IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 3, pp. 693–716. DOI:10.46586/tches.v2025.i3.693-716

Practical Opcode-based Fault Attack on AES-NI

Xue Gong[1], Xin Zhang[2], Qianmei Wu[1], Fan Zhang[1*], Junge Xu[1], Qingni Shen[2] and Zhi Zhang[3]

[1] School of Cyber Science and Technology, College of Computer Science and Technology, Zhejiang University, Hangzhou, 310027, China, {xuegong;qianmei;fanzhang;jungexu}@zju.edu.cn
[2] School of Software and Microelectronics, Peking University, Beijing 100871, China, zhangxin00@stu.pku.edu.cn,qingnishen@ss.pku.edu.cn
[3] Department of Computer Science and Software Engineering, University of Western Australia, Perth, WA 6009, Australia, zzhangphd@gmail.com

Abstract. AES New Instructions (AES-NI) is a set of hardware instructions introduced by Intel to accelerate AES encryption and decryption, significantly improving efficiency across various cryptographic applications. While AES-NI effectively mitigates certain side-channel attacks, its resilience against faults induced by active or malicious fault injection remains unclear.

In this paper, we conduct a comprehensive security analysis of AES-NI. By analyzing the opcodes of AES-NI, we identify six pairs of instructions with only a single-bit difference, making them susceptible to bit-flip-type attacks. This vulnerability allows attackers to recover AES keys in both Electronic Codebook (ECB) and Cipher Block Chaining (CBC) modes. We introduce a novel *Opcode-based Fault Analysis* (OFA) method, employing Gaussian elimination to reduce the search space of the last round key. In particular, with one pair of correct and faulty ciphertexts, OFA can reduce the key search space to 2^{32} for a 128-bit key length. To further reduce the key space for AES-192 and AES-256, we propose the *Enhanced Opcode-based Fault Analysis* (EOFA), which, compared to exhaustive search, reduces the key space by factors of 2^{160} and 2^{192}, respectively.

Finally, we demonstrate the feasibility of our findings by conducting physical end-to-end attacks. Specifically, Rowhammer is leveraged to flip vulnerable opcodes and OFA as well as EOFA techniques are applied to recover secret keys from AES implementations. Our experimental results for AES-ECB-128, AES-ECB-192, and AES-CBC-128 demonstrate that key recovery can be efficiently achieved within 1.03 to 1.36 hours, varying with the cipher. This work highlights a critical vulnerability in AES-NI and outlines a new and novel pathway for fault-based attacks against modern cryptographic implementations.

Keywords: Rowhammer · AES-NI · Fault Analysis · Fault Attack · OFA · EOFA

1 Introduction

In recent years, a number of cryptographic applications have been proposed to achieve various security goals, such as key exchange, signature, authentication and encryption [DH76, BBCT22, ABC+24]. The security analysis of these cryptographic applications can be categorized into two types: algorithm-oriented analysis and implementation-oriented analysis. Particularly, even when cryptographic algorithms are theoretically secure, their security guarantee can still be challenged after they are implemented on a certain hardware. Hence, a number of implementation-oriented analysis is imperative for ensuring the security of cryptographic applications [ZLZ+18, ZZJ+20, ZHF+23, HAZ+24].

*The corresponding author.

Received: 2025-01-15 Accepted: 2025-03-15 Published: 2025-06-05

Fault Attacks (FAs) [GZZ+25, CVM+21, LTH22, MDT+23, FKK+22] are a type of implementation-oriented attacks, which consists of two primary steps, i.e., fault injection and fault analysis. While the original FA relies on sharp voltage fluctuations [MOG+20, CVM+21, LTH22] or physical proximity [JT12] to inject the faults into hardware, recent work has removed the two requirements by applying Rowhammer-based faults [MDT+23, FKK+22], which allows a remote attacker to trigger hardware faults in the memory.

The Advanced Encryption Standard (AES) algorithm [RD01], established by NIST in 2001, is now widely deployed in various applications. In Transport Layer Security (TLS), AES encrypts data to be exchanged between clients and servers, ensuring its confidentiality and integrity [Res18]. Besides, AES is also utilized by many Virtual Private Networks (VPNs) to safeguard network traffic from eavesdropping and tampering. Various file encryption tools, including VeraCrypt and DiskCryptor, rely on AES to encrypt files and volumes, providing disk encryption functionality to safeguard sensitive data stored on computer systems. Given the pivotal role of AES in modern cryptography, analyzing the security of its implementation has become increasingly important.

AES New Instructions Set (AES-NI) is a set of instructions introduced with the 2010 Intel processor line, offering fast and secure encryption and decryption using AES. Due to AES's widespread adoption as the predominant block cipher in various protocols and applications, AES-NI is extensively utilized across diverse scenarios.

AES-NI includes six instructions for full hardware support for AES, which are listed in Table 1. Specifically, four instructions (AESENC, AESENCLAST, AESDEC, and AESDELAST) support AES encryption and decryption, while the remaining two (AESIMC and AESKEYGENASSIST) are designed for AES key expansion. Grounded on this, AES-NI provides a notable boost in performance compared to other pure-software implementations. For an n-round encryption process, AES-NI repeats the AESENC instruction $n-1$ times, followed by the AESENCLAST instruction in the final stage.

Table 1: Instructions in AES-NI.

Instruction	Description
AESENC	One Round of an AES Encryption
AESENCLAST	Last Round of an AES Encryption
AESDEC	One Round of an AES Decryption
AESDECLAST	Last Round of an AES Decryption
AESIMC	AES InvMixColumn Transformation
AESKEYGENASSIST	AES Round Key Generation Assist

Intel has stated in the white paper [Gue10] that AES-NI is designed to mitigate all of the known timing and cache side-channel leakages of sensitive data. Its latency remains consistent regardless of data input, and since all computations occur internally within the hardware, there is no need for lookup tables. Thus, when AES instructions are appropriately employed, both encryption/decryption and key expansion processes exhibit data-independent timing and solely involve data-independent memory access. Consequently, AES instructions facilitate the development of high-performance AES-dependent software that is concurrently shielded against known software side-channel attacks. Therefore, AES-NI is regarded as a main-stream countermeasure to mitigate presently known timing and cache attacks on AES [MKS12].

Although AES-NI can mitigate certain types of side-channel attacks, its effectiveness in countering actively and maliciously induced faults remains uncertain. To date, the only fault analysis targeting AES-NI is presented in [TMA11], with the necessary fault injections implemented in [MOG+20, CVM+21]. However, both voltage-based fault injection techniques have been mitigated through a microcode update. This research is primarily motivated by the absence of practical fault attacks against AES-NI.

1.1 Our Contribution

In this paper, we conduct a comprehensive security analysis on AES implementation leveraging AES-NI instructions. Interestingly, our findings reveal that the instructions in AES-NI are vulnerable to bitflip-type attacks, posing significant threats to the security of AES implementations. In detail, our contributions are as follows:

Disclosing vulnerabilities in AES-NI: a novel method. By analyzing the opcodes of AES-NI, we identified six pairs of instructions, wherein the opcodes of two instructions within each pair differ by merely 1 bit, making them potentially vulnerable to bitflip-type fault attacks. Through a thorough analysis, we confirmed that the first pair, comprising AESENC and AESENCLAST, enables the recovery of the AES key in Electronic Codebook (ECB) mode. Likewise, the second pair, AESDEC and AESDECLAST, facilitates key retrieval in Cipher Block Chaining (CBC) mode. To the best of our knowledge, this is the first paper to expose the code flip vulnerability of AES-NI.

Proposing the *Opcode-based Fault Analysis* (OFA): an efficient analysis. For AES-128, we successfully utilize Gaussian elimination to solve equations over finite fields in the fault analysis process, effectively reducing the solution space of the final round key to 2^{32}. In this case, the master key can be derived from the final round key by reversing the key expansion scheme. This optimization accelerates the analysis by a factor of approximately 2^{96} compared to brute-force search. Leveraging AES-NI acceleration, this process can be completed within 2 hours.

Presenting *Enhanced Opcode-based Fault Analysis* (EOFA) for key lengths of 192 and 256: an enhanced solution. For AES-192 and AES-256, the final round keys are 128 bits, while the master keys are 192 and 256 bits, respectively. Even if the final round keys are known, this information is insufficient to reconstruct the master key. To address this challenge, we propose an *Enhanced Opcode-based Fault Analysis* (EOFA) approach, which augments the original OFA with additional fault injections to derive new sets of correct and faulty ciphertext pairs. This refinement reduces the key space for AES-192 to 2^{32} and for AES-256 to 2^{64}, respectively.

Conducting End-to-end Attacks: a real-world application. We leverage Rowhammer to flip the identified vulnerable opcodes and exploit OFA and EOFA to achieve the secret-key recovery. We have successfully performed end-to-end attacks against four ciphers implemented using AES-NI: AES-ECB-128, AES-ECB-192, AES-CBC-128 (with a known IV), and AES-CBC-128 (with an unknown IV). With a single pair of resulted correct and faulty ciphertexts for each cipher, we perform post-fault analysis to recover the secret key. This analysis is efficient, with key recovery taking between 1.03 and 1.36 hours, depending on the cipher used.

1.2 Organization

The paper is organized as follows: Section 2 provides the background, notations, and definitions. Section 3 reviews related work. Section 4 introduces OFA on AES-NI, while Section 5 presents the enhanced one called EOFA. Subsequently, OFA on AES-CBC mode is discussed in Section 6. Section 7 showcases end-to-end attacks on four ciphers, and Section 8 provides a discussion. Finally, Section 9 concludes the paper.

2 Background

2.1 Notations

We denote the finite field of order q as $\mathbb{F}_q$. Given that AES is the primary focus of this paper, we set $q = 2^8$, and consequently, $\mathbb{F}_{2^8}$ is referred to simply as $\mathbb{F}$ throughout this paper. Let the symbols "$\oplus$" and "$\cdot$" denote the addition and multiplication over $\mathbb{F}$, respectively.

