Software Countermeasures against DVFS fault Attack for AES

Zikang Tao, Rihui Sun, and Jian Dong*
Harbin Institute of Technology, Harbin, Hei Longjiang, China
ziktao@stu.hit.edu.cn, 19B903009@stu.hit.edu.cn, dan@hit.edu.cn
*corresponding author

Abstract—Most modern processors utilize Dynamic Voltage and Frequency Scaling (DVFS) technology to dynamically adjust voltage and frequency according to the workloads of processors to reduce power consumption. However, DVFS can be exploited by attackers to introduce transient hardware faults. Combined with Differential Fault Analysis (DFA), the secure execution environment of processors is threatened. In order to minimize the threat of DVFS fault attack, we analyze its principle and focus on countermeasures for AES applications. We provide a detailed analysis of three software countermeasures, namely redundancy-based methods, parity code, and infective computation. Among them, the temporal redundant method with random latency between executions defends against high-order attacks with 34.18% timing overhead, while the infective computation-based method achieves the best security with nearly 124.35% timing overhead.

Keywords–DVFS, Fault Injection Attack, Countermeasure

1. Introduction

Nowadays, with the sharp increase in processor frequency and complexity, coupled with multi-core technology, processor power consumption is becoming a major problem. To both reduce CPU power consumption at low workloads and meet the high-frequency requirements at high workloads, dynamic voltage and frequency scaling (DVFS) is utilized. It dynamically adjusts the voltage and frequency according to the workload of the processor, thus achieving the purpose of reducing power consumption. It is available on ARM, Intel, and AMD CPUs, and can be controlled from privileged software to provide more flexible adjustments on voltage and frequency. However, malicious adjustment can lead to broken timing constraints on the CPU. This can cause silent data corruption (SDC) and finally lead to a system crash, posing a threat to security domains on the CPU.

Tang et al. proposed the first DVFS-based fault attack, CLKSCREW attack [1] in 2017, which is fully controlled by software without any hardware assistance. This attack introduces a fault by overclocking through the system software to exceed the legitimate upper bound. Combined with differential fault analysis (DFA), they successfully obtained the AES cryptographic key from ARM TrustZone and escalate a malicious kernel driver's privileges by loading self-signed code into TrustZone. However, overclocking may cause damage to the processor and some Intel and AMD processors restrict the administrator to set the processor frequency only within the legal range for safety reasons. Thus, CLKSCREW can not pose a threat to those overclocking-restricted processors.

To overcome the drawbacks of overclocking, many researchers have devoted themselves to the study of undervolting. Qiu et al. proposed the first undervolting-based DVFS attack, VoltJockey attack [2] and successfully obtained the AES cryptographic key from ARM TrustZone and loaded untrusted applications into TrustZone by invalidating the RSA verification. They then demonstrated the effectiveness of the VoltJockey attack on Intel SGX [3] by obtaining the AES cryptographic key from the enclave and leading RSA to give any expected outputs. Concurrently, Murdock et al. and Kenjar et al. present similar attacks, Plundervolt [4] and V0ltpwn [5], respectively. Despite the difference in attacking targets and experimental environments, they do share a single principle, software-undervolting by writing specific values into voltage-related MSR. In response to the vulnerability revealed Intel issued a CVE (CVE-2019-11157) [6] and modified the SGX remote attestation process to verify that overclocking mailbox (OCM) and its model-specific registers (MSRs) allowing software-undervolting are disabled via a microcode update. Differ from those software-controlled attacks mentioned before, VoltPillager [7] bypasses the CPU and OCM, and directly sends commands over the bus, thus bypassing the mitigation provided by Intel.

Therefore, simply disabling the voltage-related MSRs can not completely eliminate the vulnerability induced by DVFS. In addition, as undervolting can decrease dynamic power consumption, disabling it leads to higher power consumption on the CPU. In this paper, we focus on the AES encryption procedure and implement several software countermeasures to defend it against DVFS fault attacks. We then analyze their security and performance overheads. Compared to hardware countermeasures, software countermeasures can be implemented easily and inexpensively. Software countermeasures only add a certain amount of temporal and spatial overheads when the user's applications are running and do not require any hardware updates. Furthermore, software countermeasures can be easily deployed on devices of different platforms and are more versatile than hardware countermeasures.

The contributions of this work are:

- We detailedly discuss the principle of the DVFS fault attack and its security implications.
- We summarize different kinds of existing countermeasures and implement them and analyze their securities and performance overheads.

The rest of this paper is organized as follows. Section 2

2767-6684/23/$31.00 ©2023 IEEE
DOI 10.1109/DSA59317.2023.00080

introduces the basic backgrounds of DVFS, AES, DFA, and the typical attack procedure of DVFS fault attacks. Section 3 describes the countermeasures implemented in this paper. Section 4 evaluates the performance overhead and security. Section 5 shows relevant works on software countermeasures against fault attacks. Section 6 concludes the article.

