Chapter 2
Attacking Block Ciphers

Christophe Clavier

Abstract Differential Fault Analysis (DFA) was one of the earliest techniques invented to attack block ciphers by provoking a computational error. In the basic DFA scenario the adversary obtains a pair of ciphertexts both of which encrypt the same plaintext. One of these ciphertexts is the correct result while the other is an erroneous one resulting from a faulty computation. Though applications of DFA to DES and AES have proven to be quite effective, other techniques have also been invented which can threaten block ciphers in different ways. This chapter presents some of these fault analysis methods, including Collision Fault Analysis (CFA) and its close variant Ineffective Fault Analysis (IFA). These methods depart from DFA by the fault model they rely on, by their ability to defeat classical countermeasures against DFA or DPA, or by their application to specific implementations.

2.1 Introduction

Differential Fault Analysis (DFA) was one of the earliest techniques invented to attack block ciphers by provoking a computational error. In the basic DFA scenario the adversary obtains a pair of ciphertexts that are the result of encrypting the same plaintext. One of these ciphertexts is the correct value while the other is an erroneous one resulting from a faulty computation. Since the two encryptions performed identically up to the point where the fault occurred, the two ciphertexts can be regarded as the outputs of a reduced-round iterated block cipher where the inputs are unknown but show a small (and possibly known) differential. By analyzing the propagation of this differential over a small number of rounds, the adversary seeks to obtain information about the key material involved in the last round.

C. Clavier (✉)
XLIM (UMR 6172), Université de Limoges, Limoges, France
e-mail: christophe.clavier@3il.fr

M. Joye and M. Tunstall (eds.), *Fault Analysis in Cryptography*,
Information Security and Cryptography, DOI: 10.1007/978-3-642-29656-7_2,

Though applications of DFA to DES [49] and AES [127, 160] as described in Sects. 3.3 and 4.2 have proven to be quite effective, other techniques have also been invented which can threaten block ciphers in different ways. This chapter presents some of these fault analysis methods, which depart from DFA by the fault model they rely on, by their ability to defeat classical countermeasures against DFA or DPA, or by their applicability to specific implementations.

An important class of such attack methods is referred to as Collision Fault Analysis (CFA) and has a close variant referred to as Ineffective Fault Analysis (IFA). The first section of this chapter is devoted to the presentation of these two techniques and their successive usages ranging from a first trivial CFA/IFA on unprotected implementations of AES-like algorithms to more elaborate attacks which can either attack implementations protected against both DFA and High-Order DPA, or even reveal the key of a so-called externally encoded DES whose precise specification is unknown to an attacker. Other fault attacks on block ciphers are presented in the second section, including some that explicitly aim to reduce the number of rounds, and others which exploit a perturbation of the initial randomization of substitution tables in DPA-resistant implementations.

2.2 Attacks on Block Ciphers by Exploitation of Identical Outputs

2.2.1 Three Resemblant but Different Fault Analysis Methods

2.2.1.1 Collision Fault Analysis

While DFA exploits a differential between a genuine and a faulty ciphertext where the fault occurred in one of the last few rounds of an encryption function E, CFA gains information from a collision event where the two ciphertexts C and $C^{\lightning}$ respectively obtained from a normal and a faulty encryption are equal. An attacker typically first obtains the faulty encryption $C^{\lightning} = E^{\lightning}(M_0)$ of an arbitrary plaintext M_0, and then searches for some particular M which gives the same ciphertext $E(M) = C = C^{\lightning}$ without any fault. Intuitively, it may appear difficult to produce a collision between two outputs of a cryptographic function designed to closely behave as a random function. This difficulty is circumvented by producing the fault very early in the encryption process in order to avoid the avalanche effect between the encryptions of M and M_0, and by relying on a specific fault model.

One usually adopts a fault model where the fault has a predictable effect on a small portion—a bit or a word—of the intermediate result. Such a classical fault model assumes that a fault occurring during a logical or an arithmetic operation, e.g. an XOR between two bytes, produces a result equal to some constant value

(typically considered to be zero), whatever the input values.[1] When produced at the very beginning of the algorithm a fault will typically corrupt an intermediate value that only depends on one bit or one byte of the input. As the remainder of the intermediate value is not modified, the attacker just has to change that precise bit or byte until a collision with the corrupted value occurs. All other intermediate data being uncorrupted, this local collision propagates to the ciphertexts.

