Fault Attacks on XEX Mode with Application to certain Authenticated Encryption Modes

Hassan Qahur Al Mahri, Leonie Simpson, Harry Bartlett, Ed Dawson, and Kenneth Koon-Ho Wong

Queensland University of Technology, George St, Brisbane 4000, Australia
hassan.mahri@hdr.qut.edu.au
{lr.simpson, h.bartlett, e.dawson, kk.wong}@qut.edu.au

Abstract. The XOR-Encrypt-XOR (XEX) block cipher mode was introduced by Rogaway in 2004. XEX mode uses nonce-based secret masks (L) that are distinct for each message. The existence of secret masks in XEX mode prevents the application of conventional fault attack techniques, such as differential fault analysis. This work investigates other types of fault attacks against XEX mode that either eliminate the effect of the secret masks or retrieve their values. Either of these outcomes enables existing fault attack techniques to then be applied to recover the secret key. To estimate the success rate and feasibility, we ran simulations for ciphertext-only fault attacks against 128-bit AES in XEX mode. The paper discusses also the relevance of the proposed fault attacks to certain authenticated encryption modes based on XEX, such as OCB2, OTR, COPA, SHELL and ElmD. Finally, we suggest effective countermeasures to provide resistance to these fault attacks.

Keywords: side channel analysis, fault attack, authenticated encryption, block cipher mode, XEX

Introduction

In 2004, Rogaway [17] described a new block cipher mode called XOR-Encrypt-XOR (XEX) that can be used with any block cipher. XEX is a nonce-based mode in which each message uses a different nonce. A sequence of secret masks Δ_i (also known as offsets) is obtained from the encryption of the nonce. A different mask from this sequence is XOR-ed with each message block both before and after the underlying block cipher algorithm is applied. If the mode does not apply the last XOR operation with the secret mask, then it is called XE mode.

XEX/XE modes can be used to provide Authenticated Encryption (AE) (i.e. provide simultaneously confidentiality and integrity assurance) with the benefit that the plaintext message is processed only once. This is an attractive feature of AE modes. The drawback of such modes is that the security depends on both the key (K) and the mask (L); revealing either of them will breach the security of the AE mode as a whole [14, 17].

Fault attacks [7] are active attacks that induce an error during the operation of a cryptographic system to extract information about internal values, such as the secret key or any secret variable. The first paper that used fault attacks against cryptographic protocols was published by Boneh et al. [7] to attack RSA. Since then, fault attacks have been widely used to attack many encryption schemes including DES [4] and AES [6, 16].

Different physical means can be used to induce a fault into a cryptosystem, including supplying a voltage glitch, clock tampering, inducing a laser beam or radiating an electromagnetic field. The induced fault can flip a bit, skip the execution of an instruction or destroy a memory cell.

The most powerful fault analyses are Differential Fault Analysis (DFA) and Statistical Fault Analysis (SFA). Differential Fault Analysis [4] requires some input to be encrypted twice; with a fault being induced in the last rounds of the second run. After that, the difference between the correct and faulty ciphertexts is used to retrieve the secret key. On the other hand, SFA [12] does not require correct and faulty ciphertext pairs. It requires only a collection of faulty ciphertexts to recover the correct key. However, SFA is not directly applicable unless two conditions are met: the inputs to the block cipher are different from each other, and the faulty ciphertexts are the direct outputs of the block cipher [11].

In XEX mode, the nonce-based masks act as a barrier to conventional fault attack methods. For DFA where two identical block cipher inputs are required to produce a pair of correct/faulty ciphertexts, such as [4, 5, 22], XOR-ing different nonce-based masks with the plaintext blocks prevents this. For SFA where each block cipher output is XOR-ed with a different secret mask, we do not have direct access to the block cipher outputs. Thus, neither DFA nor SFA can be applied directly. This research is motivated by this fact.

To the best of our knowledge, the most relevant recent work investigating fault attacks on XEX-based modes is by Dobraunig et al. [11]. This work is significant as it demonstrated the practical relevance of statistical fault attacks introduced in [12] to authenticated encryption modes. The work targeted, amongst others, the XEX-based AE modes, such as OCB, OTR, COPA, SHELL and ElmD [3]. Fault attacks on XEX-based modes in [11] require access to parts of the mode where the block cipher output is either known or XOR-ed with a constant-based secret mask. These attacks are not applicable when the masks are not constant-based. The authors performed fault-injection experiments on three real hardware platforms. They showed that the key can be recovered with a couple of faulty ciphertexts. Note that all the listed XEX-based modes in [11] use constant-based masks with the exception of OCB. Although OCB uses nonce-based masks, the attack targeted the XE part, not the XEX part, where the output of the block cipher was accessible.

In this paper, we take a different approach by targeting the XEX part that uses nonce-based secret masks. We propose fault techniques to either skip the masking effect or retrieve the value of the secret mask L. In either case, conventional fault attack techniques [4, 12] can then be used to recover the secret key. In the worst case, the entire key can be retrieved with a single additional fault as described in [1, 15, 18, 21]. In addition, if XEX is used as an AE mode, an attacker can breach the AE security by constructing forged messages [14, 17].

Unlike previous fault attacks on XEX-based modes, we stress that our approach targets the part of the mode where a direct application of existing fault attack techniques is not possible.

We ran several simulations on a PC to demonstrate the effectiveness of our attacks and to calculate their success rates. The simulations use 128-bit AES as the underlying block cipher operating in XEX mode. We did not perform hardware experiments in this work. However, we consider the fault models in this paper are well documented in the literature and have been shown to be practical in certain research papers, such as [2, 19, 20]; so can be applied as outlined in this paper.

We then investigated the applicability of our proposed techniques to certain authenticated encryption modes, including candidates in the ongoing CAESAR competition [3], such as COPA, ELmD, SHELL and OTR. Our attacks show that the masking function is a point of vulnerability. Hence, efficient alternative constructions for the mask updating function are suggested as countermeasures.

This paper is organised as follows: Sect. 1 defines the notation used and briefly describes the AES and XEX schemes. Sect. 2 describes an approach to eliminate the barrier posed by the nonce-based secret masks in XEX mode. The next section shows fault attacks on the last rounds of AES to retrieve the secret masks given ciphertext pairs only. Sect. 4 verifies the relevance of our proposed approaches to certain authenticated encryption modes and Sect. 5 investigates mechanisms to avoid the proposed fault attacks. The last section draws a conclusion. A practical example of a fault attack, experimental results and figures are presented in appendices A, B and C respectively.

1 Preliminaries

1.1 Basic Notations

The following notation will be used consistently throughout this paper:

K	: k-bit key used for the block cipher and mask initialisation.
n	: the block length in bits of the block cipher.
N	: the nonce that is changed for each message.
m	: the number of blocks in the plaintext message.
$M[i]$	: the i^{th} block in the plaintext message.
$C[i]$	: the i^{th} block in the corresponding ciphertext message.
$E(.)$	: the block cipher encryption function under the key K.
$s_{jk}^{i,(r),(o)}$	: the (j,k) byte in the encryption state of plaintext $M[i]$ after the operation o of round r where $j,k \in \{0,1,2,3\}$.
$K_{jk}^{(r)}$	: the (j,k) byte in the subkey of round r where $j,k \in \{0,1,2,3\}$.
L_{jk}	: the (j,k) byte in a nonce-based secret value where $j,k \in \{0,1,2,3\}$.
sbox[.]	: the AES substitution box that replaces a byte by another byte.
sbox^{-1}[.]	: the inverse of the sbox[.] operation.
$\|X\|$	: the length of the string X in bits.

