] \qquad (10) \\ R_8 &= [02 \otimes s(P_8 \oplus \boldsymbol{K_8}) \oplus 03 \otimes s(P_{13} \oplus \boldsymbol{K_{13}}) \oplus s(P_2 \oplus \boldsymbol{K_2}) \oplus \\ &\quad s(P_7 \oplus \boldsymbol{K_7})] \oplus [K_8 \oplus K_4 \oplus K_0 \oplus s(K_{13}) \oplus RC] \qquad (11) \\ R_{12} &= [02 \otimes s(P_{12} \oplus \boldsymbol{K_{12}}) \oplus 03 \otimes s(P_1 \oplus \boldsymbol{K_1}) \oplus s(P_6 \oplus \boldsymbol{K_6}) \oplus \\ &\quad s(P_{11} \oplus \boldsymbol{K_{11}})] \oplus [K_{12} \oplus K_8 \oplus K_4 \oplus K_0 \oplus s(K_{13}) \oplus RC] \qquad (12) \end{aligned}$$



```
pr_data_0 <= r_00 when round_cnt=11
                  else mixcol_0;
...
```

```
elsif rising_edge(clk) then
  if(start_d2 = '1' and start_d1 = '0') then
    done_d1 <= '0'; done_d2 <= '0';
  elsif(round_cnt = 10) then
    done_d1 <= '1';
```

```
if(done_d1 = '1' and done_d2 = '0') then
  data_out <= (next_round_data_0(0)
             & next_round_data_0(1) ...
```

Figure 2: Structure of multicycle AES-128, where round result R^i is sent to ciphertext C only after the last round of encryption [24], code segment from Lines 319-326, 399-428.

The remaining key bytes can be determined iteratively using Equations 10-12 and Steps 1-5. Note that an adversary can also choose R_1, R_5, R_9 and R_{13} or a few other combinations to determine all 16 key bytes as well.

3.2 Proposed Fault-Injection Attack

The first attack is efficient in determining the key when an adversary can access the round registers that hold the output of each round. An adversary can use the scan architecture or exploit a faulty implementation for such a purpose. However, one cannot always assume access to registers. This motivates us to propose a fault injection attack that allows us to observe the internal state and utilize the previously presented brute-force attack. Note that the fault injection has become an effective means to launch an attack. It has been demonstrated that laser fault injection can successfully target a single register [31]. The same methodology and procedure in [31] is applicable to launching our proposed fault injection attack on AES. This attack leads to two possible scenarios, and both are elaborated below.

First, let us examine an example in OpenCores [24], which does not have the weakness of other implementations described in Section 3.1. Figure 2 shows this multicycle AES implementation where the ciphertext register receives the round register value at the last encryption round (e.g., when *done_d1 == 1* and *done_d2 == 0*). We also assume that an adversary does not have access to the internal scan chains and only observes the ciphertexts. The round operations are the same with Figure 1. We include the HDL code excerpt in Figure 2 which describes how the completion-indicator (CI) registers (e.g., *done_d1* and *done_d2*), and the ciphertext (i.e., *data_out*) are updated. As the *done_d2* register holds a logic *0* value during the round operations, the round registers values (e.g., *next_round_data_0*), are propagated to the ciphertext output (e.g., *data_out*) when *done_d1=1*. It is thus sufficient to inject only one logic *1* fault to *done_d1* register to extract round register value. Once the internal value is observed, an adversary can perform Steps 1-5 presented in Section 3.1 to retrieve the secret key completely.

Second, a hardware implementation can have the same logical condition applied to both skipping the MixColumns transformation and assigning the round register result to the ciphertext C, since both should happen at the last encryption round. In this way, if the attacker injects all the necessary fault to force the round result observable, the MixColumns step is also affected and bypassed. Alternatively, it may be possible for an adversary to inject faults on both CI register and round counter (e.g., *round_cnt*) so that he/she obtains the result from the first encryption round, but skips the MixColumns module. We briefly describe how an attack can be launched when bypassing the MixColumns operation.