2.2.1.2 Ineffective Fault Analysis

As described above, a CFA attack consists of searching for a plaintext whose corresponding ciphertext collides with some corrupted ciphertext. IFA is slightly different, and applies where an attacker tries to find an input M with the property that when a fault is induced on some precise operation during the encryption process, the intermediate data targeted by the fault is not corrupted, resulting in an identical ciphertext. This kind of analysis gains information from faults which do not locally modify the intermediate result, so-called ineffective faults from which the analysis name is derived.

While usually relying on the same kind of fault model, CFA and IFA differ in many respects. While CFA recovers some piece of information about the key with only one fault, IFA needs to compare pairs of ciphertexts ($C = E(M), C^{\lightning} = E^{\lightning}(M)$) until $C = C^{\lightning}$, so many faults[2] are required before an ineffective one is obtained. On the other hand, the higher complexity of IFA in terms of the number of required fault injections is compensated for by the property that the operation targeted by IFA does not need to occur near the beginning of the algorithm. Since the attacker compares pairs of executions with the same input, any operation during the encryption process can be targeted to identify an ineffective fault. In light of the fact that a fault appears to be ineffective if and only if the natural[3] result of the targeted operation is zero, under the previously described fault model, it is clear that IFA can be considered as a kind of probing tool. Indeed, for any given plaintext it is possible to decide whether the result of any arbitrary targeted operation[4] is zero or not.

Another property specific to IFA is that an attacker does not actually require the value of any faulty ciphertext. The only information that is required is whether the fault had an effect or not. As a consequence, IFA is not thwarted by the classical countermeasure against DFA, which consists in checking the computation and withholding the output if a fault is detected. Indeed, whether a fault is detected or not

[1] An exception to this usually adopted fault model for CFA is given by the collision/differential fault attack from Hemme in the first rounds of DES, described in Sects. 2.2.3 and 3.5. This attack is applicable even if the fault produces a random modification of an S-box output.

[2] For instance, under a byte-oriented fault model, 128 (or 256) faults are required on average per ineffective fault event when the input of the targeted operation is chosen (or not chosen).

[3] I.e. without fault.

[4] Provided that this operation is susceptible to the considered faults.

provides an attacker with the same information as whether or not the fault corrupted the ciphertext.

Finally, note that IFA relies on the fact that the attacker can safely decide that a fault attempt has been ineffective because of the natural value of the targeted data rather than because the fault did not occur. When performing IFA, it is thus very important that the fault injection tool be considered highly reliable.

2.2.1.3 Safe-Error Analysis

A third fault analysis method which exploits the identity of outputs of cryptographic algorithms is the analysis of so-called safe-errors, which we refer to as safe-error Analysis (SEA). This method was first introduced to break a private exponentiation of the RSA cryptosystem [204, 427, 429]. The basic principle consists of modifying some internal data and inferring the value of a private exponent bit from whether this modification resulted in an identical or a different output. Two examples of such analysis on a binary square and multiply exponentiation are: (i) the perturbation of a multiplication in an *always multiply* version of the left-to-right binary exponentiation algorithm, and (ii) including a dummy register in a right-to-left version of the algorithm. In both cases, depending on the particular value of the current exponent bit, the modified data may or may not be used in the sequel of the computation. The latter case corresponds to a safe-error which results in no modification of the algorithm output.

At first, ineffective fault analysis and safe-error analysis look quite similar. Indeed, they both infer information about a secret from whether an induced fault affected the output or not. Actually, there is a conceptual difference between IFA and SEA. In safe-error analysis the data that is targeted by the fault is actually modified, and the output is not modified simply because the modified data is not used. In contrast, an ineffective fault targets data involved in the computation of the algorithm, but because of the fault model and the data value, this data is not modified. In summary, IFA probes a data value while SEA probes a data usage.

A consequence of the difference between these two techniques is that IFA is highly dependent on the effect a fault has on the data being processed while SEA is not. A safe-error will be safe whatever the resulting value of the targeted data, so SEA is applicable in the general random error model. In contrast, ineffective fault analysis usually requires a more restrictive *stuck-at* fault model.