The specific numbers over $\mathbb{F}$ are represented in hexadecimal notation (e.g., "02" for "2" and "$d4$" for "211"). Note that parentheses (e.g., $\{02\}$) will be applied sometimes to distinguish these numbers in equations.

In the context of AES, the plaintext and ciphertext are denoted as P and C, respectively. Corresponding to C, the faulty ones are represented as $C^{'}$ or $C^{''}$. $\mathbb{E}_K$ and $\mathbb{D}_K$ represents the encryption and decryption process, respectively. During the key schedule of AES, K_s denotes the master key, while the r-th round key is represented as K_r. Let the symbol N_k denote the length of the key divided by 32. N_r denotes the number of AES rounds. As AES is executed, X, Y, Z and W are utilized to denote the intermediate states, which can be represented in matrix form, with X as an example shown below:

$$X = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}.$$

Note that the element $x_{i,j} \in \mathbb{F}$ with $0 \leq i, j \leq 3$ denotes one byte for AES. Accordingly, $X[c] = \begin{bmatrix} x_{0,c} \\ x_{1,c} \\ x_{2,c} \\ x_{3,c} \end{bmatrix}$ is a 32-bit word for $0 \leq c \leq 3$ and X is a 128-bit block (or saying a matrix over $\mathbb{F}^{4\times4}$). For two matrices X and Y over $\mathbb{F}$, we denote their product as $X \times Y$ (or in short XY).

2.2 AES

During the execution of AES, data is encrypted and decrypted within 128-bit blocks using key length of 128, 192, or 256 bits. The specific number of AES rounds varies with the key length. Specifically, 10 rounds for a 128-bit key, 12 rounds for a 192-bit key and 14 rounds for a 256-bit key. Typically, one AES round consists of four byte-oriented transformations applied to the state: **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey**. However, in the final AES round, **MixColumns** is omitted, leaving only **SubBytes**, **ShiftRows**, and **AddRoundKey**.

Since our proposed attack approach is largely related to **MixColumns** operation of AES, we provide a detailed description of **MixColumns** process in this sub-section to make this paper to be self-contained. **MixColumns** transforms the state by multiplying each of its four columns with a fixed matrix A, which is presented below:

$$A = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Thus, the **MixColumns** function can be represented as:

$$\begin{bmatrix} s^{'}_{0,c} \\ s^{'}_{1,c} \\ s^{'}_{2,c} \\ s^{'}_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}, \text{ for } 0 \leq c \leq 3.$$

For each element, we have:

$$\begin{aligned} s'_{0,c} &= \{02\} \cdot s_{0,c} \oplus \{03\} \cdot s_{1,c} \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus \{02\} \cdot s_{1,c} \oplus \{03\} \cdot s_{2,c} \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus \{02\} \cdot s_{2,c} \oplus \{03\} \cdot s_{3,c} \\ s'_{3,c} &= \{03\} \cdot s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus \{02\} \cdot s_{3,c} \end{aligned}, \text{ for } 0 \leq c \leq 3.$$

During the computation of **MixColumns**, XTIME(b) is defined to compute $b \cdot \{02\}$ for $b \in \mathbb{F}$. Hence, the inversion of XTIME(b) $= c$ is represented as $b = \{02\}^{-1} \cdot c$, or $b = \text{XTIME}^{-1}(c)$ for $b, c \in \mathbb{F}$. For example, as $\{57\} \cdot \{02\} = \text{XTIME}(\{57\}) = \{ae\}$, we have $\{02\}^{-1} \cdot \{ae\} = \{57\}$. In the remainder of this paper, the multiplication over $\mathbb{F}$ is represented without the use of "$\cdot$" and parentheses for simplicity. For example, $\{02\} \cdot x_{0,0} \oplus \{03\} \cdot x_{1,0}$ is simplified by $2x_{0,0} \oplus 3x_{1,0}$. Accordingly, 2^{-1} is the simple form of $\{2\}^{-1}$ and thus $2^{-1}(\cdot)$ represents $\text{XTIME}^{-1}(\cdot)$.

3 Related Works

3.1 Differential Fault Analysis

The concept of fault attacks was initially introduced by Boneh *et al.* in 1996 with their work on RSA-CRT [BDL97]. Subsequently, Biham and Shamir proposed the differential fault analysis (DFA) on DES [BS97], amalgamating fault attacks with differential cryptanalysis. Since its inception, DFA has played a crucial role in compromising various block ciphers. DFA exploits the difference between correct and faulty ciphertexts for a fixed input. In this form of fault attack, the adversary provides an initial message as input to the cipher, which then traverses the oracle to produce the original ciphertext. Meanwhile, at certain intermediate stages, the attacker introduces faults by altering bits at positions such as bit, nibble, or byte in the state, resulting in a faulty ciphertext at the output. Assuming there are n rounds in the cipher, and the faulty ciphertext $C^{'}$ is generated by injecting a fault at the r-th round, then the entire cipher can be regarded as a reduced cipher after the fault, where the difference spreads in the state through only $n - r$ rounds. Leveraging these pairs of faulty and original ciphertexts $(C, C^{'})$, the attacker endeavors to uncover secret information by employing classical cryptanalytic techniques.

3.2 Fault Attacks on AES-NI

In 2020, Murdock *et al.* demonstrated that Plundervolt [MOG+20], a software-based undervolting (also known as glitching) fault injection method, can induce a bit-flip in the leftmost two bytes of the AES-NI round function's output. By exploiting this fault in round 8 and employing the Differential Fault Analysis (DFA) technique described by Tunstall *et al.* in [TMA11], this attack can recover the 128-bit AES key using a pair of correct and faulty ciphertexts derived from the same plaintext, with an average computational complexity of $2^{32} + 256$ encryptions. In 2021, Chen *et al.* utilized VoltPillager [CVM+21], a hardware-based fault injection method, to reproduce the Plundervolt attack on AES-NI. However, Plundervolt has been mitigated by Intel, while VoltPillager requires physical access to SGX and is beyond the scope of SGX threat model. Table 2 compares our Opcode Fault Analysis approach with Plundervolt and VoltPillager. While both Plundervolt and VoltPillager induce bit-flip in the output state of AES-NI, our OFA method employs a different fault model where opcodes are faulted. In terms of fault analysis, Plundervolt and VoltPillager utilize DFA in [TMA11], whereas our approach utilizes a novel opcode-based fault analysis framework. The computational complexity of our fault analysis process is comparable to that of DFA. Regarding fault injection success rates, we provide the Rowhammer fault injection success probability for our OFA method. For Plundervolt and

VoltPillager, the probability of injecting a fault into any particular AES round is equal. However, since fault analysis specifically requires faults in the 8th round, their effective fault injection success rate is approximately 10%, assuming perfect precision in voltage fault injection.

Table 2: Comparison of Different Fault Attacks on AES-NI.

	Fault Model	Time Complexity	Success Rate
OFA	single bit flip on opcodes	2^{32}	6.2% (empirical)
Plundervolt	single bit flip on intermediate state	$2^{32} + 256$	10% (estimated)
VoltPillager	single bit flip on intermediate state	$2^{32} + 256$	10% (estimated)

3.3 Rowhammer

In our attack, we induce bit flips through *Rowhammer*, a software-triggered hardware fault that flips bits in Dynamic Random Access Memory (DRAM). This phenomenon was first introduced in 2014 by Kim *et al.* [KDK+14].

Modern DRAM designs have reduced supply voltages, leading to smaller charges in the capacitors that store individual bits. Electromagnetic interference generated during the access of neighboring bits can affect these capacitors. Kim *et al.* [KDK+14] identified voltage fluctuations on an internal wire known as the wordline as the primary cause of DRAM disturbance issues.

Specifically, DRAM is organized as a two-dimensional array of cells, with each row of cells controlled by a dedicated wordline. Accessing a cell in a specific row can enable the corresponding wordline by raising its voltage. Repeated activations of the same row cause the wordline to toggle on and off, generating voltage variations that disturb adjacent rows. This disturbance accelerates charge leakage in some cells, potentially leading to errors if the charge dissipates before being refreshed. The repeatedly activated row is called the "aggressor row", while the row experiencing induced bit flips is referred to as "victim row" [KDK+14, JWS+24]. Consequently, physical pages mapped to victim rows or aggressor rows are termed victim pages and aggressor pages, respectively. The term "hammer patterns" describes the number of aggressor rows required to induce bit flips in a victim row [ZHC+21].

4 Opcode-based Fault Attack on AES-NI

4.1 Threat Model

Similar to existing Rowhammer attacks that recover secret keys from cryptographic implementations [WTM+20, MIS20, FKK+22, MDT+23, AWK+25], we make the following assumptions. The attacker is able to initiate an arbitrary unprivileged user process without requiring root privileges, and thus lacks access to the mapping between virtual and physical memory. The victim executes the AES encryption and decryption program implemented with AES-NI, which can be queried by the attacker. The attacker is aware of the specific cryptographic scheme, including the bit length of the secret key. However, it is assumed that the kernel executing the victim process is secure and effectively enforces isolation between the attacker and the victim. Note that it is not required for the attacker and victim to share the same kernel, as long as the DRAM modules are shared and vulnerable to Rowhammer-induced bit flips.

4.2 Potential Fault Injection Locations

As highlighted in red in Table 3, six pairs of instructions in AES-NI are identified, where the opcodes of the two instructions in each pair differ by only one bit. Due to this minimal difference, a one-bit-flip fault attack can easily alter an instruction to another within the

Table 3: Difference in Opcodes of AES-NI.