2. Preliminaries

In this section, we provide backgrounds about Dynamic Voltage and Frequency Scaling (DVFS), Advanced Encryption Standard (AES), Differential Fault Analysis (DFA), and the typical attack procedure of DVFS fault attacks.

2.1. DVFS

DVFS, also known as Dynamic Voltage and Frequency Scaling, is a commonly used method to reduce energy consumption by dynamically adjusting the device's voltage and frequency. It is widely used in various computing devices such as embedded devices, personal computers, and servers. Most digital circuits, especially processors, use CMOS integrated circuits, whose power consumption can be divided into static power consumption and dynamic power consumption [8]. Dynamic power consumption is proportional to the square of the clock frequency(f) and voltage(V) of the device, as shown in (1).

$$P_{dynamic} \propto CfV^2 \tag{1}$$

where C means load capacitance of the integrated circuit. DVFS reduces the dynamic power consumption of a circuit by dynamically reducing the frequency and voltage of the device when workloads are low, thereby reducing the overall power consumption of the circuit.

On the other hand, as a single dynamic scaling strategy may not meet the needs of diverse workloads, OS provides DVFS Drivers for administrators to manually select a governor from all governors that are supported. For example, kernel module *CPUfreq* acts as DVFS driver in Linux by default, and it provides six governors: (1) *performance*, (2) *powersave*, (3) *userspace*, (4) *schedutil*, (5) *ondemand*, (6) *conservative*. Among all six governors, the *userspace* governor provides the ability to manually set frequencies of CPU cores.

Although DVFS brings considerable energy saving, dynamic voltage, and frequency may threaten processor security. A processor must meet timing constraints to run error-freely, which are shown in Figure 1. To ensure that the signal can be propagated to the destination flip-flop within one clock cycle, the following inequation must be satisfied:

$$T_{clk} \geq T_{FF} + T_{max_path} + T_{setup} \tag{2}$$

where T_{clk} is the time between two clock positive edges, T_{FF} is the latency of flip-flop, T_{max_path} is the propagating latency of the critical path, and T_{setup} is the amount of time required for the input to a flip-flop to be stable before a clock edge. T_{FF} and T_{setup} remain the same when frequency and/or voltage changes, T_{max_path} and T_{clk}, however, vary with the variations of frequency and voltage. Timing constraints can be broken when voltage is decreased by malicious attackers, resulting

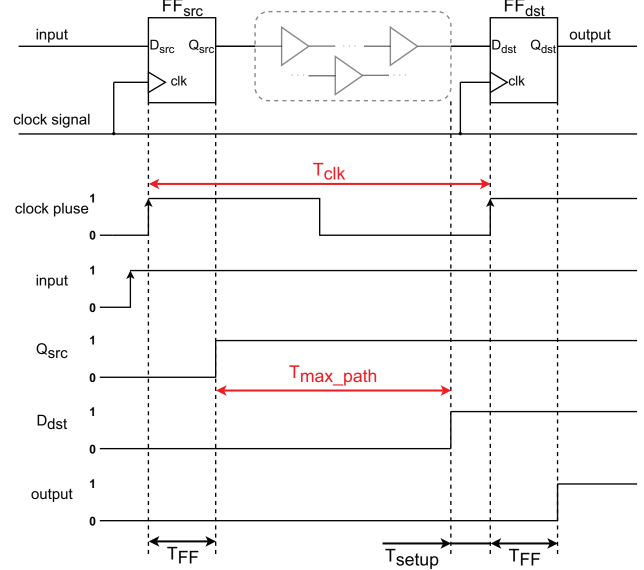

Figure 1. Timing constraint for error-free data propagation from input Q_{src} to output D_{dst} for entire circuit

in an increment in T_{max_path}; or frequency is increased by overclocking, resulting in a decrement in T_{clk}. In consequence, the signal can not be propagated to the destination flip-flop within one clock cycle, causing a bit-flip in the destination flip-flop. It can cause silent data corruption and finally lead to information leakage or system crash.