$\mathrm{msb}_c(X)$	: the most significant c bits of X provided that $\lvert X\rvert \geq c$.
$\ll$	: logical left shift operation.
$\gg$	: logical right shift operation.
$\wedge$	: bitwise-and operation.
$\oplus$	: bitwise-exclusive-OR operation.

1.2 AES Description

In this section, we briefly describe the Advanced Encryption Standard (AES), and refer the reader to [8] for more technical details.

AES is a 128-bit symmetric block cipher that allows three key sizes: 128-bit, 192-bit and 256-bit. AES is an iterated cipher that consists of a number of similar rounds. The number of rounds in AES is 10, 12 or 14 depending on the key size respectively. Each round in AES consists of four fundamental operations:

– `SubBytes` (SB): This operation is a non-linear substitution that replaces each byte in the internal state $s_{ij}^{(r),(o)}$ with another according to a fixed 8×8 s-box.
– `ShiftRow` (SR): This operation changes the order of bytes within the same state where certain bytes are shifted cyclically-left by a certain number of steps.
– `MixColumn` (MC): This is a linear transformation of the four bytes in each column of the state matrix.
– `AddRoundKey` (AK): The state matrix is combined with a round key by a bitwise XOR operation.

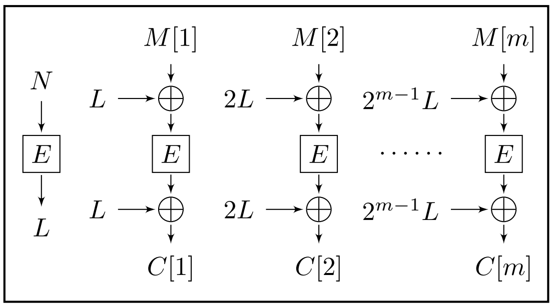
AES does not apply the `MixColumn` operation in the last round. The internal state after each operation for a plaintext block $M[i]$ is written as $s_{jk}^{i,(r),(o)}$ and organised as a matrix of 4×4 bytes where $0 \leq j < 4$ and $0 \leq k < 4$. For example, $s_{00}^{1,(9),(AK)}$ is the first byte of the encryption state of block $M[1]$ after `AddRoundKey` operation of round 9.

1.3 The Design of XEX Mode

Rogaway [17] introduced a mode of operation for block ciphers known as XEX. The underlying block cipher can be any symmetric block cipher. XEX mode is a nonce-based scheme where every plaintext message uses a different nonce (N). The nonce is encrypted to obtain a secret value $L = E(N)$. This secret value is used to obtain a sequence of secret masks $\{\Delta_i\}$ so that Δ_i is used during the processing of the i^{th} message block $M[i]$. For efficiency of implementation, every new mask Δ_{i+1} should be easily calculated from the previous one Δ_i. XEX mode uses a single key for both the block encryption operation and initialisation of the sequence of masking values.

In the example proposed in [17], Rogaway suggests a doubling masking technique, where new masks are obtained as $\Delta_{i+1} = 2\Delta_i$. If Δ_0 starts with L, the doubling masking technique results in a series of masking values: L, $2L$, 2^2L,

2^3L, ..., $2^{m-1}L$. Each message block uses a different mask that is XOR-ed both before and after the underlying block cipher algorithm is applied, as shown in Fig. 1(a).



```
    Input : $L$
(1) $Constant[2] = \{0x00,\ 0x87\}$
(2) $F = \mathrm{msb}_1(L)$
(3) $DL = L \ll 1$
(4) $DL = DL \oplus Constant[F]$
    output : $DL$
```

Fig. 1. (a) The most common masking of XEX mode. (b) Timing-resistant implementation of doubling masking technique.

The mask multiplication is performed in the finite field $\mathbb{F}_{2^n}$ by multiplying two input polynomials and finding the reminder modulo a primitive polynomial. When $n = 128$ and the finite field $\mathbb{F}_{2^{128}}$ is constructed using the commonly used primitive polynomial $f(x) = x^{128} + x^7 + x^2 + x + 1$, the doubling is as follows:

$$2L = \begin{cases} L \ll 1 & \text{if } \mathrm{msb}_1(L) = 0 \\ (L \ll 1) \oplus 0^{120}10000111 & \text{if } \mathrm{msb}_1(L) = 1 \end{cases} \tag{1}$$

and can be calculated as shown in Fig. 1(b). We note that this choice of finite field is used in Rogaway's paper [17] and has also been adopted in other designs such as COPA, ELmD, SHELL and OTR.

2 Eliminating the Masks in XEX Mode

This section presents two approaches to eliminate the effect of the secret masks, effectively converting the XEX mode to ECB mode.

2.1 Stuck-At-Zero Fault Attack

The duration of an injected fault can be permanent or transient. Permanent fault means that certain bits are disturbed permanently for the entire operation of a hardware platform, whereas transient faults change the value of certain bits temporarily. In addition to duration, the location of a fault can be either precise: affecting a certain bit in an internal register, or random.

Our fault model assumes that the fault will occur in a j-bit block anywhere in the secret mask register L except the last byte. The fault clears the block permanently, and could be performed using any practical physical means. That is, the j-bit in L, where $0 \leq j \leq 120$, will be stuck at 'zero' value permanently. We consider this fault model to be feasible using sophisticated technology, such as laser fault injection system and note that several research papers adopt this fault model (see fault attacks against AES in [6] and ACORN in [10]).

Due to the features of the primitive polynomial used in the doubling masking technique, the entire masking value L will be stuck permanently at zero after only a few faulted plaintext blocks in a multi-block message. If only one bit of L is faulted, L will reach zero after 128 blocks whereas if one byte is faulted, L will be zero only after 16 blocks. Therefore, the effect of masks in XEX mode is avoided.

The location of the j-bits cannot be between $121 \leq j \leq 127$ since these bit positions are XOR-ed with the feedback value (0x87) in the case where the most significant bit of L is 1 (see Equation (1)). Therefore, destroying such bit positions will not zero the mask L, but will increase the chance for mask collision.

Assuming a permanent fault is not necessary, the attacker can inject transient stuck-at-zero faults. This fault model has been shown to be feasible using low-budget equipment in [20]. The attacker can force certain bits of L to be 0 for few consecutive blocks. In case a byte is faulted, 16 consecutive set-to-zero faults are required to be induced to any of the first 15 bytes of L.

2.2 Skipping an Instruction Fault Attack

Assuming transient/permanent stuck-at-zero faults are not applicable or consume more time/cost, L can also be overcome in software implementations using a more efficient and easy to set-up fault model. L can be overcome using skipping an instruction, i.e. an instruction is not executed. An instruction can be skipped by applying glitch attacks [6]. This fault model was investigated and proved practical in [19].

One way to eliminate the effect of masks in XEX mode is to skip the execution of instruction (2) in Fig. 1(b) for 128 consecutive blocks. This step will cause the doubling mechanism to always choose $double[0]$ and not $double[1]$ provided that the value of F is zero before the fault injections. Thus after 128 blocks, the entire 128-bit L will be zero and L will be stuck to zero during the processing of all following blocks.