Pair	Instruction	Opcodes	Difference in Opcodes
1	AESENC	0x66,0x0F,0x38,0xDC	* * * * * * * 1100
	AESENCLAST	0x66,0x0F,0x38,0xDD	* * * * * * * 1101
2	AESDEC	0x66,0x0F,0x38,0xDE	* * * * * * * 1110
	AESDECLAST	0x66,0x0F,0x38,0xDF	* * * * * * * 1111
3	AESENC	0x66,0x0F,0x38,0xDC	* * * * * * * 1100
	AESDEC	0x66,0x0F,0x38,0xDE	* * * * * * * 1110
4	AESENCLAST	0x66,0x0F,0x38,0xDD	* * * * * * * 1101
	AESDECLAST	0x66,0x0F,0x38,0xDF	* * * * * * * 1111
5	AESIMC	0x66,0x0F,0x38,0xDB	* * * * * * * 1011
	AESDECLAST	0x66,0x0F,0x38,0xDF	* * * * * * * 1111
6	AESDECLAST	0x66,0x0F,0x38,0xDF	* * * * * 1000 * *
	AESKEYGENASSIST	0x66,0x0F,0x3A,0xDF	* * * * * 1010 * *

pair, resulting in a faulty instruction flow that can be leveraged by adversaries to retrieve the secret key. Our research is motivated by such design of opcodes in AES-NI and the associated vulnerability it brings. For instance, as detailed in the subsequent sections, the first pair (AESENC and AESENCLAST) can be exploited to recover the key of AES in ECB mode, while the second pair (AESDEC and AESDECLAST) can be furtehr leveraged to recover the key of AES CBC mode.

4.3 Opcode-based Fault Analysis

Assuming that AES has N rounds, recall that for the first $N-1$ rounds, each AES round consists of four operations, namely **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey**. While the last round excludes the **MixColumns** operation, involving only three operations. Using AES-NI, the AESENC instruction is used to execute the first $N-1$ rounds, performing all four operations for one AES round, while AESENCLAST is designed for the last AES round, executing only three operations. As highlighted in Table 3, AESENC and AESENCLAST differ by only one bit. Hence, if a one-bit-flip fault is injected, converting AESENCLAST to AESENC, the last round of AES will perform an additional **MixColumns** operation, resulting in faulty encryption that can be exploited by adversaries. On this basis, we propose Opcode-based Fault Attack (OFA), which exploits the one-bit distinction between opcode pairs to retrieve secret keys.

As illustrated in Figure 1, OFA generally performs one correct AES encryption (as shown in the left flow of Figure 1) and one faulty AES encryption (as shown in the right flow of Figure 1), subsequently narrowing the search space for the last-round key K_N to 2^{32}. Below we outline the steps for OFA one by one:

Step 1: Correct Encryption. The attacker uses the device to encrypt a random plaintext P, which is 128-bit. The plaintext is known to the attacker but not deliberately constructed. The attacker collects the ciphertext, which is denoted as C.

Step 2: Fault Injection. The attacker injects a fault by Rowhammer. The fault can cause a bit flip in the opcode of AESENCLAST. Thus, the AESENCLAST is altered to AESENC, causing the last AES round to perform an additional **MixColumns**.

Step 3: Faulty Encryption. The attacker uses the faulty algorithm to encrypt the plaintext P again. At this time, the attacker obtains the faulty ciphertext $C^{'}$.

Step 4: Fault Analysis/Key Recovery. The attacker performs the offline analysis by doing Opcode-based Fault Analysis with $(P, C, C^{'})$ to recover the key.

Now, we provide a more detailed description of OFA. We denote the 128-bit state after **ShiftRows** in the last round as X. The last round of **AddRoundKey** operation can be represented as:

$$X \oplus K_N = C \tag{1}$$

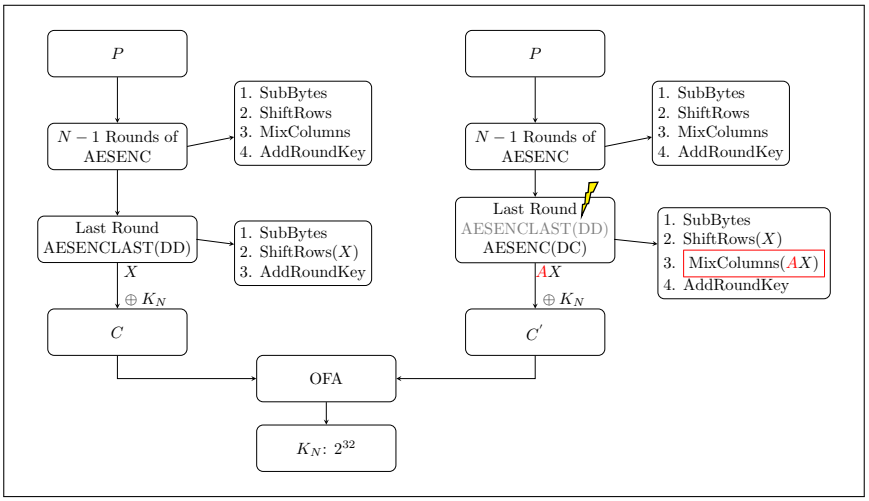


Figure 1: The overview of OFA.

By injecting faults into the device via Rowhammer, the last round operation AESENCLAST is changed to AESENC. After the fault injection, the attacker uses the faulty device to encrypt the same plaintext again. In the faulty encryption, state X goes through an additional **MixColumns** in the last round. The **MixColumns** function is equivalent to multiplying a matrix A to the left side of X. After the **MixColumns** and the **AddRoundKey** operations, the attacker gets the faulty ciphertext. This process can be represented as:

$$AX \oplus K_N = C^{'}. \tag{2}$$

By applying the XOR operation to Equation (1) and Equation (2), we obtain:

$$X \oplus AX = C \oplus C^{'}. \tag{3}$$

Equation (3) is equivalent to:

$$(I \oplus A)X = C \oplus C^{'}. \tag{4}$$

Note that in Equation (4), the coefficient matrix A is public and accessible to adversaries, while matrix I represents a fourth-order identity matrix. According to the threat model outlined in Section 4.1, both the correct ciphertext C and the faulty ciphertext $C^{'}$ are also known to adversaries. Consequently, the unknown matrix X can be determined by solving Equation (4). Subsequently, we are able to recover the last round key K_N by substituting X into Equation (1), as shown below:

$$K_N = X \oplus C. \tag{5}$$

Therefore, the key to recovering K_N lies in solving Equation (4) to obtain X, which will be detailed in the following sub-section.

4.3.1 Equation Solving

In Equation (4), recall that A is a public matrix corresponding to **MixColumns** operation and I is the fourth-order identity matrix. Hence, we have:

$$A \oplus I = \begin{bmatrix} 03 & 03 & 01 & 01 \\ 01 & 03 & 03 & 01 \\ 01 & 01 & 03 & 03 \\ 03 & 01 & 01 & 03 \end{bmatrix}. \tag{6}$$

Let us denote $C \oplus C' = Y$, and $Y = [Y[0], Y[1], Y[2], Y[3]]$ with $Y[c] = \begin{bmatrix} y_{0,c} \\ y_{1,c} \\ y_{2,c} \\ y_{3,c} \end{bmatrix}$ for $0 \leq c \leq 3$. Recall that $X = [X[0], X[1], X[2], X[3]]$ with $X[c] = \begin{bmatrix} x_{0,c} \\ x_{1,c} \\ x_{2,c} \\ x_{3,c} \end{bmatrix}$ for $0 \leq c \leq 3$. Applying Equation (4), we have:

$$\begin{bmatrix} 03 & 03 & 01 & 01 \\ 01 & 03 & 03 & 01 \\ 01 & 01 & 03 & 03 \\ 03 & 01 & 01 & 03 \end{bmatrix} \begin{bmatrix} x_{0,c} \\ x_{1,c} \\ x_{2,c} \\ x_{3,c} \end{bmatrix} = \begin{bmatrix} y_{0,c} \\ y_{1,c} \\ y_{2,c} \\ y_{3,c} \end{bmatrix}, \text{ for } 0 \leq c \leq 3. \tag{7}$$

Hence, for each element, we can obtain:

$$\begin{aligned} 3x_{0,c} \oplus 3x_{1,c} \oplus x_{2,c} \oplus x_{3,c} &= y_{0,c} \\ x_{0,c} \oplus 3x_{1,c} \oplus 3x_{2,c} \oplus x_{3,c} &= y_{1,c} \\ x_{0,c} \oplus x_{1,c} \oplus 3x_{2,c} \oplus 3x_{3,c} &= y_{2,c} \\ 3x_{0,c} \oplus x_{1,c} \oplus x_{2,c} \oplus 3x_{3,c} &= y_{3,c} \end{aligned}, \text{ for } 0 \leq c \leq 3. \tag{8}$$

To solve the equation system, Gaussian Elimination over $\mathbb{F}$ can be employed, which uses elementary row operations to create zeros below the pivot element in the first column. After Gaussian Elimination, the corresponding equation system is:

$$\begin{aligned} 3x_{0,c} \oplus 3x_{1,c} \oplus x_{2,c} \oplus x_{3,c} &= y_{0,c} \\ 2x_{0,c} \oplus 2x_{2,c} &= y_{1,c} \oplus y_{0,c}, \\ 2x_{1,c} \oplus 2x_{3,c} &= y_{2,c} \oplus y_{1,c} \end{aligned} \text{ for } 0 \leq c \leq 3. \tag{9}$$

To solve the above equation system, we need an extra function to eliminate the coefficient to "01", which is the inversion of XTIME . For efficiency, all the results of $\text{XTIME}^{-1}(b)$ with $b \in \mathbb{F}$ can be computed in advance and stored in a table before the equation solving. Note that the linear system consists of four variables but only three equations, resulting in the presence of a free variable. We designate $x_{0,c}$ as the free variable. Thus, the solution to Equation (9) is:

$$\begin{aligned} x_{1,c} &= 2^{-1}[3x_{0,c} \oplus 2^{-1}(2x_{0,c} \oplus y_{1,c} \oplus y_{0,c}) \oplus 2^{-1}(y_{2,c} \oplus y_{1,c}) \oplus y_{0,c}] \\ x_{2,c} &= 2^{-1}(2x_{0,c} \oplus y_{1,c} \oplus y_{0,c}) \\ x_{3,c} &= 2^{-1}[3x_{0,c} \oplus 2^{-1}(2x_{0,c} \oplus y_{1,c} \oplus y_{0,c}) \oplus 2^{-1}(y_{2,c} \oplus y_{1,c}) \oplus y_{0,c}] \oplus 2^{-1}(y_{2,c} \oplus y_{1,c}). \end{aligned} \tag{10}$$

Note that Y is known and there are a total of 2^{32} possible values for $x_{0,c}$, each of which corresponds to a distinct set of solutions for the system of equations. Each set of solutions, in turn, corresponds to a possible key. Thus, by solving the equations, the space of final round keys can be reduced to a size of 2^{32}.