• **Brute-force attack without MixColumns Operation.** Let us consider the first byte of round register R (i.e., R_0) after applying the first plaintext P, which can be computed as:

$$R_0 = s(P_0 \oplus \boldsymbol{K_0}) \oplus [K_0 \oplus s(K_{13}) \oplus RC].$$

After applying the second plaintext P' with $P_0 \neq P'_0$, we can write:

$$R'_0 = s(P'_0 \oplus \boldsymbol{K_0}) \oplus [K_0 \oplus s(K_{13}) \oplus RC].$$

Finally, we obtain

$$s(P_0 \oplus \boldsymbol{K_0}) \oplus s(P'_0 \oplus \boldsymbol{K_0}) = M_0$$

where, M_0 is a constant.

Then, we can follow the same procedure described in Steps 4-5 in Section 3.1 to recover key byte K_0.

3.3 Extending the Proposed Attacks to AES-192 and AES-256

The attacks presented in Sections 3.1-3.2 can retrieve the entire 16 bytes of the secret key for AES-128. It can also recover the first 16 bytes from AES-192 and AES-256, although the exact expression for round key in Key Expansion is different. However, it is necessary to extract the remaining 8 and 16 bytes for 192- and 256-bit keys, respectively. These key bytes belong to the second round key K^1, where they influence the result of the first encryption round through AddRoundKey ($\bigoplus$). As a result, the adversary can decipher these key bytes from any one of the plaintext-ciphertext pairs obtained in Section 3.1 or 3.2, without the need to give additional plaintexts to the oracle or perform extra fault injections. For AES-192, the first eight bytes of the round key K^1, $K^1_0, ..., K^1_7$, are the last 8 bytes of the input key K, $K_{16}, ..., K_{23}$, respectively [32]. Likewise, the entire round key K^1 is nothing but the last 16 bytes of input key K for AES-256, $K^1 = \{K_{16}, ..., K_{31}\}$ [32]. We show how to determine key byte K_{16} of AES-192 from the observed R in the following:

$$\begin{aligned} R_0 &= \{MC(SR(SB(P \oplus K^0))) \oplus K^1\}_0 \\ &= \{MC(SR(SB(P \oplus K^0)))\}_0 \oplus K^1_0 \\ &= [02 \otimes s(P_0 \oplus K_0) \oplus 03 \otimes s(P_5 \oplus K_5) \oplus (P_{10} \oplus K_{10}) \oplus \\ &\quad s(P_{15} \oplus K_{15})] \oplus [\boldsymbol{K_{16}}] \\ &= \boldsymbol{K_{16}} \oplus Q_1, \end{aligned} \tag{13}$$

where $Q_1 = 02 \otimes s(P_0 \oplus K_0) \oplus 03 \otimes s(P_5 \oplus K_5) \oplus (P_{10} \oplus K_{10}) \oplus s(P_{15} \oplus K_{15})$ is a constant as $K_0 - K_{15}$ are known. One can directly compute K_{16} by XORing R_0 and Q_1.

It is also possible to determine K_{16} directly, if adversary chooses to bypass the MixColumns operation. We can also write:

$$R_0 = s(P_0 \oplus K_0) \oplus [\boldsymbol{K_{16}}], \tag{14}$$

where $s(P_0 \oplus K_0)$ is a constant as $K_0 - K_{15}$ are known. The key byte K_{16} can be computed by XORing R_0 and $s(P_0 \oplus K_0)$ like before.

The other key bytes (i.e., $K_{17} - K_{23}$) can be obtained similarly from $R_1 - R_7$. One can perform similar analysis to obtain the remaining $K_{16} - K_{31}$ key bytes from $R_0 - R_{15}$ for AES-256.