2.2. AES

Advanced Encryption Standard [9] is a typical symmetric key block cipher, which divides the plaintext into several groups of fixed size (128 bits). It encrypts and decrypts each group individually, and uses the same key for the encryption and decryption processes. AES organizes every 128-bit packet into a 4×4 bytes matrix, called the state (S), and transforms it several times to obtain the encrypted/decrypted data. The state can be denoted as follows:

$$S = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \tag{3}$$

The number of transformation rounds is determined by the AES key length, which can be 128 bits, 192 bits, and 256 bits, corresponding to 10, 12, and 14 rounds of transformations, respectively. Each round of transformations uses a corresponding round key, which is obtained by a key schedule process. In each round, the transformation process consists of four transformations: (1) SubBytes, (2) ShiftRows, (3) MixColumns, and (4) AddRoundKey. In particular, the AES128 encryption procedure performs an AddRoundKey transformation before the start of the first encryption round, as well as no column mixing during the 10th round of encryption. We detail four transformations of each round as follows:

- **SubBytes** — Each byte undergoes a nonlinear substitution. It is implemented by replacing each byte $a_{i,j}$ of the state with the byte S-box($a_{i,j}$) in the corresponding position of the S-box (Substitution-box) for simplicity. The S-box is precomputed and stored in a 256×8 bits lookup table.
- **ShiftRows** — The bytes of the first, second, third, and fourth rows of the state are left-shifted by 0, 1, 2, and 3 bytes, respectively.
- **MixColumns** — The four bytes in each column are mixed together in a specific way to generate a new column. It can be decribed as follows:

$$\begin{pmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{pmatrix} = \begin{pmatrix} \alpha & \beta & 1 & 1 \\ 1 & \alpha & \beta & 1 \\ 1 & 1 & \alpha & \beta \\ \beta & 1 & 1 & \alpha \end{pmatrix} \times \begin{pmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{pmatrix} \quad (4)$$

  where $b_{i,j}$ is the byte in the ith row and the jth column of the new state obtained after the transformation, $\alpha = 02 = x$ and $\beta = 03 = x + 1$. It should be mentioned that the multiply and add operations in (4) are modulo 2 multiply and bitwise xor respectively and the α and β perform modulo the irreducible polynomial of AES128 which is $g(x) = x^8 + x^4 + x^3 + x + 1$.
- **AddRoundKey** — The round key, which is generated by using a key schedule process, is added through bitwise xor to the state.

2.3. DFA

In 1997, Boneh et al. [10] proposed a new attack on RSA, which utilizes the error outputs to steal the secret. Biham et al. [11] extended this attack to other encryption algorithms, and named it Differential Fault Analysis (DFA). It collects error outputs and correct outputs and analyzes them, extracting useful information about the secret, which can be used to reduce the possible key hypotheses. Since AES was proposed, the research on DFA for AES had begun, and many effective attacks were proposed[12][13][14].

Tunstall et al. [14] proposed one of the most effective DFA attacks for AES, using only one pair of fault cipher and correct cipher to reduce the possible key hypotheses to a mere 2^8. According to the characteristics of the AES encryption procedure, if an attacker injects a fault into the state after the $(n-3)th$ MixColumns (where n is the number of rounds of the AES encryption procedure) and before the $(n-2)th$ SubBytes, the error will propagate to the entire 4×4 state during the $(n-1)th$ MixColumns. The resulting fault cipher, which contains information about round keys, is analyzed with the correct cipher and reduces the possible key hypotheses to a much smaller set. If another fault cipher is produced, the possible key hypotheses can be further shrunk and the secret can be extracted with high possibility.

2.4. Typical Attack procedure

As overclocking/undervolting can easily induce faults into CPU cores, it can be used to collect the fault ciphers for DFA. The typical attack procedure of a DVFS fault attack on AES128 is shown as follows:

1) Running the victim (i.e. AES128 encryption procedure) on one CPU core and waiting until AES128 finishes its 7th MixColumns.
2) Undervolting or overclocking by MSRs or drivers, which may cause a fault in the state after the 7th MixColumns and before the 8th SubBytes.
3) Get a fault output cipher and analyze it with the correct cipher by DFA. Update the possible key hypotheses S according to the results. If S has more than one candidate key, go to 1); else, terminate, as the very single candidate key must be the secret key used by AES128.

According to the capabilities of injecting faults, the attacks can be divided into two categories: one-order attacks and high-order attacks. If an attack can induce a single fault at an arbitrary position during one encryption, we call it a one-order attack. Instead, if an attack can induce more than one fault at arbitrary positions during one encryption, it is called a high-order attack.

This kind of attack is fully software-controlled and can be conducted remotely. An attacker can easily launch a high-order attack by simply undervolting twice and precisely control the fault positions to bypass many redundancy-based countermeasures. Thus, effective countermeasures against the attack should be summarized and evaluated to meet different security and performance demands.

3. Software Countermeasures

In this section, we describe the basic principles underlying the software countermeasures against fault attacks: fault detection and infective computation, and give details about our implemented ones. In this section, we only focus on the encryption process of AES128 as there are minor differences between the encryption and decryption processes. The decryption process just performs inverse transformations to revert the change in the states.