4.3.2 From the Last Round Key to the Master Key

By applying Gaussian Elimination, the key space for the last round key in AES has been reduced to 2^{32}. In this subsection, we outline the process for deducing the master key K_s from the reduced key space of the last round key K_N. Since the key length of master key K_s varies across AES-128, AES-196 and AES-256, leading to different determination processes, we will address each case separately.

Considering AES-128, recall that the key length is 128 bits. Our goal is to deduce each round key, with particular emphasis on the master key K_s. Algorithm 1 illustrates the inversion of key expansion in [DR99] for $N_k \leq 6$. Note that $W[i]$ is a 32-bit word to represent the round key. For example, for AES-128, the last round key is $[W[40], W[41], W[42], W[43]]$. In Algorithm 1, **RotByte** is a cyclic permutation and $\mathbf{Rcon}[i/N_k]$ is the specific round constant [DR99]. For each candidate of the last round key, a corresponding possible master key is derived using Algorithm 1. Recall that in our fault model, plaintext is randomly selected but known to the attacker. Consequently, the attacker can utilize each possible master key to decrypt the ciphertext and verify whether it matches the plaintext. Ultimately, the actual master key K_s can be determined by successfully matching the decrypted ciphertext with the known plaintext.

Algorithm 1 Inversion of Key Expansion for $N_k \leq 6$

Input: byte $K_N[4 * 4]$: the last round key
Output: word $W[4 * (N_r + 1)]$: the full round key
1: **for** $i = 0; i < 4; i++$ **do**
2: $W[4 * N_r + i] = (K_r[4 * i], K_r[4 * i + 1], K_r[4 * i + 2], K_r[4 * i + 3])$
3: **end for**
4: **for** $i = 4 * N_r + 3; i \geq N_k; i--$ **do**
5: $temp = W[i - 1]$
6: **if** $i \ mod \ N_k == 0$ **then**
7: $temp = \mathbf{SubBytes}(\mathbf{RotByte}(temp)) \oplus \mathbf{Rcon}[i/N_k]$
8: **end if**
9: $W[i - N_k] = W[i] \oplus temp$
10: **end for**

Considering AES-192, the master key is 192 bits while the last round key remains 128 bits. With the last round key as the input for Algorithm 1, we attempt to deduce the master key. Specifically, with each candidate value obtained from the previous step of solving equations, the last round key $[W[48], W[49], W[50], W[51]]$ is determined. Recall that the penultimate round key is $[W[44], W[45], W[46], W[47]]$. When applying Algorithm 1 to get the penultimate round key, we have

$$W[45] = W[51 - 6] = W[51] \oplus W[50],$$

$$W[44] = W[50 - 6] = W[50] \oplus W[49].$$

However, $W[47]$ and $W[46]$ are unknown and thus cannot be determined according to Algorithm 1. This implies that an extra 2^{64} trials is required to recover the master key.

Considering AES-256, the master key is 256 bits while the last round key remains 128 bits as well. For AES-256, $N_k = 8$, the last round key is $[W[56], W[57], W[58], W[59]]$. When applying Algorithm 2, in the first loop, we get $W[51] = W[59] \oplus W[58]$. However, the penultimate round key $[W[52], W[53], W[54], W[55]]$ remains unknown and cannot be deduced from Algorithm 2. The attacker must perform an additional 2^{128} trials of the penultimate round key to obtain the master key.

In conclusion, the master key of AES-128 can be obtained by searching a key space of 2^{32}, which is a manageable range for modern computers. However, for AES-192 and AES-256, the key search spaces are significantly larger, up to 2^{96} and 2^{160}, respectively, making it impractical to complete the search on a personal computer.

5 Enhanced Opcode-based Fault Analysis

In this section, we propose *Enhanced Opcode-based Fault Analysis* (EOFA) to address the challenges of breaking AES-192 and AES-256. As is analyzed above, the search space for

Algorithm 2 Inversion of Key Expansion for $N_k > 6$

Input: byte $K_N[4*4]$
Output: word $W[4*(N_r+1)]$: the full round key
1: **for** $i = 0$; $i < 4$; $i++$ **do**
2: $W[4*N_r+i] = (K_r[4*i], K_r[4*i+1], K_r[4*i+2], K_r[4*i+3])$
3: **end for**
4: **for** $i = 4*N_r+3$; $i \geq N_k$; $i--$ **do**
5: $temp = W[i-1]$
6: **if** i *mod* $N_k == 0$ **then**
7: $temp = \mathbf{SubBytes}(\mathbf{RotByte}(temp)) \oplus \mathbf{Rcon}[i/N_k]$
8: **else**
9: **if** i *mod* $N_k == 4$ **then**
10: $temp = \mathbf{SubBytes}(temp)$
11: **end if**
12: **end if**
13: $W[i-N_k] = W[i] \oplus temp$
14: **end for**

AES-192 and AES-256 are impractical for an attacker with limited computing resources. Therefore, we propose a refined strategy to further reduce the key search space for AES-192 and AES-256. In this strategy, the attacker executes the aforementioned Rowhammer attack, collecting C and $C^{'}$, and applies OFA accordingly. Subsequently, a new fault is introduced in the penultimate round, changing the AESENC instruction to AESENCLAST. This fault either accumulates with the previous fault (Case 1) or can be injected upon device reboot (Case 2).

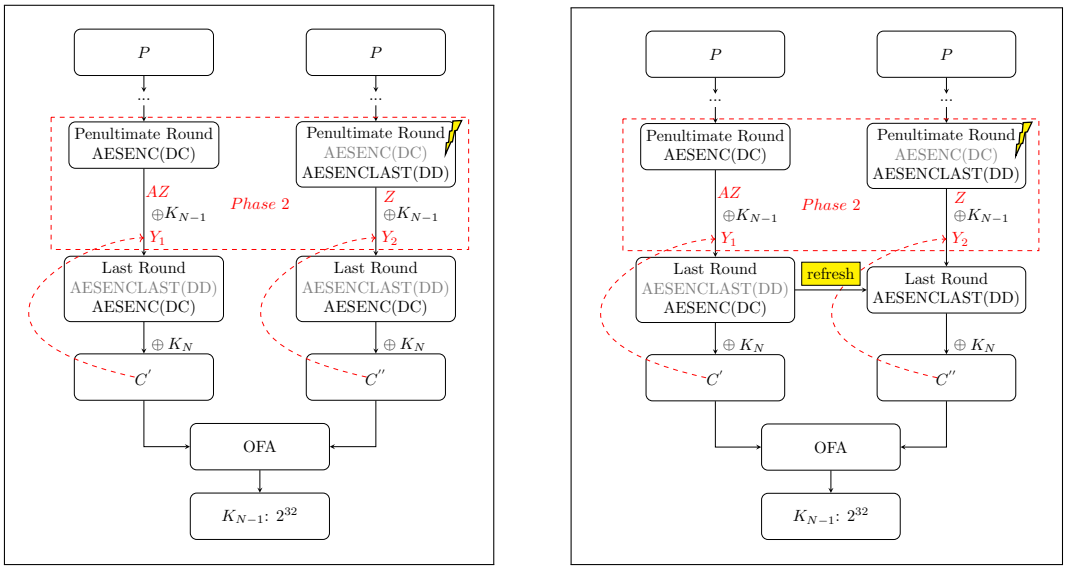
The strategy for further breaking AES-192 and AES-256 is outlined as follows:

Phase 1: The basic OFA. As shown in Figure 1, before fault injection, the attacker obtains the correct ciphertext C. A fault is injected into the AES-NI instruction AESENCLAST (0x66, 0x0F, 0x38, 0xDD), altering it to AESENC (0x66, 0x0F, 0x38, 0xDC). Then the attacker encrypts the same plaintext to produce the faulty ciphertext $C^{'}$. By applying OFA with $(C, C^{'})$, the attacker gets 2^{32} candidates for the last round key.

Phase 2: Fault injection in the penultimate round. As shown in Figure 2, in this phase, the attacker with the double-sided Rowhammer ability injects an extra fault into AES-NI, altering the opcode of the penultimate round encryption from AESENC (0x66, 0x0F, 0x38, 0xDC) to AESENCLAST (0x66, 0x0F, 0x38, 0xDD). The newly injected fault can accumulate with the fault from Phase 1, as shown in Figure 2a, or the new fault can be injected after the device reboots, as depicted in Figure 2b. After the fault injection, the attacker uses the faulty device to encrypt the same plaintext again and obtains a faulty ciphertext, $C^{''}$.

Phase 3: OFA on the penultimate round. In this phase, the attacker uses $C^{'}$ and $C^{''}$ to recover the penultimate round key K_{N-1} for AES-192 and AES-256. Recall that the attacker has 2^{32} candidates for the last round key K_N. With each candidate, the attacker can decrypt the last round. Regarding Case 1, as shown in Figure 2a, the decryption is the reversion of AESENC using K_N. While considering Case 2, as shown in Figure 2b, the decryption includes both the reversion of AESENC and AESENCLAST. After the decryption, the values of Y_1 and Y_2 are attained.