The chance that F is 0 before a fault injection is 50%. In case F was 1, we can repeat the attack a second time with another set of 128 consecutive blocks, but now with a better chance that this carry flag F is zero.

Another more effective way to overcome the masks is to omit the execution of instruction (4) (see Fig. 1(b)) for 128 consecutive blocks, regardless of the value of the carry flag F, as in the previous approach. This approach guarantees that the mask L will be stuck at a value of zero for all following plaintext blocks.

Note that this approach is implementation-dependant. That is, the attacker needs to have knowledge of the implementation to successfully overcome L.

2.3 Security Implication for Mask Elimination

Forcing the masks in XEX mode to zero reduces XEX mode to ECB mode. As a result, if the mode is used as an AE mode, this will enable attackers to breach the integrity assurance mechanism of the mode. In addition, the secret key can be recovered using additional faults. One extra fault injection can completely recover the key as described in [1, 15, 18, 21].

These proposed fault attacks are easy and efficient due to the particular form of the primitive polynomial used to define the finite field. Since the polynomial is sparse and the feedback path is from bits all located in the final byte of L, the attacks work effectively. This work demonstrates the weakness of this commonly used polynomial with respect to fault attacks. Sect. 5 suggests different primitive polynomials that avoid these fault attacks.

3 A Ciphertext Only Attack to Reveal Secret Mask L

In this section, we describe an approach to obtain the value of L under the stricter requirement of using ciphertext blocks only. For this section, we consider AES as the underlying block cipher used in XEX mode.

The challenge with attacking AES in XEX mode rather than the ECB mode is that the block cipher output is XOR-ed with a mask prior to generating the ciphertext. That is, the attacker does not have direct access to the output of the encryption. In addition, masks are guaranteed to be different from each other. A ciphertext-only statistical fault attack has been used previously to determine the secret key in AES encryption [12], but this attack requires a collection of ciphertext bytes that share the same subkey byte. Therefore, the statistical fault attack cannot be applied directly to obtain the secret key from AES in XEX mode. We show, however, that the relationship between the masks used in the doubling masking mechanism enables us to adapt this attack to reveal the initial mask value L. From this, it is then straightforward to find the secret key, completely breaking the security of the cipher. In fact, we note that the key bits can be determined using the same ciphertext bytes used to reveal the mask L.

As a first step toward retrieving L, we collect several masks that share certain mask bytes only. From Equation (1) and Fig. 1(b), we note that the doubling operation used in XEX mode causes the secret mask $L = (L_{00}, L_{01}, ..., L_{33})$ to shift by one bit to the left for each block in the message. Moreover, we note that the only bits of L that are potentially changed in this process are those in the final byte, L_{33}, of L. Thus, after eight shifts, all of the bytes in the original mask L - except for L_{00} and L_{33} - will appear again in 2^8L, but shifted a whole byte to the left. Likewise, all but three bytes of L will appear in $2^{16}L$ and the original byte L_{32} appears a total of fifteen times as:

$$\left\{ \begin{array}{c} L_{32}, (2^8L)_{31}, (2^{16}L)_{30}, (2^{24}L)_{23}, (2^{32}L)_{22}, (2^{40}L)_{21}, (2^{48}L)_{20}, (2^{56}L)_{13}, \\ (2^{64}L)_{12}, (2^{72}L)_{11}, (2^{80}L)_{10}, (2^{88}L)_{03}, (2^{96}L)_{02}, (2^{104}L)_{01}, (2^{112}L)_{00} \end{array} \right\}$$

before being shifted out of L. In addition, Equation (1) can be used to show that:

- there is a one-to-one relation between the values of $(2^8L)_{33}$ and L_{00}.
- the value of $(2^8L)_{32}$ depends only on L_{00} and L_{33} such that L_{33} can be determined uniquely from L_{00} and $(2^8L)_{32}$.

The first of these results effectively gives us a sixteenth copy of L_{32} from the mask byte $(2^{120}L)_{33}$. In total (for a sufficiently long message), we can have up to sixteen copies of L_{32}, as shown in Fig. 3 in Appendix C. (We have denoted $(2^{120}L)_{33}$ as L'_{32} in this figure.)

In some situations, 16 repetitions of the same byte are not sufficient to determine the byte's value with high probability. One way to overcome this problem is to increase the number of repetitions by using two bytes rather than one. For example, if we use the byte L_{32} and the byte $(2^8L)_{32}$, then each byte will repeat 16 times. In addition to these 32 repetitions, we can calculate the value of the byte $(2^{128}L)_{32}$ since we know both $(2^{120}L)_{33}$ and $(2^{120}L)_{00}$. This new byte $(2^{128}L)_{32}$ will also repeat another 16 times. In summary, each of the two bytes L_{32} and $(2^8L)_{32}$ will repeat 16 times and the combination of the two will repeat 16 more times. At the end, we have 48 occurrences depending only on 16 bits. The same concept applies if we take three bytes or more. In the case of three bytes (24 bits), we can have 96 repetitions in total.

In the following experiments, we consider two fault models as in [12]. The first model (we refer as fault model A) assumes that the attacker has a perfect control on the faulty byte. The fault induces a constant value. The second fault model (fault model B) assumes that the injected fault causes a bias to the targeted byte. Let e be error uniformly distributed in $[0, 255]$. The two fault models are as follows:

A. *Stuck-at-zero* with probability 1:

$$s_{jk}^{i,(r),(o)} = s_{jk}^{i,(r),(o)} \quad \text{AND} \quad 0 \quad \text{with probability } 1$$

.

B. *Stuck-at-zero* with probability 1/2:

$$s_{jk}^{i,(r),(o)} = \begin{cases} s_{jk}^{i,(r),(o)} & \text{AND} \quad 0 \quad \text{with probability} \quad 1/2 \\ s_{jk}^{i,(r),(o)} & \text{AND} \quad e \quad \text{with probability} \quad 1/2 \end{cases}$$

We will apply a fault attack to the internal state s at rounds 8 and 9 of the AES encryption operation. As addressed in [2], these fault models are possible, but for accurate value/location fault injections, high technical skills and high cost might be needed. However, [2] emphasises that the inability to inject only the desired fault does not imply the inability to induce the fault. In either case, our paper outlines the vulnerability of the XEX mode if these faults are possible.

3.1 Fault Model A at Round 9

Suppose that a fault is injected on a certain byte at the end of round 9 ($s_{jk}^{i,(9),(AK)}$). For example, a fault is induced on the byte $s_{00}^{i,(9),(AK)}$ for the first and second plaintext blocks ($i \in \{1, 2\}$) as shown in Fig. 4 in Appendix C.