In the remainder of this section, we describe a series of research papers which consider CFA and IFA techniques to recover secret keys of block ciphers. We present them in the chronological order, which roughly corresponds to a trend of increasing ability to defeat countermeasures.

2.2.2 Bit-Wise Collision/Ineffective Fault Analysis on AES

The first attack from the CFA/IFA family was proposed by Blömer et al. [55], where they apply this technique to the AES algorithm. They assume a fault model where an attacker is able to force to 0 any chosen individual bit of an intermediate result. While they advocate its practicality, this fault model can be considered, according to the authors, as a rather strong one.

The principle of their attack is quite simple. An attacker first encrypts an all-zeros plaintext and obtains the corresponding reference ciphertext. Then for each arbitrary bit of the output of the first `AddRoundKey` operation, a faulty execution is performed where the fault sets this particular bit to 0. If the resulting ciphertext and the reference one are the same then the corresponding key bit is 0. If they differ the corresponding key bit is 1. By scanning all the bits of the first `AddRoundKey` output, the whole AES key is recovered with 128 fault injections.

Due to the bit-oriented nature of the fault model, this attack can be considered either as a degenerate form of CFA[5] or as an IFA. The computation checking countermeasure does not prevent the attack since only whether or not the fault corrupted the reference ciphertext is informative, not the value of the eventually corrupted result.

Interestingly, one can notice that this attack applies not only to the AES cipher, but more generally to the whole class of block ciphers whose very first operation, and the only one involving the plaintext, is an XOR with the key. Note also that the attack does not require the knowledge of the transformation subsequent to the first XOR with the key. It is thus applicable to any (even unknown) algorithm belonging to this class.

2.2.3 Collision Fault Analysis on DES

Designing a collision fault analysis on a Feistel network presents a specific difficulty. This is because, when encrypting a modified version of the initial plaintext M_0 used for the faulty execution, the input differential is involved in both the left and the right paths of the function. It is thus impossible to provoke a modification *only* on the input of one targeted operation and obtain a collision of intermediate results at the very beginning of the function.

Hemme dealt with this difficulty [178] by using the concept of characteristic, borrowed from differential cryptanalysis [46].[6] A characteristic is a set of prescribed differentials that an execution may follow from one round to another. Hemme considers only characteristics whose differential after $k = 2$, 3 or 4 rounds is the same

[5] The probability that the faulty ciphertext is not corrupted is as much as $\frac{1}{2}$.

[6] Hemme's attack can be regarded as a CFA since the adversary eventually finds a plaintext which encrypts to the same ciphertext as the faulty one. On the other hand it can also be considered as a kind of DFA since the adversary must analyze which differential characteristic could lead to the colliding message. The reader will find a more detailed description of this attack in Sect. 3.5.

as the differential that may have been produced by a fault. Assuming an attacker can modify the output Y_k of the kth round to $Y_k \oplus \varepsilon$ where ε belongs to a prescribed set defined by the fault model,[7] a pool of characteristics is created which includes, for each possible ε, the most probable characteristic ending after k rounds with a null differential on the left side and a differential equal to ε on the right side. After obtaining a faulty ciphertext, and for each ε, the attacker tries to exhibit a normal execution which follows the k-round $(0, \varepsilon)$-characteristic belonging to the pool. Provided that ε is the actual differential effect of the fault, a normal execution following the $(0, \varepsilon)$-characteristic on k rounds starts the $(k + 1)$th round with the same intermediate data as the faulty one, and results in a colliding ciphertext. Each obtained collision gives some information about the first round key K_1. This information is gathered by counting how many times each key candidate is suggested by a colliding pair. An improved version of the attack which makes use of an extended pool of characteristics can retrieve the whole K_1 with about 400, 1×10^4 and 5×10^6 faulty executions[8] when k is respectively equal to 2, 3 and 4.

One benefit of this attack is the possibility to attack the DES by fault analysis even when the implementation is protected against classical DFA by recomputing the last few rounds of a block cipher with a verification of the result. Note that this is an advantage only when the attacker does not have access to a decryption function that uses the same key. In the case where the attacker can both encrypt and decrypt any data of his choice, the redundancy of the last few rounds only is not sufficient to prevent DFA. Indeed, an attacker could then decrypt any input C while inducing a fault at the beginning of the algorithm (preferably during the second round) and obtain the faulty output M'. Then he can encrypt M' without fault and get C'. With respect to M', C' is a genuine ciphertext, while C is a faulty one analogous to what would have resulted from a classical DFA on the penultimate round.