The attacker applies OFA in to (Y_1, Y_2). The input is the same plaintext, and the functions remain identical before the penultimate round. In the penultimate round, after the additional fault injection, AESENC is changed to AESENCLAST. Consequently, for Y_2, the **MixColumns** function is omitted. We denote the state after the **ShiftRows** in the penultimate round as Z. For Y_1, the **MixColumns** function operates as a matrix



(a) Case 1. **(b)** Case 2.

Figure 2: Phase 2: Fault injection on the penultimate round.

multiplication. Therefore, we have:

$$AZ \oplus K_{N-1} = Y_1 \tag{11}$$

$$Z \oplus K_{N-1} = Y_2 \tag{12}$$

By applying XOR operation on Equation (11) and Equation (12), we have:

$$(A \oplus I)Z = Y_1 \oplus Y_2 \tag{13}$$

By performing Opcode-based Fault Analysis (detailed in Section 4.3.1) with (Y_1, Y_2), we can obtain 2^{32} candidates for K_{N-1}. Consequently, the total search space for AES-192 and AES-256 is reduced to $2^{32} \times 2^{32} = 2^{64}$.

A Further Reduction for AES-192. In AES-192, for each candidate of the last round key, the key schedule reveals the 64 bits of the penultimate round key K_{N-1}. When solving Equation (13), the 32-bit free variable has a unique value. Thus, the search space for K_{N-1} is reduced to one, resulting in a total search space of 2^{32} for the AES-192 key.

Discussion on Further Reduction for AES-256. According to the key expansion scheme of AES-256, it is impossible to derive information about the penultimate round's key from the last round's key. When performing an EOFA, AES-256 cannot utilize the same method as AES-192 to further reduce the key search space. We attempted to inject faults into deeper rounds, causing the AESENC instruction in the third-to-last round to be replaced with AESENCLAST, thereby obtaining an additional set of correct and faulty ciphertext pairs. However, when employing OFA on the third-to-last round, it is necessary to decrypt the last two rounds using 2^{64} possible keys from the previous step. Consequently, the total key search space in this approach remains 2^{64}.

In summary, by applying EOFA, the key search space of AES-192 is reduced to 2^{32}, and the key search space for AES-256 is reduced to 2^{64}. Compared to an exhaustive search, EOFA reduces the key space by factors of 2^{160} and 2^{192}, respectively.

6 Extensions to AES-CBC Mode

In this Section, we discuss how to extend our fault attack to the CBC mode of AES, considering different assumptions about the attacker.

6.1 The CBC Mode

The characteristic of AES-CBC mode is that the encryption result of the previous ciphertext block is XORed with the current plaintext block prior to encryption, except for the first block, which uses an Initialization Vector (IV) instead. The purpose of this additional operation is to increase the complexity of the encryption process, ensuring that identical plaintext blocks do not produce the same ciphertext blocks, thereby enhancing the overall security of the encryption. The CBC mode is defined as follows:

CBC Encryption:

$$\begin{aligned} C_1 =& \mathbb{E}_K(P_1 \oplus IV); \\ C_j =& \mathbb{E}_K(P_j \oplus C_{j-1}), \text{ for } j = 2 \ldots n. \end{aligned} \tag{14}$$

CBC Decryption:

$$\begin{aligned} P_1 =& \mathbb{D}_K(C_1) \oplus IV; \\ P_j =& \mathbb{D}_K(C_j) \oplus C_{j-1}, \text{ for } j = 2 \ldots n. \end{aligned} \tag{15}$$

6.2 Key Recovery of CBC Mode

In our fault model, the attacker needs to collect both the correct and the faulty ciphertexts for the same input. As our attack requires only one pair of correct and faulty ciphertexts, the block size in CBC mode can be one (*i.e.,* 128 bits). In CBC mode, the first input block of the encryption process is $P \oplus IV$. However, as stated in Appendix C of NIST SP 800-38A [Dwo01], an IV must be generated for each execution of the encryption operation. This implies that after the attacker injects faults into the opcode and encrypts the same plaintext, the input of the encryption will change due to the generation of a new IV. The primary challenge in extending our attack to CBC mode lies in ensuring that the input to the encryption process remains consistent across two encryption attempts, despite the variation of IV.

If the attacker is assumed to be able to set IV to zero, then the first 128 bits block is just the same as in ECB mode. Hence, the attack can proceed as described in previous sections. However, in most cases, the attacker cannot control or predict the IV in the encryption process. Therefore, based on the attacker's assumptions, we provide two different methods to carry out the attack.

6.3 Attack on CBC Mode with Known IV

We assume in each encryption call, the attacker can generate and observe the value of IV before the encryption. The attack process is shown in Figure 3 and proceeds as follows:

Step 1: Correct Encryption. The attacker generates IV, which is denoted as IV_1. The attacker uses the device to encrypt a random plaintext P, which is 128-bit. As in ECB mode, the plaintext is known to the adversary but not deliberately constructed. Then the attacker collects the ciphertext, which is designated as C.

Step 2: Fault Injection. The attacker injects a fault by using Rowhammer, converting AESENCLAST to AESENC.

Step 3: Faulty Encryption. In the faulty encryption, a new IV, denoted as IV_2 is generated. To ensure that the input of the faulty encryption matches that of the correct encryption, the attacker constructs a plaintext satisfying $P^{'} = P \oplus IV_1 \oplus IV_2$. The attacker then uses faulty algorithm to encrypt $P^{'}$, resulting in the faulty ciphertext $C^{'}$.

Step 4: Fault Analysis/Key Recovery. The attacker performs the offline analysis by doing Opcode-based Fault Analysis with $(P, C, C^{'})$ to recover the key.

Note that for CBC mode with 192-bit and 256-bit key lengths, EOFA can also be applied to further reduce the key space. Specifically as discussed in Section 5, a new fault can be injected into the penultimate round, flipping AESENC to AESENCLAST and producing a new faulty ciphertext $C^{''}$. The attacker uses $C^{'}$ and $C^{''}$ to obtain 2^{32}

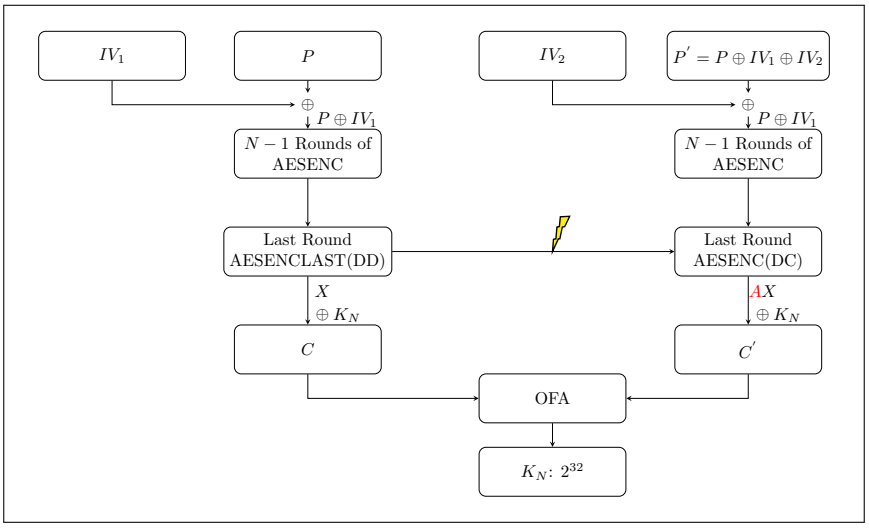


Figure 3: Attack on CBC mode with known IV.

candidates for K_{N-1}. The total key search space is reduced to 2^{32} for 192-bit keys and 2^{64} for 256-bit keys.

6.4 Attack on CBC Mode with Unknown IV

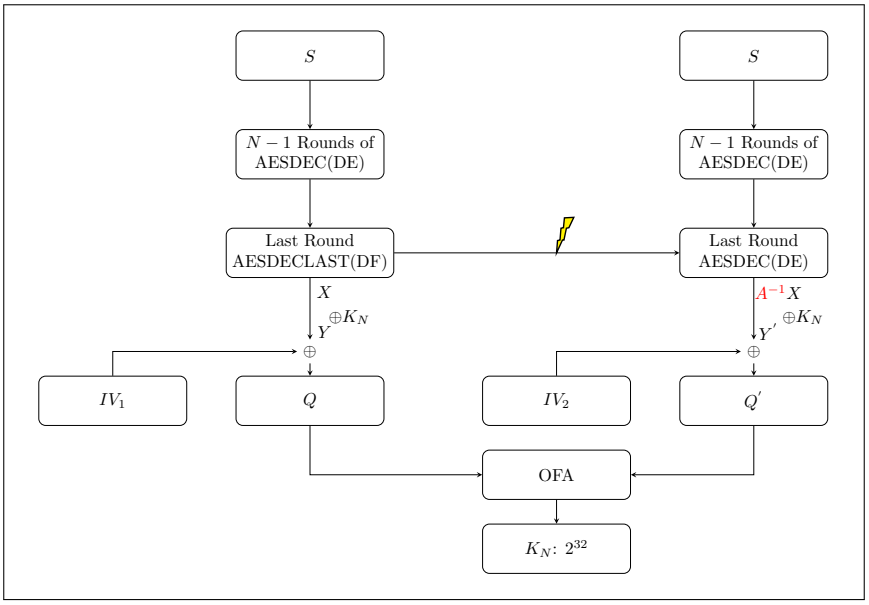


Figure 4: Attack on CBC mode with unknown IV.