The injected faults cause the two specified bytes to take a constant value. The faulty bytes will be identical during propagation in the `SubBytes`, `ShiftRow` and `AddRoundKey` operations of round 10 as follows:

$$\begin{aligned}
s_{00}^{1,(10),(SR)} &= s_{00}^{1,(10),(SB)} = \mathrm{sbox}[s_{00}^{1,(9),(AK)}] \\
s_{00}^{2,(10),(SR)} &= s_{00}^{2,(10),(SB)} = \mathrm{sbox}[s_{00}^{2,(9),(AK)}] \\
C[1]_{00} &= s_{00}^{1,(10),(SR)} \oplus K_{00}^{10} \oplus L_{00} \\
C[2]_{00} &= s_{00}^{2,(10),(SR)} \oplus K_{00}^{10} \oplus (2L)_{00} \\
C[1]_{00} \oplus C[2]_{00} &= L_{00} \oplus (2L)_{00} = (3L)_{00}
\end{aligned}$$

When we XOR $C[1]_{00}$ and $C[2]_{00}$, the two bytes $s_{00}^{1,(10),(SR)}$ and $s_{00}^{2,(10),(SR)}$ cancel each other since they are identical, and we obtain the value of $(3L)_{00}$. Note that our attack is based on the XOR of two consecutive blocks to obtain $(3L)_{00}$ and there is no need to find the subkey byte (K_{00}^{10}).

Repeating the above experiment for blocks $i \in \{9, 10\}$ will retrieve the byte $(2^8L)_{00} \oplus (2^9L)_{00} = (2^8 3L)_{00}$ which is equivalent to $(3L)_{01}$ (the second byte of $3L$). Similarly, block $i \in \{17, 18\}$ will enable us to determine the third byte of $3L$, and so on. Thus, by inducing faults in 32 blocks of a cipher in XEX mode we can retrieve the whole $3L$ mask, and consequently, we can easily obtain the original mask L. (To determine the final byte $(3L)_{33}$ it is necessary to adjust for the known value of $(3L)_{00}$.)

3.2 Fault Model A at Round 8

Assume that the fault is injected on a full diagonal at the end of round 8. For example, we inject a fault to the state $s_{jk}^{i,(8),(AK)}$ where $i \in \{1, 2\}$ as shown in Fig. 5 in Appendix C. The diagonal consists of four bytes that can have jk indexes as: $\{00, 11, 22, 33\}$, $\{01, 12, 23, 30\}$, $\{02, 13, 20, 31\}$ or $\{03, 10, 21, 32\}$. Injecting faults to a full diagonal seems infeasible; however, in software implementation running on 32-bit CPUs, a fault to one instruction can distribute to four bytes (see for example [9]).

In this case, we have one `MixColumn` operation. Hence, we need to know one full column of the internal state in order to reverse the `MixColumn` operation. Again the injected faults make the four bytes in the diagonal a constant value and they will remain identical through round 9 and 10. XOR-ing the ciphertext blocks will retrieve four bytes of $3L$ mask. For instance, if faults are injected into the diagonal $\{00, 11, 22, 33\}$, then the bytes $\{(3L)_{00}$, $(3L)_{13}$,$(3L)_{22}$, $(3L)_{31}\}$ will be retrieved.

Repeating the experiment with another faulty diagonal for plaintext blocks $i \in \{9, 10\}$, will retrieve four bytes of the mask $2^8(3L)$. However, note that these retrieved bytes are each shifted one byte to the left of the bytes in the mask $3L$. Hence, four diagonal fault injections can retrieve the original mask L completely. That is, in total, we need 8 faulty blocks where each block has a faulty diagonal.

3.3 Fault Model B at Round 9

Unlike fault model A, the fault induced by fault model B does not give a fixed output. Thus, we need to collect several faulty bytes that share the same mask byte and apply a statistical fault analysis with a distinguisher in order to predict the correct value for this mask byte from all possible hypothetical values. As discussed in Sect. 3, with reference to Fig. 3 in Appendix C, the position of the target byte will move through the various locations in the mask $2^{i-1}L$ as subsequent blocks of the message are processed, so we will need to fault different bytes of the AES encryption operation for different message blocks. For our attack, we will use the hamming weight distinguisher, which chooses the hypothetical mask value that minimises the average hamming weight for the faulty stuck bytes.

Retrieving One Byte As in Sect. 3.1 and 3.2, we aim to obtain information on the content of the mask 2^i3L. If we retrieve 2^i3L completely, then we can easily calculate the original mask L.

Suppose that a fault is injected on a certain byte at the end of round 9 ($s_{jk}^{i,(9),(AK)}$). The byte index (jk) will vary depending on the value of the block index (i). The attack is performed in four steps as follows:

1. Collect ciphertext bytes that share two mask bytes $\{(2^iL)_{32}, (2^{i+8}L)_{32}\}$. At the beginning of Sect. 3 we have seen that a maximum of 48 faulty ciphertext bytes can be collected that share two mask bytes.
2. Collect another 48 faulty ciphertext bytes that share two mask bytes $\{(2^{i+1}L)_{32}, (2^{i+9}L)_{32}\}$ from blocks consecutive to the blocks in step 1. The index of each faulty byte in the first set is the same as the index of the corresponding faulty byte in the second set.
3. XOR the two faulty ciphertext bytes in each pair from consecutive blocks to eliminate the shared subkey byte and obtain only two mask bytes $\{(2^i3L)_{32}, (2^{i+8}3L)_{32}\}$, and their continued mask $\{(2^{i+128}3L)_{32}$.
4. Use the hamming weight distinguisher to predict the correct value for $\{(2^i3L)_{32}, (2^{i+8}3L)_{32}\}$ and their continued mask $\{(2^{i+128}3L)_{32}$ from 2^{16} possible candidates.

An example of this process is presented in greater detail in Appendix A.

Retrieving All Bytes Extending this attack to determine the remaining mask bytes requires careful manipulation of the fault injections. The timing of consecutive fault injections is critical to the success of the attack. If the process to recover a byte after a fault injection is still in progress, injecting a subsequent fault will cause multiple faults in the internal state. This makes the attack impractical. However, allowing a delay after recovering the first byte before injecting the subsequent fault results in the first retrieved mask byte being shifted out of the internal state. The retrieved byte is no longer useful.

All of the bytes in a mask can be obtained by performing fifteen consecutive iterations of fault injections with the appropriate timing as discussed above. This means that any internal state contains at most two faulty bytes. Most modern devices come with 16-bit or 32-bits registers which makes faulting two bytes at a time feasible. This approach is discussed in detail at the end of Appendix A.

We simulate the proposed fault attacks in Sect. 3.3 using 128-bit AES. The attacks are developed in C on a standard desktop computer. We compute the success rate over 1000 iterations using different plaintext messages and nonces. We find that 96 faulty ciphertext bytes are enough to allow one byte in the secret mask to be retrieved with a success rate of 99.9%. Under the same conditions, the attack can be extended to recover the entire 128-bit secret mask with a success rate of 99.2%. For details of these simulations refer to Appendix B.

4 Application to Authenticated Encryption Modes

We examined AE schemes that use the doubling masking technique including the candidates of the ongoing CAESAR competition: OTR, COPA, ELmD and SHELL; and other AE modes, such as ISO 19772 OCB2 [17]. All of these AE block cipher modes use the masking technique of XEX/XE mode.

A summary of the relevance of our techniques against the secret masks in these authenticated encryption modes is presented in Table. 1. The (✓) mark indicates that the fault attack technique in the corresponding section of our paper can be applied to the mode, whereas the ($\times$) mark indicates that our technique can not be applied. Note that attacks in Sect. 3 cannot be applied to OTR as OTR is XE-based and not XEX-based.

The (*) symbol in Table. 1 indicates that the secret mask in these modes can be retrieved more effectively by direct application of SFA [11] than our technique in Sect. 3 since the masks are constant-based. Note that our attack is directly applicable to OCB2 whereas attacks in [11] are not.