2.2.4 *Defeating a DPA-Resistant AES by Collision Fault Analysis*

Amiel et al. presented several fault attacks which apply to DPA-resistant implementations of AES and DES [12]. In this section, we describe two of these that are collision fault analyses applied to AES.

[7] The author chose the set of 32 one-bit errors at the end of the round function. While this choice is naturally suited for a bit-oriented fault model, the author noticed that it also happens to be the best one for the less restrictive byte-oriented random error fault model.

[8] The attack also necessitates 4×10^4, 1.11×10^6 and 8.10×10^8 normal executions respectively.

2.2.4.1 Attacking the First `AddRoundKey`

A simple CFA attack on AES consists of an adaptation of the attack described in Sect. 2.2.2 to a byte-oriented fault model. Given the faulty ciphertext $C^{\lightning}$ produced by encrypting M_0, and producing a fault which forces to zero the output of the ith XOR operation in the first `AddRoundKey`, the attacker exhaustively encrypts all 256 plaintexts $M = (m_0, \ldots, m_{15})$ which coincide with M_0, except for the byte m_i, until one of the ciphertexts collides with $C^{\lightning}$. The plaintext byte m_i leading to a collision verifies that $m_i \oplus k_i = 0$, so the attacker decides that k_i is equal to this particular m_i. By scanning all 16 XOR operations, this attack allows a key to be completely recovered with only 16 faulty executions.

While quite efficient, this attack does not apply when the implementation is protected against first-order DPA by a Boolean masking scheme. In such implementations, all intermediate bytes are masked by an XOR with a random value R which is different from one execution to another, but which is the same from one byte to another. During the first `AddRoundKey`, each byte m_i of the plaintext is thus XORed with a masked key byte $\tilde{k}_i = k_i \oplus R$. When one tries to apply the above CFA on the result of the ith iteration of the `AddRoundKey`, the physical zero value forced on the XOR output actually corresponds to a logical value equal to R. While exhausting all m_i, the collision occurs when the logical XOR output is also equal to the random mask R of the faulty execution. This occurs for m_i such that $m_i \oplus k_i = R$, so the attacker erroneously decides that the key byte is equal to $k_i \oplus R$ instead of k_i. As the value R is unknown to the attacker, and different from one faulty execution to another, the attack fails.

Amiel et al. [12] devised a variant of the CFA that can break such DPA-resistant implementations of AES. The method is based on the observation from [21, 300] that a fault may allow an attacker to prematurely terminate a **for** loop before it has gone through all its iterations. If the memory location where the result of the XOR between the plaintext and the key is stored has not been used previously, then there are good odds that it is in an uninitialized state and contains bits all set to 0.[9] Now, suppose that one produces a fault two iterations before the end of the loop of the `AddRoundKey`. Then all 14 first result bytes are correct, but the last two are both have a physical value equal to zero, meaning a logical value equal to some random mask R that is identical for these two bytes. Encrypting all 2^{16} plaintexts where m_{14} and m_{15} take all possibilities will produce a collision for (m_{14}, m_{15}), verifying that

$$m_{14} \oplus k_{14} = R \quad \text{and} \quad m_{15} \oplus k_{15} = R.$$

An attacker will not know R but can nevertheless infer that

$$k_{14} \oplus k_{15} = m_{14} \oplus m_{15}.$$

The size of the key space is thus reduced from 2^{128} to 2^{120}.

[9] Or all set to 1 depending on the logical representation of the physical state.

By repeating the attack with the three last result bytes unaffected by the loop, the colliding ciphertext is obtained when logical values $m_{13} \oplus k_{13}$, $m_{14} \oplus k_{14}$, and $m_{15} \oplus k_{15}$ are all equal to a same unknown value R[10]

$$m_{13} \oplus k_{13} = R, \quad m_{14} \oplus k_{14} = R \quad \text{and} \quad m_{15} \oplus k_{15} = R.$$

Combining the first two equations now also reveals the value of $k_{13} \oplus k_{14}$.