In some circumstances, the value of IV is unknown to the attacker before encryption. Hence the attacker cannot construct the plaintext P' such that $P' = P \oplus IV_1 \oplus IV_2$. As a result, the attacker cannot collect the correct and faulty ciphertexts pair for the same input in the encryption process. However, in the decryption process, IV must be provided by the user along with the ciphertext as input to the module. In this scenario, it is reasonable to assume that the attacker has access to the decryption process, allowing them to input

a randomly chosen IV and a string, and subsequently obtain the corresponding output. When the attacker lacks knowledge of the key, constructing a valid ciphertext is not feasible. However, in our approach, the attacker only needs to input a 128-bit random string into the decryption module, rather than relying on valid ciphertexts. Consequently, the output is also a string of random bits. Based on this assumption, the attack proceeds as follows:

Step 1: Arbitrary Decryption. The attacker uses the device to decrypt a random string S, which is 128-bit. S is known to the attacker but not deliberately constructed. The attacker collects the output, that is denoted as Q.

Step 2: Fault Injection. The attacker injects a fault by using Rowhammer. The fault can cause a bit flip in the opcode of AESDECLAST. Thus, the AESDECLAST operation is altered to AESDEC, which implies that the last round of decryption becomes the same as the previous rounds.

Step 3: Faulty Decryption. The attacker uses the faulty algorithm to decrypt the 128 bits S again, resulting in faulty output $Q^{'}$, which can be collected by the adversary.

Step 4: Fault Analysis/Key Recovery. The attacker conducts Opcode-based Fault Analysis with $(S, Q, Q^{'})$ to recover the key.

In AES-NI, the AESDEC instruction comprises four operations: **InvShiftRows**, **InvSubBytes**, **InvMixColumns**, and **AddRoundKey**. The AESDECLAST instruction, however, consists of only three operations: **InvShiftRows**, **InvSubBytes**, and **AddRoundKey**. Changing the AESDECLAST instruction to AESDEC results in the inclusion of an additional **InvMixColumns** operation, denoted as A^{-1}. The subsequent fault analysis steps are identical to those described in Section 4.3.

Regarding the CBC mode with 192-bit and 256-bit keys and considering an unknown IV, EOFA can be applied by injecting an additional fault into the penultimate round of decryption, flipping AESDEC to AESDECLAST and generating a faulty output $Q^{''}$. Using $(Q^{'}, Q^{''})$, the search space for the penultimate round key can be deduced to 2^{32}. The total key search space is reduced to 2^{32} for 192-bit keys and 2^{64} for 256-bit keys.

7 Evaluation

After identifying potentially vulnerable opcodes, we exploit Rowhammer to flip these opcodes and recover the secret key. There are three main steps to achieve this. In the first step of *profiling memory*, we profile the physical memory allocated to the attacker, aiming to collect sufficient vulnerable pages that contain Rowhammer bit flips and their paired aggressor pages. In the second step of *faulting targeted opcodes*, the attacker, co-located with the victim on the same physical core and the same OS, tricks the OS into using a victim page to store targeted opcode of the victim. When the attacker sends encrypting queries to the victim, the victim generates and returns the ciphertexts. During the encryption process, the attacker induces bit flips in the opcode, thus collecting faulty ciphertexts. In the last step of *recovering secret keys*, we perform a post-fault analysis via OFA or EOFA to recover the secret key from the victim. For each end-to-end attack, the victim employs one of four ciphers implemented using AES-NI: AES-ECB-128, AES-ECB-192, AES-CBC-128 (with a known IV), and AES-CBC-128 (with an unknown IV).

7.1 Setup

Our experiment is conducted on Ubuntu 22.04.6 LTS environment with the kernel version of 5.4.0-84-generic. The CPU is Intel Core i3-10100 at 3.60GHz. The DRAM memory is an 8 GB Apacer DDR4 DIMM with part number D12.2324WC.001. The targeted AES implementation is based on the sample code described in the AES-NI white paper [Gue10], which is commonly used to develop programs for AES encryption and decryption, and is widely regarded as a standard reference.

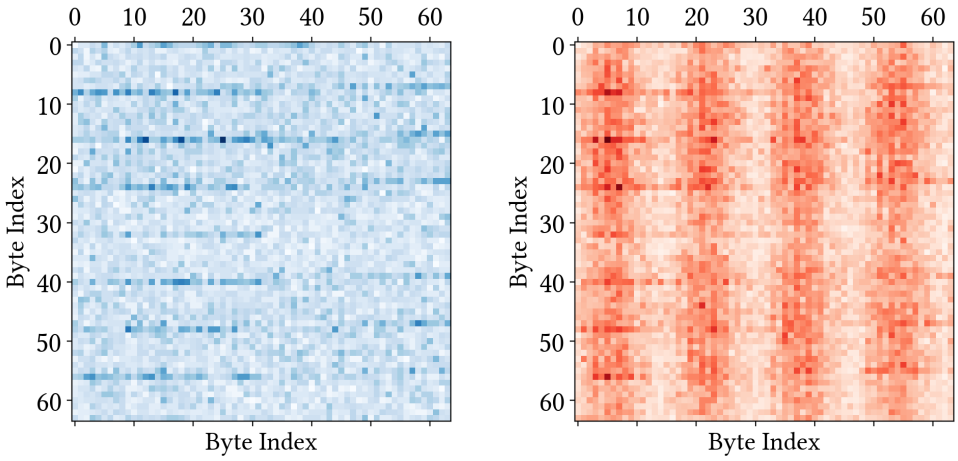


Figure 5: The distribution of flippable-bit offsets over 4 KB-aligned pages on our DDR4 module. Bit flips from 1 to 0 (blue) and bit flips from 0 to 1 (red) accumulated over 4 KB pages.

7.2 Profiling Memory

Residing in an attack process, we profile 6 GB of physical memory to identify sufficient victim pages containing Rowhammer-based bit flips and their paired aggressor pages. Achieving this requires determining whether the virtual addresses (VAs) being hammered map to different rows within the same bank. This, in turn, necessitates partial knowledge of the mapping between VAs and physical addresses (PAs), and the mapping between PAs and DRAM addresses. However, the OS kernel, prevents user processes from accessing the privileged `pagemap` interface, which translates VAs to PAs. Additionally, the second mapping, managed by the memory controller, is proprietary and not publicly disclosed by Intel.

To derive the first mapping partially, we exploit the deterministic behavior of the buddy allocator to allocate contiguous 2 MB memory blocks. In such blocks, a VA shares the least significant 21 bits with its corresponding PA. To obtain these 2 MB memory blocks, we first exhaust all free memory blocks no larger than 2 MB by using the `mmap` system call with the `MAP_POPULATE` flag. Next, we issue 2 MB memory requests via `mmap`, forcing the kernel to split 4 MB memory blocks. This yields multiple 2 MB blocks where VAs and PAs share the same lowest 21 bits.

For the second mapping, we use DRAMDig [WZCN20] to reverse engineer the machine's DRAM address function. The results show that the DRAM bank index is determined by bitwise operations on physical address bits (6, 13), (14, 17), (15, 18), and (16, 19), while the row index is determined by bits 17 to 32. Besides, there are 16 banks in total, defined by the four pairs of address bits, and 2^{16} rows per bank, with each row comprising 2^{17} bytes. Consequently, it is possible to use a VA to determine its DRAM bank and whether two VAs belong to the same row.

Last, we use TRRespass [FVH+20] to identify an effective hammering pattern for the tested DDR4 modules. Modern DDR4 and DDR5 modules are equipped with Target Row Refresh (TRR) as a defense against Rowhammer attacks. However, TRR is not foolproof and has been circumvented by TRRespass using an evasive hammering pattern. The results show that a double-sided hammer can induce reproducible bit flips.

Using the identified hammering pattern and the partial mapping knowledge described above, we initiate our hammering attempts to profile memory within the attack process.

Each attempt involves a finite loop of hammering a pair of virtual pages (a double-sided hammer) for 5,000,000 rounds. After each attempt, we inspect other pages for bit flips. Through this process, we compile a set of victim pages along with their corresponding aggressor pages.

Figure 5 shows the distribution of bit-flip page offsets across 4 KB-aligned pages. Bit flips, either from 1 to 0 or vice versa, can occur at most page offsets. As summarized in Table 4, all vulnerable pages with bit flips at the specific page offsets corresponding to targeted opcodes have been identified. Specifically, for AES-ECB-128, the targeted opcode's page offset is 0x986, with four vulnerable pages showing bit flips at this offset. For AES-ECB-192, two targeted opcodes with page offsets requiring opposite bit-flip directions must be hammered: one at offset 0x986 (flipping from bit 0 to bit 1) and another at offset 0x98c (flipping from bit 1 to bit 0). We identified five vulnerable pages at offset 0x986 and three at offset 0x98c. For AES-CBC-128 with a known IV, we found five vulnerable pages at offset 0xe2d, and for an unknown IV, eight vulnerable pages at offset 0xfd9 were identified. We note that successfully faulting a targeted opcode requires flipping a single vulnerable page with the corresponding page offset, as detailed below.

7.3 Faulting Targeted Opcodes

In this step, the attacker tricks the system into allocating a desired vulnerable page to store the targeted opcode and injects Rowhammer-based faults into the page, the so-called *memory massaging* [GLS+17]. As outlined in the first step, the page contains flippable bits at the desired page offset corresponding to the targeted opcode in the targeted AES-NI binary. We note that the target binary contains various opcodes, including those related to AES-NI instructions. When the binary is executed, all of its opcodes are loaded into memory, making them potentially susceptible to Rowhammer-induced faults. Additionally, while the page-aligned virtual address (VA) of the opcode is randomized by address space layout randomization, its page offset remains consistent.

To achieve this, we utilize a technique called `Frame Feng Shui` [KGGY20, FKK+22], which exploits the predictable behaviors of the Linux buddy allocator. This technique takes advantage of the allocator's first-in-last-out (FILO) policy for recently unmapped physical pages. Specifically, the Linux system employs the buddy system for page allocation management. Physical memory is organized into zones by the buddy allocator. When a process running on a specific CPU frees a physical page, the page is not immediately returned to the global memory pool. Instead, it is placed into a local, fast-access structure known as the `per-CPU pageset`. When the buddy allocator later needs to allocate a new page, it first attempts to retrieve a page from the top of the `per-CPU pageset`, following a stack-like access policy. This design ensures that the most recently deallocated page from the `per-cpu pageset` is prioritized when fulfilling a page request.