Table 1. Summary of our attacks on secret masks in certain AE modes.

AE mode	Classification	Mask type	Our fault attack technique	
			Sect. 2	Sect. 3
COPA	XEX	Constant-based	✓	✓*
ELmD	XEX	Constant-based	✓	✓*
SHELL	XEX	Constant-based	✓	✓*
OCB2	XEX	Nonce-based	✓	✓
OTR	XE	Nonce-based	✓	$\times$

5 Countermeasures

The success of the fault attacks we have presented depends on the properties of the primitive polynomial used to construct the finite field for updating mask values in XEX mode. The polynomial used in Sect. 1.3 (also adopted by OCB2, COPA, ELmD, SHELL and OTR) is sparse and the feedback is obtained only from bits located in the final byte. Changing the mask updating function is one approach to prevent our attacks. We outline two alternative techniques for the mask updating function so that the proposed attack are not applicable.

The technique in the CAESAR candidate, OCB3, is an alternative option for updating masks which makes our attacks irrelevant. In OCB3 [13], although OCB3 still uses the doubling mechanism, masks depend on an index and each mask is XOR-ed with the prior one which prevents the repetition of mask bytes.

Another approach to preclude our attacks is to use a different function for incrementing masks. Krovetz and Rogaway [13] investigate several maximal 128-bit Linear Feedback Shift Registers (LFSRs); their internal states could be used as secret masks. An example of an efficient maximal LFSRs that has performance comparable to the doubling masking is:

$$S(X, Y) = (Y, (X \lll 1) \oplus (X \ggg 1) \oplus (Y \wedge 148))$$

where $|X| = |Y| = 64$. This LFSR does not include the most significant bit of the previous mask to increment the next one and does not allow repetition of mask bytes. Thus, using this LFSR for incrementing masks will avoid our attack.

6 Conclusion

The masking technique in XEX mode acts as a barrier to the fault attack methods commonly used to recover the secret key of the underlying block cipher. This paper presented different fault attack techniques against the generic XEX mode for block ciphers by targeting the secret masks used.

Firstly, we demonstrated three fault attack methods that convert XEX mode into ECB mode by forcing the secret mask L to zero. Injecting a permanent fault into a bit (or a byte) anywhere in the register containing the secret mask L, except for the final byte, will overcome the masking barrier after only 128 (resp. 16) blocks. This can also be achieved using transient faults on a few consecutive message blocks. For software implementations of XEX mode, we demonstrated that L can be eliminated through skipping instruction faults.

Secondly, instead of eliminating L, we provided a detailed ciphertext-only attack to retrieve L. The polynomial used in the doubling masking technique allows repetition of mask bytes. We used SFA with a collection of faulty ciphertext blocks to retrieve L bytes. Finding the secret mask enables retrieving the key using the same faulty blocks.

Thirdly, we verified the ciphertext-only attacks to retrieve L through simulations. In the case of fault model B, we found that the success rate of retrieving one byte of L is 99.9%, and that of retrieving the entire mask is 99.2%.

In addition, we identified certain authenticated encryption modes that are susceptible to our proposed fault attack techniques. These modes all used XEX with a primitive polynomial that makes them vulnerable to our attack.

Our work demonstrates that it is the mask updating function that makes XEX vulnerable to these fault attacks. Hence, an efficient solution to preclude these attacks is to change this primitive polynomial used for updating the mask.

References

1. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. IACR Cryptology ePrint Archive 2004, 100 (2004)
2. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. Proceedings of the IEEE 100(11), 3056–3076 (2012)
3. Bernstein, D.J.: Cryptographic competitions: CAESAR (2014), `http://competitions.cr.yp.to/caesar-submissions.html`
4. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: CRYPTO. Lecture Notes in Computer Science, vol. 1294, pp. 513–525. Springer (1997)
5. Blömer, J., Krummel, V.: Fault based collision attacks on AES. In: FDTC. Lecture Notes in Computer Science, vol. 4236, pp. 106–120. Springer (2006)
6. Blömer, J., Seifert, J.: Fault based cryptanalysis of the Advanced Encryption Standard (AES). In: Financial Cryptography. Lecture Notes in Computer Science, vol. 2742, pp. 162–181. Springer (2003)
7. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults (extended abstract). In: EUROCRYPT. Lecture Notes in Computer Science, vol. 1233, pp. 37–51. Springer (1997)
8. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography, Springer (2002)
9. Dehbaoui, A., Mirbaha, A., Moro, N., Dutertre, J., Tria, A.: Electromagnetic glitch on the AES round counter. In: COSADE. Lecture Notes in Computer Science, vol. 7864, pp. 17–31. Springer (2013)
10. Dey, P., Rohit, R.S., Adhikari, A.: Full key recovery of ACORN with a single fault. J. Inf. Sec. Appl. 29, 57–64 (2016)
11. Dobraunig, C., Eichlseder, M., Korak, T., Lomné, V., Mendel, F.: Statistical fault attacks on nonce-based authenticated encryption schemes. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 10031, pp. 369–395 (2016)
12. Fuhr, T., Jaulmes, É., Lomné, V., Thillard, A.: Fault attacks on AES with faulty ciphertexts only. In: FDTC. pp. 108–118. IEEE Computer Society (2013)
13. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: FSE. Lecture Notes in Computer Science, vol. 6733, pp. 306–327. Springer (2011)
14. Minematsu, K.: Improved security analysis of XEX and LRW modes. In: Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 4356, pp. 96–113. Springer (2006)
15. Mukhopadhyay, D.: An improved fault based attack of the Advanced Encryption Standard. In: AFRICACRYPT. Lecture Notes in Computer Science, vol. 5580, pp. 421–434. Springer (2009)
16. Piret, G., Quisquater, J.: A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In: CHES. Lecture Notes in Computer Science, vol. 2779, pp. 77–88. Springer (2003)
17. Rogaway, P.: Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 3329, pp. 16–31. Springer (2004)
18. Saha, D., Mukhopadhyay, D., Chowdhury, D.R.: A diagonal fault attack on the Advanced Encryption Standard. IACR Cryptology ePrint Archive 2009, 581 (2009)

19. Schmidt, J., Herbst, C.: A practical fault attack on square and multiply. In: FDTC. pp. 53–58. IEEE Computer Society (2008)
20. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: CHES. Lecture Notes in Computer Science, vol. 2523, pp. 2–12. Springer (2002)
21. Tunstall, M., Mukhopadhyay, D., Ali, S.: Differential fault analysis of the Advanced Encryption Standard using a single fault. In: WISTP. Lecture Notes in Computer Science, vol. 6633, pp. 224–233. Springer (2011)
22. Yen, S., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Trans. Computers 49(9), 967–970 (2000)

Appendix A: Practical Example for a Fault Attack using Fault Model B

Retrieving One Byte We demonstrate the fault attack in Sect. 3.3 with the following example:

Steps [1-2] We first collect $(2 \times 48 = 96)$ faulty bytes in which 48 bytes share the two mask bytes $\{L_{32}, (2^8 L)_{32}\}$, and the other 48 bytes share their consecutive mask bytes $\{(2L)_{32}, (2^9 L)_{32}\}$. These 96 bytes can be obtained using three sets: A, B and G, where each set contains 32 consecutive blocks (see Fig. 6). Set A shares the two mask bytes $\{L_{32}, (2L)_{32}\}$, set B shares $\{(2^8 L)_{32}, (2^9 L)_{32}\}$ and set G shares $\{(2^{128} L)_{32}, (2^{129} L)_{32}\}$. Remember that the mask bytes $(2^{128} L)_{32}$ and $(2^{129} L)_{32}$ are continued masks of $\{L_{32}, (2^8 L)_{32}\}$ and $\{2L_{32}, (2^9 L)_{32}\}$ respectively.