Note that the exhaustive search on (m_{13}, m_{14}, m_{15}) does not cost 2^{24} encryptions but only 2^{16} since one can restrict the search to tuples of the form $(m_{13}, m_{14}, m_{14} \oplus \delta_{14,15})$ where $\delta_{14,15} = k_{14} \oplus k_{15}$ is known.

The attack continues repeatedly in the same way, each time revealing the value of $k_i \oplus k_{i+1}$ for $i = 12, 11, \ldots, 0$, ending with the knowledge of the complete key up to an XOR with a constant of the form $(c, c, \ldots, c)$. The correct value is identified by exhaustively trying the 256 remaining candidates. Overall the attack necessitates 15 faulty encryptions and $15 \times 2^{16} \approx 2^{20}$ normal encryptions.

A common complementary countermeasure against DPA consists in computing as much as possible in a random order. Note that the above attack can also be adapted to the case where the first `AddRoundKey` loop is executed in a random order. One needs to precompute a dictionary of approximately 2^{23} ciphertexts, which may be somewhat time consuming. A trade-off can speed up this precomputation by generating only a fraction of the dictionary at the cost of an increased number of faulty executions. An interesting practical property of the attack on the random order implementation is that the attacker does not need to change the instant of the perturbation from one fault injection to another.

Both attacks have been practically performed on smart cards. These experiments are described in detail in [400].

2.2.4.2 Attacking the Key Transfer from NVM to RAM

Prior to its use in the computation of a block cipher, each key byte must be XORed with the same random mask value R that is used to randomize any substitution table. This key masking with the value $(R, R, \ldots, R)$ commonly occurs when the key is transferred from nonvolatile memory (NVM) to RAM. For security purposes the key stored in NVM is also typically masked, where the mask is full size, meaning that every mask byte will be different, but static and diversified from one device to another. Denoting this full-size long-term mask by $r = (r_0, r_1, \ldots, r_{15})$, the masked key in NVM is equal to

$$\tilde{K}_{NVM} = (k_0 \oplus r_0, \ldots, k_{15} \oplus r_{15})$$

while the masked key in RAM is equal to

[10] Note that the value of R does not necessarily remain the same from one faulty ciphertext to another.

Algorithm 2.1: Key masking

Input: $\tilde{K}_{NVM} = (k_0 \oplus r_0, \ldots, k_{15} \oplus r_{15})$
$r = (r_0, r_1, \ldots, r_{15}), R$
Output: $\tilde{K}_{RAM} = (k_0 \oplus R, \ldots, k_{15} \oplus R)$

for $i \leftarrow 0$ **to** 15 **do**
  $\tilde{K}_{RAM,i} \leftarrow \tilde{K}_{NVM,i} \oplus R$ $\tilde{K}_{RAM,i} \leftarrow \tilde{K}_{RAM,i} \oplus r_i$
end
return $\tilde{K}_{RAM}$

$$\tilde{K}_{RAM} = (k_0 \oplus R, \ldots, k_{15} \oplus R) \ .$$

The mask conversion performed during the transfer follows Algorithm 2.1 with the difference that key bytes are actually processed in a random order. By faulting the loop so that only the first iteration has executed, the uninitialized memory space storing $\tilde{K}_{RAM}$ physically contains 15 bytes equal to zero and another one at a random index i equal to $k_i \oplus R$. This corresponds to a logical value of the form $(R, \ldots, R, k_i, R, \ldots, R)$, which is the actual key used in the faulty encryption of M_0.

The attack consists in precomputing a dictionary of about 2^{20} ciphertexts produced by the encryption of the reference plaintext M_0 under all keys of the previous form where i, k_i and R take all possible values. A search of the faulty ciphertext in the dictionary immediately reveals k_i. Each new faulty execution gives the opportunity to learn the value of a key byte at a different index. An average of only 54 faults allows all 16 key bytes to be determined. Note that the dictionary does not depend on the actual value of the attacked key.