3.1. Fault Detection

Fault detection can be implemented by redundancy and error-detecting codes. This kind of method has a verification stage where original information and redundant one are compared to each other. A cipher will be produced only when the two match. No information about fault cipher should be produced when the verification stage fails.

3.1.1. **Redundancy-based Methods**: Redundancy-based methods can be divided into algorithm-level, round-level, and operation-level. As they share the same principle — duplicating, we only implement algorithm-level redundancy for the sake of brevity. Redundancy-based methods duplicate the encryption process (using either temporal or spatial redundancy) and compare the two results and output results only when they match. These methods can be divided into two categories: temporal and spatial redundancy.

- Temporal redundancy runs the encryption process twice serially and output ciphers when two results match. It provides the fault-detecting ability at a cost of almost doubled execution time theoretically. A simple extension is

running the encryption process three times and we can easily correct one error by simply choosing the major outputs, undertaking an even larger performance overhead. However, they are vulnerable to high-order attacks, which are capable to induce the same faults at the same position of different encryption processes.
- Spatial redundancy, on the other hand, runs two encryption processes on different CPU cores in parallel and output ciphers when two results match. It provides the fault-detecting ability at a cost of almost doubling hardware resources. However, as multi-thread require an extra multi-thread library, it asks for a lot of memory spaces and has a great impact on performance because thread creating is a time-consuming procedure as we will show in Section 4.

3.1.2. **Error-detecting Code**: Differing from entirely duplicating the whole encryption process, the error-detecting code attaches less amount of redundant information to the input plaintext. During the encryption process, the redundant information transforms accordingly and finally obtains a bunch of check words. They will be compared to those calculated by the result of the encryption process and an output will be produced only when they match. Due to the complexity of the AES encryption process, transforms of error-detecting codes during the encryption process can be hard to implement and consume a lot of time and/or memory space. Bertoni et al. proposed an effective method that is based on parity code [15] and implemented it on hardware. Its basic idea is to attach one parity bit for each byte of the state of the AES encryption process, obtaining a 4×4 bits parity matrix. The parity matrix transforms along with transformations of the state, called parity predictions. We detail four transformations of the parity matrix on even parity in each round as follows:
- During SubBytes, every byte of the state is split into upper four bits and lower four bits and used as row index and column index of S-Box respectively. Therefore, we can precompute parity bits for every byte in S-Box and store them in memory. When the original byte of the state is substituted with the byte indexed, the parity bit is substituted accordingly.
- During ShiftRows, rows of the state are shifted progressively. Thus, we just need to shift rows of the parity matrix accordingly.
- During MixColumns, columns of the state are mixed together as shown in (4). Transformations of the parity matrix are shown as follows:

$$\begin{pmatrix} p_{0,j} \\ p_{1,j} \\ p_{2,j} \\ p_{3,j} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} p_{0,j} \\ p_{1,j} \\ p_{2,j} \\ p_{3,j} \end{pmatrix} + \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} a^{(7)}_{0,j} \\ a^{(7)}_{1,j} \\ a^{(7)}_{2,j} \\ a^{(7)}_{3,j} \end{pmatrix} \quad (5)$$

where $p_{i,j}$ is the parity bit associated with state byte $a_{i,j}$, and $a^{(7)}_{i,j}$ is the most significant bit of $a_{i,j}$. We give a brief proof of the formula of $p_{0,j}$, the remainings can be proven similarly and we omit them for the sake of brevity.

Proof. As shown in (4), $b_{0,j}$ can be obtained by the following formula. It should be mentioned that $+$ and $\times$ are bitwise xor and modulo 2 multiple, respectively.

$$b_{0,j} = \alpha \times a_{0,j} + \beta \times a_{1,j} + a_{2,j} + a_{3,j} \quad (6)$$

Define $par(a)$ as the parity of integer a, $p_{0,c}$ can be calculated as follows:

$$\begin{aligned} p_{0,j} &= par(\alpha \times a_{0,j} + \beta \times a_{1,j} + a_{2,j} + a_{3,j}) \\ &= par(\alpha \times a_{0,j}) + par(\beta \times a_{1,j}) \\ &\quad + p_{2,j} + p_{3,j} \end{aligned} \quad (7)$$

As $\alpha \times a_{0,j} = a_{0,j} << 1$, $\beta \times a_{1,j} = (a_{1,j} << 1) + a_{1,j}$, the following formula holds:

$$par(\alpha \times a_{0,j}) = p_{0,j} + a^{(7)}_{0,j} \quad (8)$$

$$par(\beta \times a_{1,j}) = a^{(7)}_{1,j} \quad (9)$$

Bring them all together, $p_{0,j} \mapsto p_{0,j} + p_{2,j} + p_{3,j} + a^{(7)}_{0,j} + a^{(7)}_{1,j}$ holds. □

- During AddRoundKey, the round key is added through bitwise xor to the state. Consider two integers a and b whose numbers of bit 1 are x and y respectively and the number of bit 1 in both a and b is t. Then, the number of bit 1 in $a \oplus b$ can be calculated as $p = x + y - t \times 2$. Therefore, the parity of $a \oplus b$ equals that of $x + y$, which can be simply calculated by adding the parities of a and b together through bitwise xor. So, we can simply add the parities of bytes in the round key through bitwise xor to the parity matrix.