3.4 Number of Plaintext Requirement

In Section 3.1, three plaintexts are sufficient to derive one key byte. Note, in these three plaintexts, we only vary one byte of the same index and constrain the bytes at the three other indices to remain unchanged. Because we do not have constraints on all 15 remaining plaintext bytes, we are allowed with more flexibility on other plaintext bytes that do not belong to the same MixColumns computation as the key byte of interest. Instead of the apparent $3 * 16 = 48$ plaintexts to recover all 16-byte key, we can reduce this number by having four plaintext bytes, where no two bytes resides in the same MixColumns operation, vary concurrently in one plaintext, e.g., P_0, P_4, P_8, P_{12}. Hence, the minimum number of plaintexts needed for this attack is $1+2\times4 = 9$, where the minimal three plaintexts are satisfied by having one reference plaintext and its corresponding two variants. Once all nine ciphertexts are obtained, all sixteen key bytes can be determined in parallel, making the worst-case complexity of $\boldsymbol{O}(2^8)$. Suppose the round result skips MixColumns transformation, as of the second possible scenario in Section 3.2, each key byte is still recovered with three plaintext-ciphertext pairs. However, since we do not have the restriction on the other three plaintext bytes as for Section 3.1, it does not matter if other bytes stay the same or not. Hence, we can reduce the required number of plaintexts to break the entire key from 9, as for the attack in Section 3.1, down to 3, as long as the byte at the same index (e.g., index j) is different in all three plaintexts, $P_j \neq P'_j \neq P''_j$. The worst-case complexity of this attack is still $2^8 = \boldsymbol{O}(2^8)$, since all sixteen key bytes, K_j's, can be recovered concurrently, without the need to wait for any other bytes to be resolved first.

4 THEORETICAL JUSTIFICATION OF DOUBLE-SOLUTION FOR EQUATION 5

Aside from the exhaustive simulation we performed in Section 3.1, we present another perspective on the dual solutions, k^I and k^{II}, for Equation 5. Instead of the common approach [10] to expand the S-box to a system of equations in a bit-by-bit manner, we consider the polynomial in $\boldsymbol{GF}(2^8)$ as the fundamental unit. To differentiate matrix multiplication from polynomial multiplication under $\boldsymbol{GF}(2^8)$ with irreducible polynomial $IP = [1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1]$, we use $\cdot$ for matrix multiplication. S-box $s(x) = y$ contains two operations [32]. It first find the inverse polynomial, x^{-1} of its input byte x under $\boldsymbol{GF}(2^8)$ and IP. Then, it applies the affine transformation on x^{-1} with a reversible matrix H of size 8×8 bits and an 8-bit column vector $c = [1\ 1\ 0\ 0\ 0\ 1\ 1\ 0]^T$ to get output byte $y = H \cdot x^{-1} \oplus c$ [32].

For our analysis, we can expand Equation 5 with the details of the internal construction of S-box as:

$$\left(H \cdot (p \oplus \boldsymbol{k})^{-1} \oplus c\right) \oplus \left(H \cdot (p' \oplus \boldsymbol{k})^{-1} \oplus c\right) = m.$$

Note that $(p \oplus \boldsymbol{k})^{-1}$ and $(p' \oplus \boldsymbol{k})^{-1}$ are the inverse of $(p \oplus \boldsymbol{k})$ and $(p' \oplus \boldsymbol{k})$, respectively. After rearranging terms, we get

$$H \cdot \left((p \oplus \boldsymbol{k})^{-1} \oplus (p' \oplus \boldsymbol{k})^{-1}\right) = m.$$

Since 8-by-8 bit matrix H has inverse, denoted as H^{-1}, we obtain

$$(p \oplus \boldsymbol{k})^{-1} \oplus (p' \oplus \boldsymbol{k})^{-1} = H^{-1} \cdot m.$$

Now, both sides of the equation are polynomials in $GF(2^8)$. We use constant d to represent $H^{-1} \cdot m$, $d = H^{-1} \cdot m$, for clarity.

If we multiply both side with polynomial $(p \oplus \boldsymbol{k})$ and $(p' \oplus \boldsymbol{k})$, we get

$$(p' \oplus \boldsymbol{k}) \oplus (p \oplus \boldsymbol{k}) = d \otimes (p \oplus \boldsymbol{k}) \otimes (p' \oplus \boldsymbol{k}).$$

We can further simplify it as

$$p \oplus p' = d \otimes \left[(\boldsymbol{k} \otimes \boldsymbol{k}) \oplus (p \oplus p') \otimes \boldsymbol{k} \oplus (p \otimes p')\right].$$

Since d is a polynomial under $GF(2^8)$ and it is not the zero polynomial, the inverse of d exists and we denote it as d^{-1}. Multiply both side with d^{-1} and abbreviate $k \otimes k$ as k^2 under $GF(2^8)$, we have

$$\boldsymbol{k}^2 \oplus (p \oplus p') \otimes \boldsymbol{k} \oplus (p \otimes p') \oplus d^{-1} \otimes (p \oplus p') = 0.$$

Thus, we complete the derivation and it is clear that Equation 5 is a quadratic equation with respect to the unknown variable $\boldsymbol{k}$.