The targeted block indexes in each set are:
Set A:

$$i \in \left\{ \begin{array}{c} 1, 9, 17, 25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 105, 113, 121, \\ 2, 10, 18, 26, 34, 42, 50, 58, 66, 74, 82, 90, 98, 106, 114, 122 \end{array} \right\}$$

Set B:

$$i \in \left\{ \begin{array}{c} 9, 17, 25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 105, 113, 121, 129, \\ 10, 18, 26, 34, 42, 50, 58, 66, 74, 82, 90, 98, 106, 114, 122, 130 \end{array} \right\}$$

Set G:

$$i \in \left\{ \begin{array}{c} 129, 137, 145, 153, 161, 169, 177, 185, 193, 201, 209, 217, 225, 233, 241, 249 \\ 130, 138, 146, 154, 162, 170, 178, 186, 194, 202, 210, 218, 226, 234, 242, 250 \end{array} \right\}$$

The index (jk) of the faulty ciphertext byte corresponds to each block in every row of all sets A, B and G is:

$$jk \in \{ 32, 31, 30, 23, 22, 21, 20, 13, 12, 11, 10, 03, 02, 01, 00, 33 \}$$

However, to target each ciphertext byte, we inject a fault to its corresponding internal state byte $s_{jk'}^{i,(9),(AK)}$ where $k' = (k + j) \mod 4$. The faulty ciphertext byte indexes are not the same as the targeted internal byte indexes because of the last `ShiftRow` operation.

Step 3 XOR-ing two ciphertext bytes of the same index jk from two consecutive blocks will give the following result:

$$\begin{aligned} C[i+1]_{jk} &= s_{jk}^{i+1,(10),(SR)} \oplus K_{jk}^{10} \oplus 2^i L_{jk} \\ &= 2^i L_{jk} \oplus K_{jk}^{10} \oplus s_{jk'}^{i+1,(10),(SB)} \\ &= 2^i L_{jk} \oplus K_{jk}^{10} \oplus \text{sbox}[s_{jk'}^{i+1,(9),(AK)}] \\ C[i+2]_{jk} &= 2^{i+1} L_{jk} \oplus K_{jk}^{10} \oplus \text{sbox}[s_{jk'}^{i+2,(9),(AK)}] \\ C[i+1]_{jk} \oplus C[i+2]_{jk} &= (2^i 3L)_{jk} \oplus \text{sbox}[s_{jk'}^{i+1,(9),(AK)}] \oplus \text{sbox}[s_{jk'}^{i+2,(9),(AK)}] \end{aligned}$$

where $k^{'} = (k+j) \mod 4$. We calculate the value of $(C[i+1]_{jk} \oplus C[i+2]_{jk})$ from every two consecutive blocks in each set. This yields the following equations:
Set A:

$$C[i+1]_{jk} \oplus C[i+2]_{jk} = (3L)_{32} \oplus \text{sbox}[s_{jk'}^{i+1,(9),(AK)}] \oplus \text{sbox}[s_{jk'}^{i+2,(9),(AK)}]$$

Set B:

$$C[i+1]_{jk} \oplus C[i+2]_{jk} = (2^8 3L)_{32} \oplus \text{sbox}[s_{jk'}^{i+1,(9),(AK)}] \oplus \text{sbox}[s_{jk'}^{i+2,(9),(AK)}]$$

Set G:

$$C[i+1]_{jk} \oplus C[i+2]_{jk} = (2^{128} 3L)_{32} \oplus \text{sbox}[s_{jk'}^{i+1,(9),(AK)}] \oplus \text{sbox}[s_{jk'}^{i+2,(9),(AK)}]$$

where the value of $(2^{128}3L)_{32}$ is uniquely determined by the values of $(3L)_{32}$ and $(2^8 3L)_{32}$, as discussed previously. Each set gives 16 values for $(C[i+1]_{jk} \oplus C[i+2]_{jk})$, and in total 48 values.

The sbox in AES is resistant against differential analysis. Thus, knowing the XOR of $\text{sbox}[s_{jk'}^{i+1,(9),(AK)}]$ and $\text{sbox}[s_{jk'}^{i+2,(9),(AK)}]$ neither uniquely determines $s_{jk'}^{i+1,(9),(AK)}$ nor $s_{jk'}^{i+2,(9),(AK)}$. However, the injected faults will bias the faulty internal bytes to the all-zero byte. We, therefore, proceed by assuming that one of the faulty bytes is zero, namely that $s_{jk'}^{i+2,(9),(AK)} = 0$. This assumption is valid 50% of the time only. We then apply our statistical distinguisher to the value of $s_{jk'}^{i+1,(9),(AK)}$ that is determined by this assumption.

Step 4 For each of the 2^{16} candidates for $(3L)_{32}$ and $(2^8 3L)_{32}$, compute the value for $s_{jk'}^{i+1,(9),(AK)}$ in sets A, B and G. By completing this step, we will have 48 values for $s_{jk'}^{i+1,(9),(AK)}$ for each of the 2^{16} candidates. Use the hamming weight distinguisher to predict the correct value for $(3L)_{32}$ and $(2^8 3L)_{32}$ and their continued mask $(2^{128}3L)_{32}$.

Retrieving All Bytes This attack requires a message of at least 3722 full blocks encrypted using 128-bit AES in XEX mode. Note that this attack does not require all the 3722 blocks to be faulted. The steps to retrieve the entire mask L (see Fig. 7 in Appendix C) are as follows:

1. For blocks $(1 \leq i \leq 250)$, perform the fault attack, as in Sect. 3.3 and the example at the beginning of this appendix, on certain blocks (shown as orange bytes in Fig. 7) to retrieve the two mask bytes $\{(3L)_{32}, (2^8 3L)_{32}\}$ and their continued byte $(2^{128} 3L)_{32}$.
   Note that the byte $(2^{128} 3L)_{32}$ continues to appear as $(2^{136} 3L)_{31}, \cdots, (2^{240} 3L)_{00}$, $(2^{248} 3L)_{33}$.
2. For blocks $(1 + 248j) \leq i \leq (250 + 248j)$ where $j \in \{1, \ldots, 15\}$, perform the same attack in step 1. Each iteration is shown in Fig. 7 as successive coloured bytes as yellow, blue, pink, ..., lime.
   Each iteration retrieves extra bytes and starts just after the previous one.
3. Work backwards to calculate the mask bytes and begin with the last faulty block (shown as lime in Fig. 7). Use Equation (1) during the transition from one iteration of faults to the previous iteration. For example, we can calculate $(2^{3464} 3L)_{33}$ (grey byte) from $(2^{3464} 3L)_{00}$ (pink byte) and $(2^{3472} 3L)_{32}$ (lime byte). That is, the byte retrieved in the last iteration is not lost.
4. Repeat this approach working backward every 250-block iteration until we retrieve the entire $2^{128} 3L$.
   Note that any internal state contains at most two faulty bytes.
5. Compute the original mask L from $2^{128} 3L$ using the primitive polynomial.