During each transformation, the state is modified, and the parity matrix is predicted accordingly. Therefore, verification can be inserted after every transformation, every encryption round, or the whole encryption process, leading to degressive overheads, yet degressive capacities for locating the faults injected. We implemented this parity-code-based method in the software version and analyze its performance overhead and security in Section 4.

3.2. Infective Computation

Fault detection schemes have an intrinsic drawback — a verification stage is requisite before output ciphers, which can be injected and bypassed. To prevent an adversary from using high-order attacks to bypass the verification stage, Yen et al. [16] first introduced the principle of infective computation to protect RSA against fault attacks. The basic idea of infective computation is not detection, but infection — if a fault is injected in the encryption process, the output cipher will be polluted, so that even if the attacker gets the erroneous cipher, he/she can't utilize it to steal secrets. There are two ways to realize infective computation:
- Use secret error to infect input and remove the effect of previously introduced error before the program outputs.

- Additional operations are introduced, which will not affect the program output if no fault is injected, on the contrary, the output will be infected if a fault injection occurs.

Tupsamudre et al. [17] proposed a countermeasure based on additional operations. It duplicates AES encryption rounds and randomly inserts dummy rounds between them, namely redundant rounds and dummy rounds, which play roles as additional operations. The dummy rounds always take a fixed state as input and output the same state called β. Moreover, the randomness of dummy rounds from a time perspective disrupts the consecutive execution of redundant rounds and cipher rounds, which effectively protect AES from high-order attacks, as the attacker can not easily induce two faults at the very same position of the states in two rounds. Details of this countermeasure are described in Algorithm 1.

Algorithm 1 Tupsamudre et al.'s countermeasures for AES128

Input: P, k^j for $j \in \{1, \cdots, n\}, (\beta, k^0), (n = 11)$
Output: BlockCipher(P, K)

1: $R_0 \leftarrow P, R_1 \leftarrow P, R_2 \leftarrow \beta$
2: $i \leftarrow 1, q \leftarrow 1$
3: $rstr \leftarrow \{0,1\}^t \quad //\#1(rstr) = 2n, \#0(rstr) = t - 2n$
4: **while** $q \leq t$ **do**
5: $\quad \lambda \leftarrow rstr[q]$
6: $\quad \kappa \leftarrow (i \wedge \lambda) \oplus 2(\neg\lambda)$
7: $\quad \zeta \leftarrow \lambda \cdot \lceil i/2 \rceil$
8: $\quad R_\kappa \leftarrow RoundFunction(R_\kappa, k^\zeta)$
9: $\quad \gamma \leftarrow \lambda(\neg(i \wedge 1)) \cdot BLFN(R_0 \oplus R_1)$
10: $\quad \delta \leftarrow (\neg\lambda) \cdot BLFN(R_2 \oplus \beta)$
11: $\quad R_0 \leftarrow (\neg(\gamma \vee \delta) \cdot R_0) \oplus ((\gamma \vee \delta) \cdot R_2)$
12: $\quad i \leftarrow i + 1, q \leftarrow q + 1$
13: **return** R_0

where λ indicates whether the current round is a cipher/redundant round; κ indicates which state should the current round use; ζ indicates the round key used in the current round; γ and δ indicate whether there are any faults occur in the current cipher/redundant and dummy round, respectively; $BLFN$ maps all non-zero input to 1 and only returns 0 when input is 0. The algorithm has a total of t rounds, consisting of n cipher rounds, n redundant rounds, and $t - 2n$ dummy rounds. The timing of dummy rounds is randomly set at the beginning of the algorithm, and the intermediate results of these rounds are stored in R_0, R_1, and R_2, respectively. Before cipher rounds are executed, the corresponding redundant rounds will be executed. Then the two results will be bitwise xored together. If the xor result is not 0, γ will be set to 1, indicating there are faults injected in cipher rounds or redundant rounds. When faults are injected in dummy rounds, R_2 will be changed and not the same with β. Faults injected in either cipher/redundant rounds or dummy rounds will lead to an infection on R_0, eventually making the faults unexploitable. As the input and output of the dummy round remain the same and R_2 is initialized with β, the following equation holds:

$$RoundFunction(\beta, k^0) = \beta \tag{10}$$

For every randomly generated β, k^0 can be easily calculated by running an AES encryption round with β as its input plaintext. Let S_M be the state after MixColumns, according to (10), $S_M \oplus k^0 = \beta$ holds. Thus, k^0 can be simply calculated by $k^0 = \beta \oplus S_M$. In this way, β can be set randomly and k^0 is calculated accordingly.