2.2.5 Ineffective Fault Analysis on Externally Encoded DES

Contrarily to Kerckhoffs' principle, many applications of modern cryptography still adopt the *security by obscurity* paradigm. A particular way of designing a proprietary algorithm consists in surrounding a well-known and widely used block cipher $\mathtt{E}$ with two secret external encoding permutations $\mathtt{P}_1$ and $\mathtt{P}_2$ (one-to-one mappings over the input and output spaces respectively), leading to the new, secret, obfuscated block cipher $\mathtt{E}' = P_2 \circ \mathtt{E} \circ \mathtt{P}_1$. By basing the construction on a well-known block cipher $\mathtt{E}$ we allow the design to inherit proven or empirical cryptographic strength. Also, the two secret encodings $\mathtt{P}_1$ and $\mathtt{P}_2$ ensure that inputs to and outputs from $\mathtt{E}$ cannot be known by an attacker, so physical attacks requiring this knowledge should not be feasible.

A particular case studied by Clavier [93] depicted in Fig. 2.1 considers $\mathtt{E}$ instantiated as the DES function. Despite the impossibility of applying classical DFA [49],

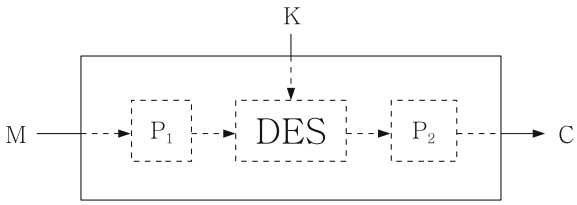

Fig. 2.1 A DES obfuscated by secret layers $\mathtt{P}_1$ and $\mathtt{P}_2$

which needs the output of the block cipher, and CFA [178], described in Sect. 2.2.3, which needs the control of the DES input, the author devised an ineffective fault analysis which recovers the secret key and applies to any member of the large class of unknown (to the attacker) cryptographic functions.

The attack assumes a classical software implementation of DES on an eight-bit architecture. Also, we assume an attacker is able to precisely control which instruction is executed when a fault is injected. As for other CFAs/IFAs the fault model assumes that a fault injected during the execution of an XOR between two eight-bit operands results in a zero[11] output whatever the input operand values were. Finally, the attacker is supposed to have control over the input given to the encryption function $\mathtt{E}'$ as well as knowledge of its output.[12]

The attack is somewhat complex and makes use of *pairs* of related ineffective faults. We now give a sketch of the principle of a basic version of the attack. As we assume an eight-bit architecture, there are 12 XOR operations per round: eight for the computation of the inputs of each S-box—denoted by $\mathtt{xor\_key}[j]\,(j = 1, \ldots, 8)$—and four others at the end of the round for combining its 32-bit output with the left part—denoted by $\mathtt{xor\_left}[i]$ $(i = 1, \ldots, 4)$. The attack intensively uses IFA to probe the output of these different eight-bit XOR operations that may appear at any round.

First, suppose that for some arbitrary plaintext M, a fault injected during some $\mathtt{xor\_left}[i]$ at round $(h-1)$ turns out to be ineffective (i.e. the ciphertext obtained with M by faulting this XOR is identical to the one obtained with the same input without fault). This implies that the corresponding output byte is zero. Thus, eight of the 32 bits at the input of the next round h are known to be 0. The subsequent permutation expands them to 12 bits, which are involved in four adjacent S-Boxes at round h as in Fig. 2.2. Now, suppose that for another execution with the same plaintext M, a fault on $\mathtt{xor\_key}[j]$ (for $j \in \{2i-1, 2i\}$) at round h also turns out to be ineffective. Then we know that the input x_j of this S-box is one of the four preimages of $S_j(0)$.[13] As x_j is the XOR between a key byte k_j and a six-bit value having five

[11] Note that the attack works equally well if the faulted XOR output is supposed to be any arbitrary known constant instead of zero.

[12] These assumptions may be relaxed, since an attacker only needs to be able to replay many different arbitrary inputs, and to detect whether two outputs are equal.

[13] As the S-box is a compressive function from six-bit inputs to four-bit outputs, any preimage of $S_j(0)$ behaves exactly as the input 0—which has been forced in the faulty execution—and thus produces the same ciphertext.