To this end, faulting a targeted opcode consists of four steps. First, we allocate multiple 4 KB junk pages using `mmap`. The number of junk pages is calculated based on the number

Table 4: The fault injection rate and success rate for a single attack attempt. We overcome the low fault injection rate by repeatedly hammering the victim page and invoking the targeted AES-NI binary to execute until a faulty ciphertext is generated.

Cipher	Targeted-Opcode Page Offset	Number of Vulnerable Pages	Fault Injection Rate	Success Rate
AES-ECB-128	0x986	4	4.8%	6.2%
AES-ECB-192	0x986	5	3.6%	3.4%
	0x98c	3	13.2%	5.0%
AES-CBC-128 (Known IV)	0xe2d	5	6.3%	5.8%
AES-CBC-128 (Unknown IV)	0xfd9	8	4.5%	4.6%

Table 5: Comparisons of the AES-ECB-128 attack on our DDR4 and DDR3 modules.

DRAM Chip	Targeted-Opcode Page Offset	Number of Vulnerable Pages	Fault Injection Rate	Success Rate
Apacer DDR4-2666	0x986	4	4.8%	6.2%
Samsung DDR3-1300	0x39d	5	42.0%	3.8%

of pages the targeted binary will use before allocating a page to host the target opcode. In our four end-to-end attacks, this number ranges between 126 and 130. Next, we reserve a vulnerable page that has a bit flip at a desired offset. To do this, we apply for a large number of memory pages (i.e., 4 GB in our evaluation) and then search for suitable pages that contain bit flips at the page offset corresponding to the targeted opcode in the targeted binary. Third, we release a predictable number of junk pages via `munmap`, followed by the release of the vulnerable page. These operations place the vulnerable page at the top of the allocator's stack. Last, we invoke the targeted binary to run its encryption service, the buddy allocator, adhering to its FILO policy, reuses the vulnerable page to store the targeted opcode.

In the first invocation of the victim, the process returns a correct ciphertext, ensuring that the vulnerable page with the opcode is loaded into the page cache. After this invocation completes, a Rowhammer-induced bit flip is performed to fault the opcode. During the second invocation, the victim binary reuses the faulted page from the page cache, producing a faulty ciphertext. Following this pair of invocations, we load a number of executables [GLS+17] to evict the victim binary from the page cache, ensuring that subsequent pairs of invocations are unaffected by the page cache.

Experimental results. For each victim algorithm, we perform 500 pairs of invocations. Throughout these experiments, the OS remains stable and does not crash. For AES-ECB-128, we obtained 31 pairs of correct and faulty ciphertexts. For AES-ECB-192, we collected 17 desired pairs when faulting the first opcode and 25 pairs when faulting the second opcode. For AES-CBC-128, with a known IV, we obtained 29 ciphertext pairs, and 23 pairs when the IV was unknown. We summarize these results and computes the success rate in Table 4. Besides, we also investigate the fault injection rate for the proposed attacks, which represents the difficulty of inducing Rowhammer-based bit flips into the target opcodes. We note that once our memory massaging succeeds in tricking the victim to reuse a released vulnerable page, the attacker can repeatedly invoke and hammer the victim until faulty ciphertexts are produced, effectively overcoming the challenge posed by low fault injection rates. Further, as shown in Table 5, we mount the same attack on an additional Lenovo T420 equipped with Intel Core i5-2430M CPU (Sandy Bridge), which uses a Samsung DDR3-1300 4G DIMM (part number: M473B5273DH0-YK0) to show its practicality. The results show that this additional platform is also vulnerable to our proposed attack. Last, we note that a successful full key recovery requires only one pair of correct and faulty ciphertexts.

7.4 Recovering Secret Keys

After obtaining the correct and faulty ciphertexts, we then use the method proposed in Section 4.3 through Section 6 to recover the full secret key. For each victim cipher, only one pair of correct and faulty ciphertexts is needed for post-fault analysis to recover the secret key. The results are summarized in Table 6.

For AES-ECB-128, we utilized OFA to analyze the pair of ciphertexts, achieving key recovery in 1.07 hours. For AES-ECB-192, we applied EOFA-based key recovery, with a recovery time of 1.36 hours. Regarding AES-CBC-128, we carried out key recovery experiments for each scenario: one with a known IV and the other with an unknown IV, with recovery times of 1.17 and 1.03 hours, respectively.

Table 6: The time and space complexity of OFA and EOFA (Reduction in Search Space denotes the extent to which a post-fault analysis method reduces the key search space compared to an exhaustive search).

Cipher	Fault	Post-Fault Analysis Method	Time for Online Injection	Time for Post-Fault Analysis	Reduction in Search Space
AES-ECB-128	$DD \rightarrow DC$	OFA in Sec 4.3	0.14 h	1.07 h	$2^{96}\times$
AES-ECB-192	1.$DD \rightarrow DC$ 2.$DC \rightarrow DD$	EOFA in Sec 5	0.41 h	1.36 h	$2^{160}\times$
AES-CBC-128 (Known IV)	$DD \rightarrow DC$	OFA in Sec 6.3	0.18 h	1.17 h	$2^{96}\times$
AES-CBC-128 (Unknown IV)	$DF \rightarrow DE$	OFA in Sec 6.4	0.14 h	1.03 h	$2^{96}\times$

8 Discussion

Countermeasures. We discuss how to mitigate faults on AES-NI and Rowhammer attacks. As shown in Table3, the opcodes for each instruction pair differ by only a single bit, making AES-NI susceptible to OFA. A potential countermeasure is to re-encode these pairs such that the difference between their opcodes is greater than one bit. Specifically, we propose modifying the opcode mapping so that targeted instructions, such as AESENC and AESENCLAST, are no longer adjacent in their opcode binary representations. Instead of allowing transitions between such instructions via a single-bit flip, the re-encoding enforces a minimum Hamming distance of two or more bits, thereby mitigating single-bit faults. This process begins by analyzing critical instruction pairs to identify AES-NI opcodes with a Hamming distance of 1. These opcodes are then remapped to ensure that no critical pair has a Hamming distance of 1. For instance, if AESENC is encoded as 0x66, 0x0F, 0x38, 0xDC, AESENCLAST could be remapped to 0x66, 0x0F, 0x38, 0xD3 (with a Hamming distance of 4), rather than the original 0xDD. Crucially, this re-encoding must remain transparent to software, necessitating either microcode updates or CPU-level support to translate legacy opcodes into the revised encoding scheme. However, if Intel adopts this approach to mitigate OFA, this can render legacy software incompatible with the updated hardware and distributing microcode updates across all affected devices requires coordination with relevant vendors.

To defend against Rowhammer-based bit flips and its attacks, a number of solutions have been proposed including software-only and hardware-based approaches (for more details, please refer to [ZCQ+24]). Software-only mitigations [AYQ+16, BBG+19, KOT+18, ZCW+22] do not require hardware changes, making them compatible with existing hardware. The most relevant migitation is RIP-RH [BBG+19] which modifies the underlying operating system's memory allocator to provide DRAM-aware memory isolation for local user processes. This prevents a victim process from reusing vulnerable pages released by an attacker process. Hardware-based solutions [KDK+14, SJM18, LKL+19, MJFR22, JLK+23] aim to eliminate Rowhammer bit flips by modifying hardware, though they have yet to be widely adopted in the industry, with a few exceptions [JED12, JED15]. Even for those that have been adopted, they have proven to be insecure against Rowhammer. For instance, ECC (Error Checking and Correcting) and TRR (Targeted Row Refresh) are used by DRAM manufacturers in production, but both have been reverse-engineered and bypassed [CRGB19, FVH+20].

Comparison with DFA. There are three primary distinctions between OFA and Differential Fault Analysis (DFA). First, while DFA introduces faults to alter the data state, our approach emphasizes manipulating the operations of the encryption algorithm. Second, in DFA, the states before and after the fault are similar in essence but differ only by a few bits. Conversely, in our method, the states undergo different functions due to changes in operations, such as adding or omitting a function, resulting in significant state modifications. Third, DFA often involves examining differential paths to demonstrate how

minor alterations in intermediate states affect the ciphertext. In contrast, our method bypasses this step, directly applying XOR operations to faulty and correct ciphertexts and solving equations to recover the key.

Extension on Galois/Counter Mode (GCM) and Counter Mode (CTR). AES-GCM, specified in NIST SP 800-38D [Dwo07], consists of two components: authentication and encryption/decryption, with the encryption/decryption module utilizing the AES CTR mode. Assuming that the attacker can authenticate and access the encryption/decryption module, the attack becomes equivalent to applying OFA to AES Counter Mode (CTR).

The CTR applies the forward cipher to a sequence of counters T_j and generates output blocks $O_j = \mathbb{E}_K(T_j)$. Then the output blocks are XORed with plaintext P_j to produce ciphertext C_j. In OFA, only the first block of each encryption. Thus, we omit the subscript. When applying OFA, the attacker inputs a random counter T and a plaintext P, obtaining a ciphertext C. The attacker then injects a fault by Rowhammer, altering AESENCLAST to AESENC. The faulty algorithm is then used to encrypt the plaintext P with counter T again, resulting in the faulty ciphertext $C^{'}$. Let $\mathbb{E}_{K\_correct}$ denote the correct encryption algorithm and $\mathbb{E}_{K\_faulty}$ its faulty variant resulting from the injection. The distinguishing characteristic of $\mathbb{E}_{K\_faulty}$ is the inclusion of an additional **MixColumns** operation. Let X be the 128-bit state after **ShiftRows** operation in the last round. As shown in Equation (1) the last round of **AddRoundKey** operation can be represented as: $X \oplus K_N = C$. Then we derive:

$$\begin{aligned} C \oplus C^{'} =& (P \oplus \mathbb{E}_{K\_correct}(T)) \oplus (P \oplus \mathbb{E}_{K\_faulty}(T)); \\ =& \mathbb{E}_{K\_correct}(T) \oplus \mathbb{E}_{K\_faulty}(T); \\ =& X \oplus K_N \oplus AX \oplus K_N; \\ =& X \oplus AX. \end{aligned} \tag{16}$$

The fault analysis methodology described in Section 4.3 can then be directly employed to recover the secret key.