Under any quadratic equation $x^2 + ax + b = 0$ in $\mathbb{R}$, the two roots x_1, x_2 satisfy $x_1 + x_2 = -b, x_1 \times x_2 = c$. We made an interesting observation that these properties also hold true for Equation 4. The two solution for Equation 5, k^I, k^{II} uphold both

$$\begin{aligned} k^I + k^{II} &= p \oplus p', \text{ and} \\ k^I \otimes k^{II} &= (k^{II} \oplus p \oplus p') \otimes k^{II} \\ &= (k^{II})^2 \oplus (p \oplus p') \otimes k^{II} \\ &= (p \otimes p') \oplus d^{-1} \otimes (p \oplus p'). \end{aligned}$$

5 CONCLUSION

In this paper, we presented two novel attacks targeting the hardware implementations of multicycle AES. In both attacks, each key byte requires only three plaintext-ciphertext pairs to retrieve its value. The entire secret key is recovered by solving all key bytes in parallel, resulting in a $O(2^8)$ worst-case complexity. If the internal round result is not observable in the output, we propose to inject fault on the completion-indicator register to reveal the internal state. Any traditional method can be applied to inject faults, and the protection against the fault injection attacks can be bypassed since no intermediate result is affected. We also showed the algebraic perspective on the dual solutions of Equation 5. Finally, we provide the theoretical extension of the properties of a regular quadratic equation to the finite field $GF(2^8)$, which support Claim-1.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant number CNS-1755733.

REFERENCES

[1] Sk Subidh Ali and Debdeep Mukhopadhyay. 2011. A Differential Fault Analysis on AES Key Schedule using Single Fault. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 35–42.

[2] Sabine Azzi, Bruno Barras, Maria Christofi, and David Vigilant. 2017. Using linear codes as a fault countermeasure for nonlinear operations: application to AES and formal verification. *Journal of Cryptographic Engineering* 7, 1 (2017), 75–85.

[3] Achiya Bar-On, Orr Dunkelman, Nathan Keller, Eyal Ronen, and Adi Shamir. 2020. Improved Key Recovery Attacks on Reduced-Round AES with Practical Data and Memory Complexities. *Journal of Cryptology* 33, 3 (2020), 1003–1043.

[4] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. 2012. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proc. IEEE* 100, 11 (2012), 3056–3076.

[5] Johannes Blömer and Jean-Pierre Seifert. 2003. Fault based cryptanalysis of the advanced encryption standard (AES). In *International Conference on Financial Cryptography*. Springer, 162–181.

[6] Charles Bouillaguet. 2011. https://www-almasty.lip6.fr/~bouillaguet/static/attacks/aes_1r_5_guesses.tar.gz

[7] Charles Bouillaguet, Patrick Derbez, Orr Dunkelman, Pierre-Alain Fouque, Nathan Keller, and Vincent Rijmen. 2012. Low Data Complexity Attacks on AES. *IEEE transactions on information theory* 58, 11 (2012), 7002–7017.

[8] Michael Bushnell and Vishwani Agrawal. 2004. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Vol. 17. Springer Science & Business Media.

[9] Intel Corp. 2012. Intel Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html

[10] Nicolas T Courtois and Josef Pieprzyk. 2002. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In *International conference on the theory and application of cryptology and information security*. Springer, 267–287.

[11] Orr Dunkelman and Nathan Keller. 2010. The Effects of the Omission of Last Round's MixColumns on AES. *Inform. Process. Lett.* 110, 8-9 (2010), 304–308.

[12] Christophe Giraud. 2004. DFA on AES. In *International Conference on Advanced Encryption Standard*. Springer, 27–41.