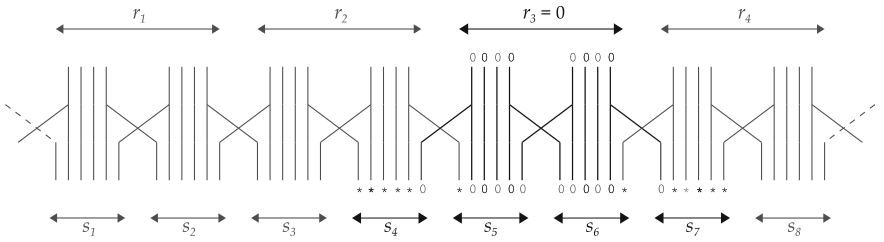

Fig. 2.2 A zero byte through the expansive permutation

bits known to be 0, we clearly see that k_j belongs to a set of only eight possible values, which reveals three bits of information about the key.

Overall, the attack repeatedly tries to detect a plaintext producing such a rare pair of related ineffective faults. As much information as possible is gathered by exploiting this kind of event on every possible loci—defined by a couple (`xor_left`[i], `xor_key`[j])—at every round. According to simulation results, the median residual entropy on the key is reduced from 56 to 26.49 bits (or 22.32 bits) after using 5×10^4 faults (or 1×10^6 faults). The author also described an improved version of this attack which gains information from more complex kinds of events. This improved attack achieves a median residual entropy of 13.95 bits (or 6.68 bits) after using 5×10^4 faults (or resp. 1×10^5 faults).

Compared to fault analysis on known cryptosystems, this attack requires a large number of fault injections. This can be considered as a fair price to pay for the 'magic' property of being able to retrieve the key of such unknown functions regardless of the two secret external encodings P_1 and P_2.

2.2.6 *Passive and Active Combined Attacks on AES*

The Boolean masking countermeasure that the two CFA attacks described above in Sect. 2.2.4 can bypass is only intended to protect against first-order DPA. State-of-the-art implementations must actually protect against high-order analyses, as introduced by Messerges [284]. The basic principle of such a countermeasure is to use a Boolean masking scheme based on one random byte that is different from one execution to another (as is the case for a first-order countermeasure), also from one S-box operation to another and from one round to another.

The recent attack presented by Clavier et al. in [94] actually succeeds in breaking even this kind of protection. The state-of-the-art implementation considered by the authors implements the S-box description from Oswald et al. [313] and is designed to resist HO-DPA attacks such as those presented in [11, 270]. The method used to break this implementation is a passive and active combined attack. The passive part of the attack consists in an adaptation of Correlation Power Analysis (CPA), originally described by Brier et al. [72]. As with the original analysis on unprotected

implementations, an attacker predicts, under some guess about a key byte, the series—one value for each power curve—of intermediate data manipulated during the AES computation. The difference with this classical scheme is that the targeted intermediate data is not the output of a first round S-box, but rather the random value by which this S-box is masked. Provided the value of this random mask can be predicted, under a key guess, for each execution, a CPA can be performed which will show a correlation peak for the correct guess at each instant the mask is manipulated. The need to identify, up to the corresponding key byte, the series of values taken by some random mask is precisely the reason why the passive part of the attack is preceded by an active phase which identifies the random series by means of a collision fault analysis.

As often, the target of this CFA is an output byte of the first `AddRoundKey` which is forced to zero by a fault. In the present implementation the plaintext M and the key K have been respectively masked, before they are XORed together, by two 16-byte randoms rm and rk respectively. Iteration i of the `AddRoundKey` thus computes $(m_i \oplus rm_i) \oplus (k_i \oplus rk_i) = (m_i \oplus k_i) \oplus r_i$ with $r_i = rm_i \oplus rk_i$ being the random value used to mask the ith S-box in the first round. As forcing to zero this XOR output is equivalent to introducing a differential equal to $\delta = m_i \oplus k_i \oplus r_i$, the attacker is able to identify $k_i \oplus r_i$, which is precisely the value for the ith plaintext byte that produces a collision with the faulty ciphertext. Thus, for each faulty execution, the attacker identifies the value $k_i \oplus r_i$ by means of a CFA, and stores the corresponding power curve. When sufficiently many $k_i \oplus r_i$ values and corresponding curves have been obtained, it is possible to perform a CPA where the series of r_i is predicted by simply guessing k_i. On average, 126 faults are enough to collect traces with 100 different r_i values, while 1,568 are required to perform the CPA with all 256 occurrences. Note that this attack is applicable even if the 16 XOR operations are computed in a random order. In that case the attacker has to search for the colliding ciphertext in an extended set of 2^{12} plaintexts where the input byte to modify can be in any position. The plaintext producing the collision automatically informs the attacker about the index i of the XOR corrupted by the fault. The attack strategy is slightly different since the different sets of power curves for all possible indices must be collected at the same time rather than successively.