For longer keys, Enhanced Opcode-based Fault Analysis (EOFA) introduces a second fault that converts AESENC to AESENCLAST in the penultimate round, producing additional faulty ciphertext $C^{''}$. This approach reduces the key search space to 2^{32} candidates for AES-192 and 2^{64} for AES-256

9 Conclusion

In conclusion, this paper presents a novel fault attack, which injects single bit flip faults into the opcodes, enabling the recovery of full AES keys. Our key observation is that the opcodes for various AES-NI operations differ by only a single-bit, so they can be altered to another one by introducing Rowhammer-based faults. This lays the foundation for our proposed *Opcode-based Fault Analysis* (OFA) method, which adeptly utilizes a single pair of correct and faulty ciphertexts to recover the key. Our methodological approach is underpinned by the algebraic properties of **MixColumns** operation and Gaussian elimination over finite fields. In particular, with one pair of correct and faulty ciphertexts, OFA can reduce the key search space to 2^{32} for a 128-bit key. To further reduce the key space for AES-192 and AES-256, we propose the *Enhanced Opcode-based Fault Analysis* (EOFA), which, compared to exhaustive search, reduces the key space by factors of 2^{160} and 2^{192}, respectively. We detail the attack methods for two modes (ECB and CBC) and extend the discussion to two additional modes (CTR and GCM). Our research reveals that although AES-NI can mitigate certain side-channel attacks, it is susceptible to actively induced faults caused by fault injection attacks.

Acknowledgments

This work was supported in part by National Key R&D Program of China (2023YFB3106800).

References

[ABC+24] Ravi Anand, Subhadeep Banik, Andrea Caforio, Tatsuya Ishikawa, Takanori Isobe, Fukang Liu, Kazuhiko Minematsu, Mostafizar Rahman, and Kosei Sakamoto. Gleeok: A family of low-latency PRFs and its applications to authenticated encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 545–587, 2024.

[AWK+25] Samy Amer, Yingchen Wang, Hunter Kippen, Thinh Dang, Daniel Genkin, Andrew Kwong, Alexander Nelson, and Arkady Yerukhimovich. PQ-hammer: End-to-end key recovery attacks on post-quantum cryptography using rowhammer. In *IEEE Symposium on Security and Privacy*, pages 48–48, 2025.

[AYQ+16] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd Austin. ANVIL: Software-based protection against next-generation rowhammer attacks. In *Architectural Support for Programming Languages and Operating Systems*, pages 743–755, 2016.

[BBCT22] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. Opensslntru: Faster post-quantum TLS key exchange. In *USENIX Security Symposium*, pages 845–862, 2022.

[BBG+19] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. RIP-RH: Preventing rowhammer-based inter-process attacks. In *Asia Conference on Computer and Communications Security*, pages 561–572, 2019.

[BDL97] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51, 1997.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology*, pages 513–525, 1997.

[CRGB19] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: on the effectiveness of ECC memory against rowhammer attacks. In *IEEE Symposium on Security and Privacy*, pages 55–71, 2019.

[CVM+21] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David F. Oswald, and Flavio D. Garcia. Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security Symposium*, pages 699–716, 2021.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, pages 644–654, 1976.

[DR99] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael document version 2. AES Algorithm Submission, September 1999. Available at https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf.

[Dwo01] Morris Dworkin. NIST special publication 800-38a 2001 edition. *NIST Special Publication*, 800(3), 2001.

[Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC, 2007. NIST Special Publication (SP) 800-38D.

[FKK+22] Michael Fahr, Hunter Kippen, Andrew Kwong, Thinh Dang, Jacob Lichtinger, Dana Dachman-Soled, Daniel Genkin, Alexander Nelson, Ray A. Perlner, Arkady Yerukhimovich, and Daniel Apon. When frodo flips: End-to-end key recovery on frodokem via rowhammer. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 979–993, 2022.

[FVH+20] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. In *IEEE Symposium on Security and Privacy*, pages 747–762, 2020.

[GLS+17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. *IEEE Symposium on Security and Privacy*, pages 245–261, 2017.

[Gue10] Shay Gueron. Intel advanced encryption standard AES new instructions set. *Intel Corporation*, 128, 2010.

[GZZ+25] Xue Gong, Fan Zhang, Xinjie Zhao, Jie Xiao, and Shize Guo. Key schedule guided persistent fault attack. *IEEE Transactions on Information Forensics and Security*, pages 767–780, 2025.

[HAZ+24] Junhao Huang, Alexandre Adomnicai, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium implementations on ARMv7-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–24, 2024.

[JED12] JEDEC. DDR4 SDRAM Specification, 2012.

[JED15] JEDEC Solid State Technology Association. Low power double data rate 4 (LPDDR4). https://www.jedec.org/standards-documents/docs/jesd209-4b, 2015.

[JLK+23] Jonas Juffinger, Lukas Lamster, Andreas Kogler, Maria Eichlseder, Moritz Lipp, and Daniel Gruss. CSI:Rowhammer – cryptographic security and integrity against Rowhammer. In *IEEE Symposium on Security and Privacy*, pages 236–252, 2023.

[JT12] Marc Joye and Michael Tunstall. *Fault analysis in cryptography*, volume 147. Springer, 2012.

[JWS+24] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. Zenhammer: Rowhammer attacks on AMD zen-based platforms. In *USENIX Security Symposium*, pages 1615–1633, 2024.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture*, pages 361–372, 2014.

[KGGY20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *IEEE Symposium on Security and Privacy*, pages 695–711, 2020.

[KOT+18] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. ZebRAM: comprehensive and compatible software protection against rowhammer attacks. In *Operating Systems Design and Implementation*, pages 697–710, 2018.

[LKL+19] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: preventing row-hammering by exploiting time window counters. In *International Symposium on Computer Architecture*, pages 385–396, 2019.

[LTH22] Xiang Li, Russell Tessier, and Daniel Holcomb. Precise fault injection to enable DFIA for attacking AES in remote FPGAs. In *Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 1–5, 2022.

[MDT+23] Koksal Mus, Yarkın Doröz, M Caner Tol, Kristi Rahman, and Berk Sunar. Jolt: Recovering TLS signing keys via rowhammer faults. In *IEEE Symposium on Security and Privacy*, pages 1719–1736. IEEE, 2023.

[MIS20] Koksal Mus, Saad Islam, and Berk Sunar. *QuantumHammer: A Practical Hybrid Attack on the LUOV Signature Scheme*, page 1071–1084. 2020.

[MJFR22] Michele Marazzi, Patrick Jattke, Solt Flavien, and Kaveh Razavi. PROTRR: Principled yet optimal in-dram target row refresh. In *IEEE Symposium on Security and Privacy*, 2022.

[MKS12] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. Are AES x86 cache timing attacks still feasible? In *Cloud computing security Workshop*, pages 19–24, 2012.

[MOG+20] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *IEEE Symposium on Security and Privacy*, pages 1466–1482, 2020.

[RD01] Vincent Rijmen and Joan Daemen. Advanced encryption standard. *Proceedings of federal information processing standards publications, national institute of standards and technology*, 19:22, 2001.

[Res18] Eric Rescorla. The transport layer security TLS protocol version 1.3. https://www.rfc-editor.org/rfc/rfc8446, 2018.

[SJM18] Seyed Mohammad Seyedzadeh, Alex K Jones, and Rami Melhem. Mitigating wordline crosstalk using adaptive trees of counters. In *International Symposium on Computer Architecture*, pages 612–623, 2018.

[TMA11] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *IFIP international workshop on information security theory and practices*, pages 224–233, 2011.

[WTM+20] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–25, 2020.

[WZCN20] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In *Design Automation Conference*, pages 1–6, 2020.

[ZCQ+24] Zhi Zhang, Decheng Cheng, Jiahao Qi, Yueqiang Cheng, Shijie Jiang, Yiyang Lin, Yansong Gao, Surya Nepal, Yi Zou, Jiliang Zhang, and Yang Xiang. SoK: Rowhammer on commodity operating systems. In *Asia Conference on Computer and Communications Security*, 2024.

[ZCW+22] Zhi Zhang, Yueqiang Cheng, Minghua Wang, Wei He, Wenhao Wang, Nepal Surya, Yansong Gao, Kang Li, Zhe Wang, and Chenggang Wu. Softtrr: Protect page tables against rowhammer attacks using software-only target row refresh. In *USENIX Annual Technical Conference*, 2022.

[ZHC+21] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Minghua Wang, Kang Li, Surya Nepal, and Yang Xiang. Bitmine: An end-to-end tool for detecting rowhammer vulnerability. *IEEE Transactions on Information Forensics and Security*, pages 5167–5181, 2021.

[ZHF+23] Fan Zhang, Run Huang, Tianxiang Feng, Xue Gong, Yulong Tao, Kui Ren, Xinjie Zhao, and Shize Guo. Efficient persistent fault analysis with small number of chosen plaintexts. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 519–542, 2023.

[ZLZ+18] Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent fault analysis on block ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 150–172, 2018.

[ZZJ+20] Fan Zhang, Yiran Zhang, Huilong Jiang, Xiang Zhu, Shivam Bhasin, Xinjie Zhao, Zhe Liu, Dawu Gu, and Kui Ren. Persistent fault attack in practice. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):172–195, 2020.