Appendix B: Experimental Results

We ran a simulated experiment to retrieve one byte of a secret mask given faulty ciphertexts only and using the attack method in Sect. 3.3 and Appendix A. After that, the experiment is extended to retrieve the entire L mask. The experiment uses 128-bit AES as the underlying block cipher. We implemented this using the C language and the GNU GCC compiler run on a desktop computer.

To simulate fault model B, we used the pseudo-random C function *rand()* and the AES with different input messages to determine when the fault should occur. In either case, we used one bit of the output to determine when the stuck-at-zero action occurs. These generated faults are injected to AES in XEX mode.

Retrieving One Byte As a preliminary step, we performed several sub-experiments with different numbers of faulty bytes to determine how many faulty bytes are needed to obtain a high success rate. We started with 2 faulty bytes that share one mask byte and increase by 2 for every following iteration till the number of faulty bytes is 32 such that every targeted block has only one faulty byte. For each iteration, we computed the success rate over 1000 simulations using different plaintext messages and nonces. We performed these experiments twice: one run uses faults generated from the *rand()* function and the second uses faults from the output of AES in XEX mode.

Secondly, we computed the success rate for the attack using 96 faulty bytes, as described in Sect. 3.3 and Appendix A. This approach provides the hamming weight distinguisher with 48 faulty bytes that share only two mask bytes.

The results of this experiment are presented in Table 2 for the number of faulty bytes ranging from 2 to 32 and lastly 96. Note that the success rates in both columns are close to each other. Note also that only the last row provides a high success rate of at least 99.9%.

Table 2. Success rate of fault attacks using fault model B at round 9.

Number of faulty bytes	Success Rate (1000 iterations) *rand()* as PRF	Success Rate (1000 iterations) AES as PRF
2	0.280	0.289
4	0.233	0.238
6	0.405	0.377
8	0.467	0.472
10	0.528	0.556
12	0.637	0.631
14	0.673	0.703
16	0.753	0.744
18	0.787	0.790
20	0.821	0.839
22	0.873	0.861
24	0.885	0.888
26	0.909	0.901
28	0.930	0.929
30	0.940	0.936
32	0.957	0.956
96	0.999	0.999

Finally, we evaluated the success rate to retrieve one mask byte considering a more relaxed injection probabilities (p) to bias the faulty byte towards zero. Fig. 2 compares the success rate and data complexity for $p \in \{0.5, 0.375, 0.25\}$. Note that the success rate is about 0.96 when $p = 0.5$ and about 0.87 when $p = 0.375$ in case of (1 faulty byte/block), and these probabilities increase to 0.999 and 0.975 respectively in case of (2 faulty bytes/block). That is, for low injection probabilities, if an attacker is able to fault more bytes per block, the success rate will increase.

Retrieve the Whole Mask We performed an experiment to demonstrate the success rate of retrieving the entire secret mask $2^{128}3L$ as discussed in Sect. 3.3 and Appendix A. Each byte is retrieved using 96 faulty bytes. The success rate is also computed over 1000 different plaintext messages each of length 3722 blocks and each with a different nonce. We found that the success rate to retrieve every bit in the mask $2^{2144}3L$ is 99.2% when using AES as the pseudo-random function, and 99.3% when using the *rand()* function.

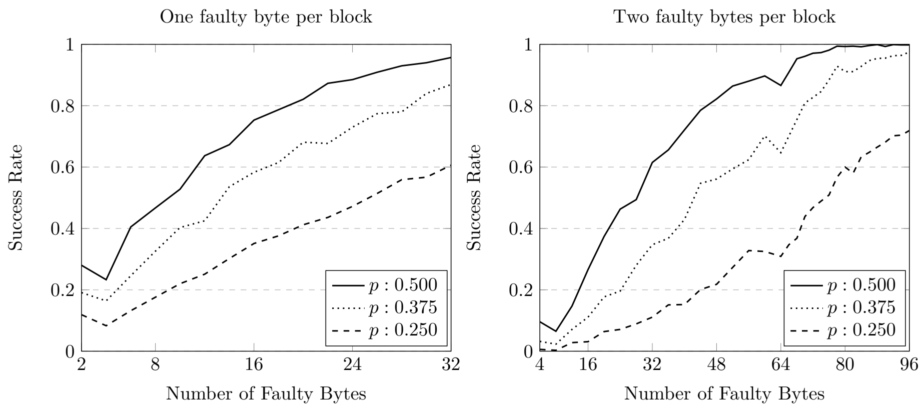


Fig. 2. Success rate to determine one mask byte for different probabilities.

Appendix C: Figures

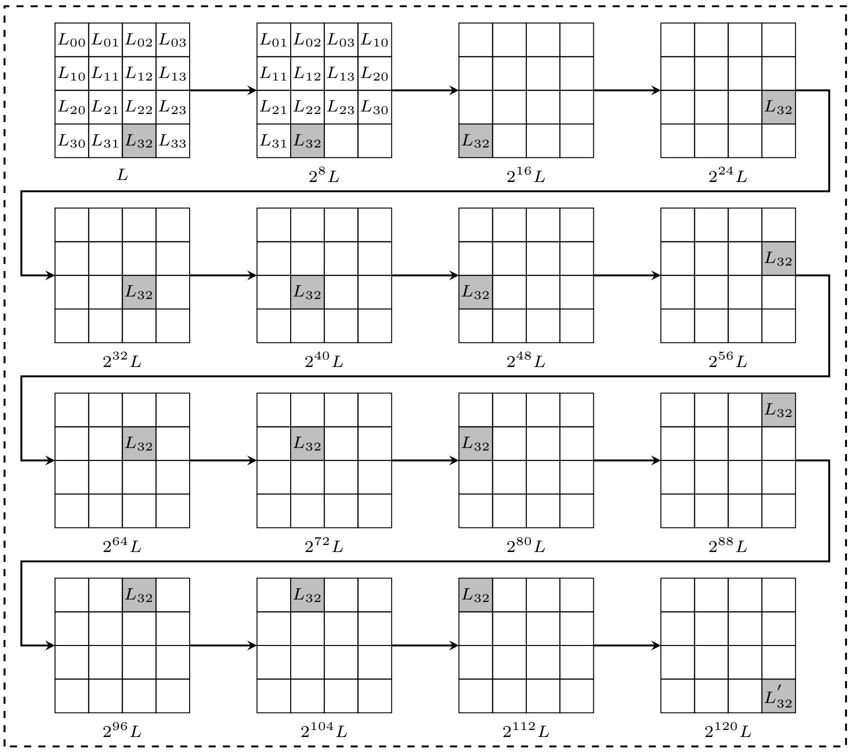


Fig. 3. Masks containing the byte L_{32}.