4. Evaluation

In this section, we analyze the security of the three countermeasures implemented and compare their performance overheads from both temporal and spatial perspectives.

4.1. Security Analysis

Redundancy-based methods hypothesize that the injected faults are transient and will not inject the same faults exactly at the same time in two executions. Therefore, they are vulnerable to high-order fault attacks capable of injecting the same faults during two executions. An effective way to prevent this from happening is adding random latency between two executions.

The parity-based method fails when the number of bits flipped in each byte of the state matrix is even. According to Karpovsky's analysis [18], the number of undetectable errors is 2^k for (n,k)-linear code, indicating coverage of $1 - 2^{-(n-k)}$. For the parity-based method which attaches a parity bit for every byte of the state, the coverage is $1-2^{-16} \approx 99.99847\%$, which is close to the simulation result in [15]. Therefore, this method can protect AES against one-order attacks with an acceptable probability. When it comes to high-order attacks, it also fails when an attacker injects an odd-bit error at both state transformations and parity predictions. To make it even worse, if an attacker can induce fault into the status register and bypass the verification stage, this method will fail as the verification becomes invalid. The same conclusion holds for redundancy-based methods. Fortunately, we can check parity after every prediction, making it impracticable to bypass all verifications.

The infective computation-based method implemented in this paper shares similarities with the temporal redundancy-based method. It infects the cipher when a fault is injected in additional operations and disrupts two executions using random dummy rounds. Therefore, it can protect AES from high-order attacks. Moreover, because there is no verification stage in this method based on infective computation, it is immune to verification-bypass attacks. Therefore, this scheme outperforms the other schemes from the perspective of security.

4.2. Overhead

Experimental Setup: The experiment device has an Intel Core i5-9400F processor, 16-GB memory; the OS is Ubuntu 20.04 LTS whose kernel version is 5.13.0–30. The processor has 6 physical cores and 6 logic cores whose frequency can be up to 4.1 GHz.

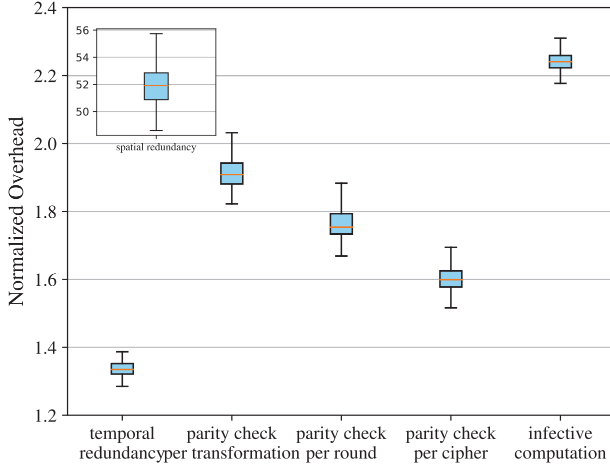

Figure 2. Normalized temporal overheads of the implemented countermeasures. The boundaries of the whiskers are based on the 1.5 times IQR value as popularly elected. We dismiss the outliers for aesthetics.

4.2.1. Temporal Overhead: Temporal overhead is one of the most important parts of performance overhead. Temporal overheads of all implemented methods in this paper are shown in a box and whiskers plot[1] in Figure 2. The method based on infective computation carries out 10 rounds of cipher rounds, 10 rounds of redundant rounds, and 6 rounds of dummy rounds. We evaluate every countermeasure 300 times and average the results.

Due to the good locality, the temporal redundancy-based method incurs 34.18% temporal overhead, rather than 100%, as many data and instructions are already stored in caches during the second encryption. Contrary to popular belief, the spatial redundancy-based scheme incurs more than 5100% temporal overhead, leading to severe performance degradation. After in-depth analysis, it is found that the time consumed in creating a thread and waiting for the termination of the thread just created is about 22 and 28 times more than the AES encryption procedure, respectively.

According to the different timing for verification, the parity code-based method incurs 93.06%, 77.08%, and 61.42% temporal overhead, respectively, far more than that of the temporal redundancy-based method. It seems to be contradictory to the opinion that error-detecting codes usually require a smaller overhead compared to straightforward duplication as they only attach a smaller amount of redundant information. This is because error-detecting codes are more suitable for hardware implementation. When implemented in hardware, it is easy to implement predictions of various error-detecting codes in parallel, saving a lot of time used to predict serially in the software implementation.