[13] Dion Harris. 2021. Nvidia's new CPU to 'grace' world's most powerful AI-capable supercomputer. https://blogs.nvidia.com/blog/2021/04/12/cpu-grace-cscs-alps/

[14] Apple Inc. 2018. Apple T2 Security Chip: Security Overview. https://www.apple.com/ca/mac/docs/Apple_T2_Security_Chip_Overview.pdf

[15] Apple Inc. 2021. Apple unleashes M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/

[16] Wei Jiang, Liang Wen, Jinyu Zhan, and Ke Jiang. 2020. Design optimization of confidentiality-critical cyber physical systems with fault detection. *Journal of Systems Architecture* 107 (2020), 101739.

[17] Chong Hee Kim. 2011. Improved differential fault analysis on AES key schedule. *IEEE transactions on information forensics and security* 7, 1 (2011), 41–50.

[18] Ana Lasheras, Ramon Canal, Eva Rodríguez, and Luca Cassano. 2020. Lightweight Protection of Cryptographic Hardware Accelerators against Differential Fault Analysis. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, 1–6.

[19] Stefan Mangard. 2002. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In *International Conference on Information Security and Cryptology*. Springer, 343–358.

[20] Hassen Mestiri, Noura Benhadjyoussef, and Mohsen Machhout. 2019. Fault Attacks Resistant AES Hardware Implementation. In *2019 IEEE International Conference on Design & Test of Integrated Micro & Nano-Systems (DTS)*. IEEE, 1–6.

[21] Amir Moradi, Mohammad T Manzuri Shalmani, and Mahmoud Salmasizadeh. 2006. A generalized method of differential fault attack against AES cryptosystem. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 91–100.

[22] Mehran Mozaffari-Kermani and Arash Reyhani-Masoleh. 2008. A lightweight concurrent fault detection scheme for the AES S-boxes using normal basis. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 113–129.

[23] Exhaustive Search on AES S-box. 2021. https://github.com/yadi14/esearch.

[24] OpenCores. 2002. AES128. https://opencores.org/projects/aes_crypto_core

[25] OpenCores. 2010. high throughput and low area aes core. https://opencores.org/projects/aes_highthroughput_lowarea

[26] OpenCores. 2013. AES encryption all keylength. https://opencores.org/projects/aes_all_keylength

[27] OpenCores. 2019. AES-VHDL. https://opencores.org/projects/aes

[28] OpenCores. 2021. OpenCores. https://opencores.org/

[29] Trevor E Pogue and Nicola Nicolici. 2019. Incremental Fault Analysis: Relaxing the Fault Model of Differential Fault Attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 3 (2019), 750–763.

[30] Shahed E Quadir, Junlin Chen, Domenic Forte, Navid Asadizanjani, Sina Shahbazmohamadi, Lei Wang, John Chandy, and Mark Tehranipoor. 2016. A survey on chip to system reverse engineering. *ACM journal on emerging technologies in computing systems (JETC)* 13, 1 (2016), 1–34.

[31] M Tanjidur Rahman, Shahin Tajik, M Sazadur Rahman, Mark Tehranipoor, and Navid Asadizanjani. 2020. The key is left under the mat: On the inappropriate security assumption of logic locking schemes. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 262–272.

[32] Vincent Rijmen and Joan Daemen. 2001. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology* (2001), 19–22.

[33] Saeideh Sheikhpour, Seok-Bum Ko, and Ali Mahani. 2021. A low cost fault-attack resilient AES for IoT applications. *Microelectronics Reliability* 123 (2021), 114202.

[34] Arvind Singh, Monodeep Kar, Sanu K Mathew, Anand Rajan, Vivek De, and Saibal Mukhopadhyay. 2018. Improved Power/EM Side-Channel Attack Resistance of 128-Bit AES Engines With Random Fast Voltage Dithering. *IEEE Journal of Solid-State Circuits* 54, 2 (2018), 569–583.

[35] Randy Torrance and Dick James. 2009. The State-of-the-Art in IC Reverse Engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*. 363–381.

[36] Unit42. 2020. 2020 Unit 42 IoT Threat Report. https://unit42.paloaltonetworks.com/iot-threat-report-2020/