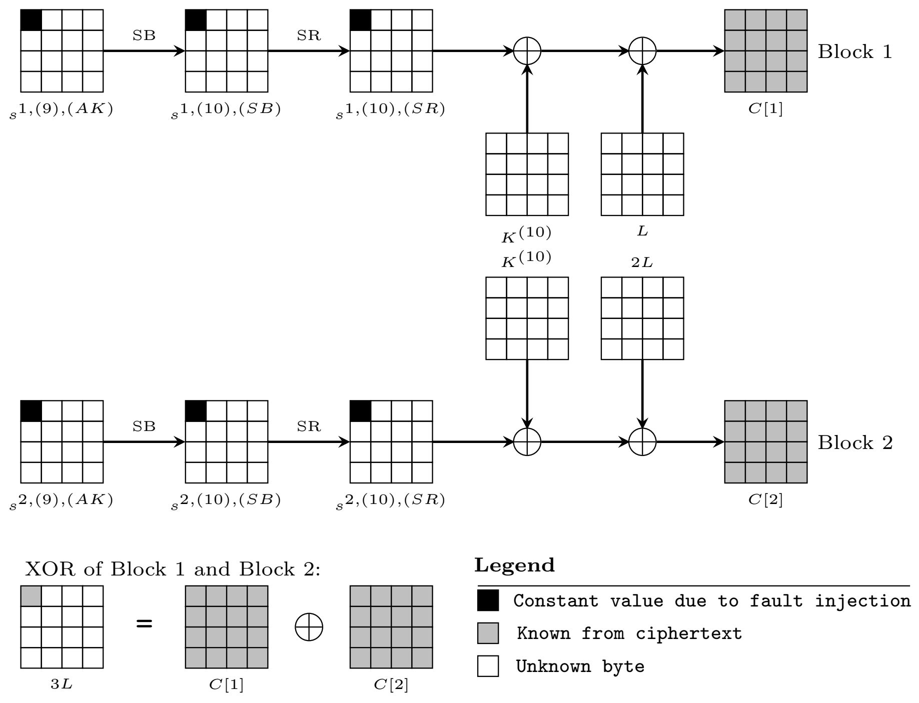


Fig. 4. Graphical representation of round 9 attack to retrieve the value of $(3L)_{00}$.

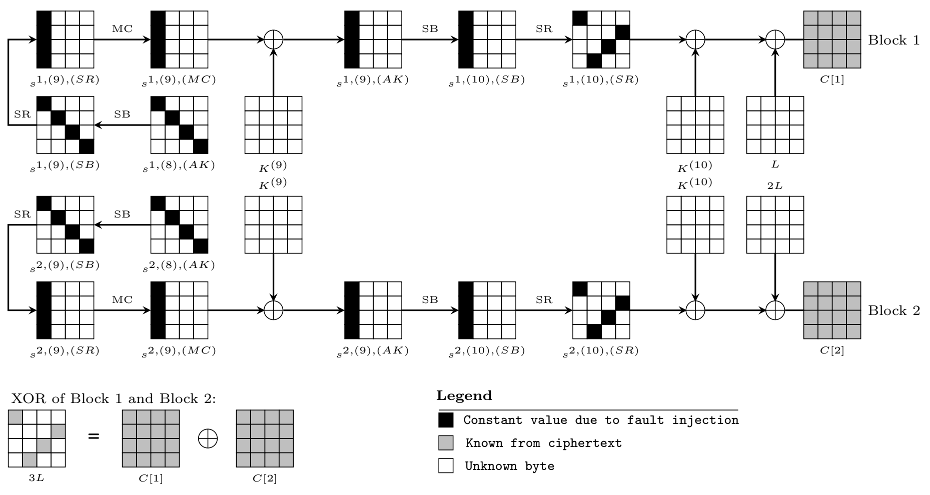


Fig. 5. Graphical representation of round 8 attack to retrieve four bytes in L.

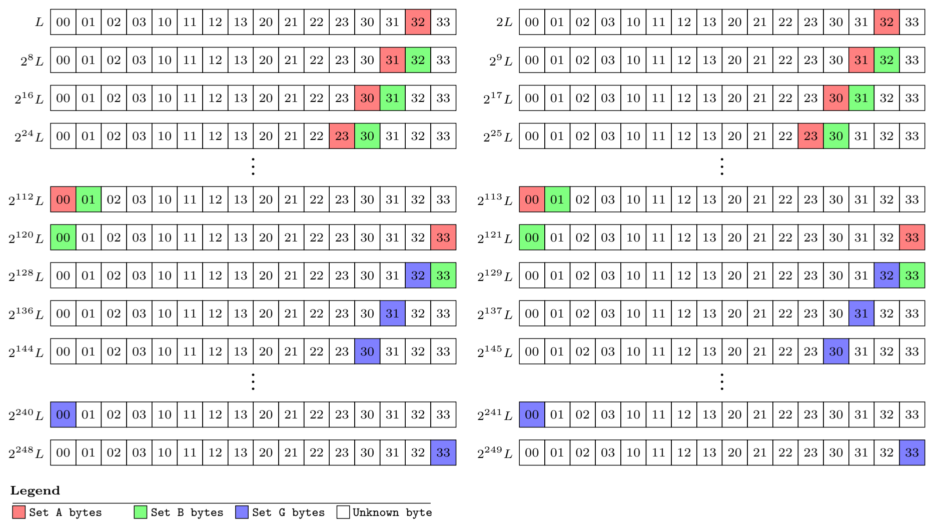


Fig. 6. Mask bytes targeted according to the position of faulty bytes.

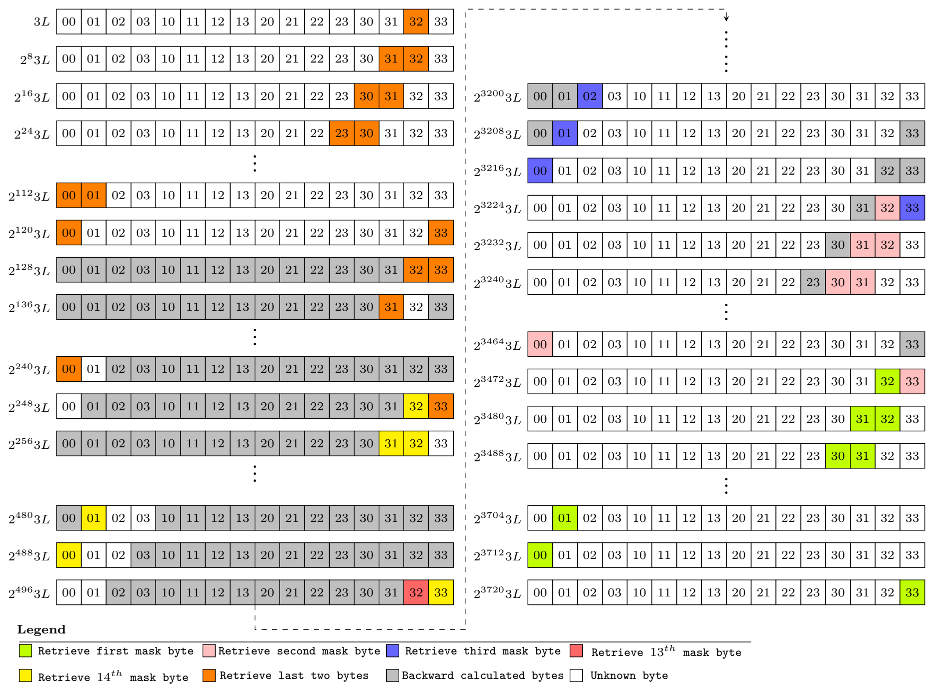


Fig. 7. Technique to retrieve all bytes of $2^{128}3L$.