[1]A box-and-whiskers plot emphasizes the important metrics of a dataset's distribution. The box is drawn from the first quartile to the third quartile with a horizontal line drawn in the middle to denote the median. The interquartile range (IQR) is the distance between the first and third quartiles (i.e., box size). The whiskers end at an observed data point, but can be defined in various ways.

Table 1. Spatial overheads of implemented countermeasures

Countermeasures		code size	data structure(bit)
Redundancy	Temporal	+12.04%	+128
	Spatial	+12.90%	+3584
Parity Code		+79.96%	+448
Infective Computation		+65.34%	+400

The infective computation-based method implemented in this paper incurs a temporal overhead of 124.55%. Considering it carries out 26 rounds in all, these methods are very cost-ineffective compared to the temporal redundancy-based method which carries out 20 rounds and only incurs 34.18% temporal overhead. To defend against verification-bypass attacks, infective computation removes all vulnerable condition jump instructions, replacing them with several computations that will be executed in every single round, whether faults are injected or not, even if some computations (e.g. calculating δ in cipher and redundant rounds) are pre-determinately not necessary in the current round. They offer higher security at an acceptable cost for safety-critical systems and applications.

4.2.2. Spatial Overhead: The temporal-redundancy-based method executes the same encryption process several times and only needs a copy of input plaintext, yielding extra 128 bits for AES128. The code segment size of the executable is 2085 bytes, which is only 12.04% larger than that of non-protect AES128 (1861 bytes). The spatial-redundancy-based method requires two CPU cores to run the encryption process at the same time, thus the size of data structures used by another instance is $256 \times 8 + 11 \times 16 \times 8 + 128 = 3584$ bits. The code segment size of the executable is 2101 bytes, yielding an increment of 12.90%. However, multi-threads need extra dynamic link library support and extra system structures used in another thread, thus consuming far more space than twice the original.

The extra space cost of the method based on parity code mainly focuses on the storage of the parity matrix and the increment of code size. It is necessary to store the parity matrix of the state matrix, the round key, and the S-box, yielding a total of $16 + 11 \times 16 + 256 = 448$ bits of extra space. The code segment size of this method is 3349 bytes, which is increased by 79.96% compared with non-protect AES128.

The extra space cost of the method based on infective computation implemented in this paper mainly focuses on the input plaintext copies for redundant rounds and dummy rounds, the random state used by dummy rounds, a map used for accelerating the calculation of $BLFN$, and the increment of code size. Extra data structures used are $128 \times 3 + 16 = 400$ bits in total. The code segment size of this method is 3077 bytes, which is 65.34% larger than non-protect AES128.

The spatial overheads of all implemented countermeasures are summarized in Table 1. Note that the data structure column in the table only considers the structures used in the AES procedure.

5. Related Works

5.1. Error-dectecting Codes

Karri et al. proposed a parity-based countermeasure for substitution-permutation network (SPN) symmetric block ciphers [19]. The basic idea compares a carefully modified parity of the input plaintext with that of the output cipher. Despite the low overhead on performance and hardware resources, it fails when there are even-numbered bits flipped among the 128 bits input. Karpovsky et al. proposed a countermeasure based on checksum [20]. It divides the verification stage into two steps and verifies AES's linear and non-linear parts respectively. It can fairly detect multi-fault errors at a cost of 35% area overhead, yet a weak detecting ability of single-fault errors. They subsequently proposed a countermeasure based on a systematic nonlinear robust (n, k)-code [18], reducing the fraction of undetectable errors from 2^{-r} to 2^{-2r} as compared to the corresponding $(n, k)-$linear code (where $n - k = r$ and $k >= r$) at the cost of 50% area overhead. However, due to the complexity of the nonlinear code, it is rather hard to implement it in software and can incur a dramatic performance overhead.

5.2. Infective Computation

Infection countermeasures based on deterministic diffusion functions were proposed in [21][22] which strongly modify the resulting diffusion pattern if there was a fault injected, therefore preventing the exploitation of a faulty cipher. They are vulnerable to the differential fault attacks proposed in [23][24]. Lomné et al. introduced randomness and proposed an enhanced countermeasure using multiplicative masks [25] to protect against the aforementioned attacks. Similarly, Gierlichs et al. proposed an infection countermeasure based on additional operations and first introduced dummy rounds to randomly shuffle the redundant and cipher rounds [26]. However, Battistello et al. showed that both two countermeasures were flawed and the full AES cryptographic key can be retrieved with 2200 and 36 faults, respectively, on average [27]. Tupsamudre et al. [17] improved Gierlichs' countermeasure using independent randoms irrespective of the round in which the fault is injected. This countermeasure outperforms all aforementioned countermeasures from the security perspective and is implemented in this paper.

6. Conclusion

In this paper, we analyze the principle of DVFS fault attacks and implement several software countermeasures for AES based on redundancy, parity code, and infection calculation. We then analyze the security and performance overhead of the implemented countermeasures. The temporal redundancy-based scheme achieves good security and low-performance overhead while retaining simplicity to implement, yet remains vulnerable to verification-bypass attacks. Contrary to popular belief, the spatial redundancy-based scheme incurs a large overhead on memory space and causes severe performance degradation. The parity-based method is vulnerable to even-numbered bit-flips and occurs quite a few performance overheads compared to the temporal redundancy-based scheme. The infective computation-based scheme achieves the best security and is immune to verification-bypass attacks, at a cost of much higher performance overhead though.

References

[1] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Clkscrew: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, volume 2, pages 1057–1074, 2017.

[2] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[3] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. Voltjockey: A new dynamic voltage scaling-based fault injection attack on intel sgx. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(6):1130–1143, 2020.

[4] 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 *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.

[5] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1445–1461, 2020.

[6] Intel. Intel® processors voltage settings modification advisory, Dec 2019. Retrieved April 23, 2023, from https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html.

[7] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David 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.

[8] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[9] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[10] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology—EUROCRYPT'97: International Conference on the Theory and Application of Cryptographic Techniques Konstanz, Germany, May 11–15, 1997 Proceedings*, pages 37–51. Springer, 2001.

[11] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology—CRYPTO'97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17*, pages 513–525. Springer, 1997.

[12] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential fault analysis on aes. In *Applied Cryptography and Network Security: First International Conference, ACNS 2003, Kunming, China, October 16-19, 2003. Proceedings 1*, pages 293–306. Springer, 2003.

[13] Debdeep Mukhopadhyay. An improved fault based attack of the advanced encryption standard. *Africacrypt*, 5580:421–434, 2009.

[14] Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication: 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings 5*, pages 224–233. Springer, 2011.

[15] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri, and Vincenzo Piuri. Error analysis and detection procedures for a hardware implementation of the advanced encryption standard. *IEEE transactions on Computers*, 52(4):492–505, 2003.

[16] Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on computers*, 49(9):967–970, 2000.

[17] Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Destroying fault invariant with randomization: A countermeasure for aes against differential fault attacks. In *Cryptographic Hardware and Embedded Systems–CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings 16*, pages 93–111. Springer, 2014.

[18] Mark Karpovsky, Konrad J Kulikowski, and Alexander Taubin. Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard. In *International Conference on Dependable Systems and Networks, 2004*, pages 93–101. IEEE, 2004.

[19] Ramesh Karri, Grigori Kuznetsov, and Michael Goessel. Parity-based concurrent error detection of substitution-permutation network block ciphers. In *Cryptographic Hardware and Embedded Systems-CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings 5*, pages 113–124. Springer, 2003.

[20] Mark Karpovsky, Konrad J Kulikowski, and Alexander Taubin. Differential fault analysis attack resistant architectures for the advanced encryption standard. In *Smart Card Research and Advanced Applications VI: IFIP 18th World Computer Congress TC8/WG8. 8 & TC11/WG11. 2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS) 22–27 August 2004 Toulouse, France*, pages 177–192. Springer, 2004.

[21] Marc Joye, Pascal Manet, and Jean-Baptiste Rigaud. Strengthening hardware aes implementations against fault attacks. *IET Inf. Secur.*, 1(3):106–110, 2007.

[22] Michel Agoyan, Sylvain Bouquet, Jacques Fournier, Bruno Robisson, Assia Tria, Jean-Max Dutertre, and Jean-Baptiste Rigaud. Design and characterisation of an aes chip embedding countermeasures. *International Journal of Intelligent Engineering Informatics*, 1(3-4):328–347, 2011.

[23] Gilles Piret and Jean-Jacques Quisquater. A differential fault attack technique against spn structures, with application to the aes and khazad. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 77–88, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[24] Thomas Roche, Victor Lomné, and Karim Khalfallah. Combined fault and side-channel attack on protected implementations of aes. In *Smart Card Research and Advanced Applications: 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers 10*, pages 65–83. Springer, 2011.

[25] Victor Lomné, Thomas Roche, and Adrian Thillard. On the need of randomness in fault attack countermeasures-application to aes. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 85–94. IEEE, 2012.

[26] Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output. In *Progress in Cryptology–LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings 2*, pages 305–321. Springer, 2012.

[27] Alberto Battistello and Christophe Giraud. Fault analysis of infective aes computations. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 101–107. IEEE, 2013.