The attack described above does not apply in the case where the computation verification countermeasure is implemented since no corrupted ciphertext is then given to the attacker. In that case it is still possible to adapt the attack by preferring an IFA strategy. When the plaintext byte m_i is set to the value u, each injected fault is an attempt to obtain a power curve for which $k_i \oplus r_i = u$. The probability of success of each attempt is 2^{-8} (or 2^{-12} in the case of random order), so the expected number of faults required to collect n power curves with different masks for each attacked key byte amounts to $2^{12} \times n$ (or $2^{16} \times n$). This is certainly a large number of faults, so this attack is hardly practical, but a door is opened here to break strongly protected implementations which include at the same time: high-order Boolean masking, computation verification and random order execution.

Note that an interesting and unexpected property of the ineffective fault variant is that it is easier to perform when a computation verification countermeasure is

implemented than when one is not. Indeed, when such a countermeasure is present the attack becomes a known plaintext attack instead of a chosen plaintext attack.

2.3 Other Fault Attacks on Block Ciphers

2.3.1 Reducing the Number of Rounds

Most block ciphers are *iterated cryptosystems*, which are families of cryptographically strong functions that iterate n times a weaker round function. As the security of the block cipher is an increasing function of the number of rounds, an obvious attack path could be to try to produce a perturbation in the normal sequencing which would reduce the number of rounds. This idea has originally been formulated in [15], where the authors suggest corrupting the appropriate loop variable or conditional jump by means of a glitch in either the clock or the power supply to the chip. In the following we give the descriptions of two concrete experiments which put this idea into practice.

2.3.1.1 Round Reduction Using Faults on AES

Choukri and Tunstall demonstrated in [89] that reducing the number of rounds of a block cipher is indeed possible. They used glitches on the power supply as the fault injection media, and conducted the attack on a Silvercard (`PIC16F877`) which does not contain any senors to protect against this sort of attack. The AES algorithm implementation used does not contain any countermeasure intended to prevent any sort of attack. The aim was to show that precise faults can be induced within a chip that can lead to the desired effect.

After a search among a wide range of combinations of relevant parameters such as the clock speed, the size of the glitch, the applied voltage and the time position in the computation of the AES,[14] various different configurations happened to be successful in inducing a fault that reduced AES computation to only one round. The exploitation of such a result is trivial and requires only two pairs of known plaintexts/ciphertexts. For each S-box an exhaustive search for the key byte value verifying

$$S(m_1 \oplus k) \oplus S(m_2 \oplus k) = \texttt{MixColumns}^{-1}(c_1 \oplus c_2),$$

leads to two expected hypotheses for each key byte, leading to an overall exhaustive search amongst 2^{16} keys.

[14] The identification of the correct time position has been helped by the capture of a power curve that clearly shows the different rounds.

2.3.1.2 A Case Study of Fault Attacks on Asynchronous DES Cryptoprocessors

Asynchronous circuits represent a class of circuits which are controlled not by a global clock but by the data. The class of circuit described by Monnet et al. [292] use so-called Quasi-Delay-Insensitive technology and multi-rail encoding, which are often thought to offer native resistance against faults. The authors designed two DES hardware implementations, a reference one and a hardened one, and studied their susceptibility to fault attacks using round reduction.

The round counter of the reference version is an asynchronous state machine that uses a 1-out-of-17 code: each round number is coded using one wire. Sixteen wires are used to encode the 16 rounds. The hardened version implements the same counter but protected with alarm cells. These cells are able to detect any wrong code generated in the counter module, i.e. any state using two or more wires is detected. Alarms inform the environment when a wrong code is detected.

Faults were induced using a laser beam, which offers the advantage of its directionality that allows the precise targeting of a small circuit area (e.g. $5\,\mu m^2$). In a time and space scan of the counter block which represented over 5^3 shots for each circuit, about 40 % of the shoots revealed errors. Some of them were identified as having modified the sequence of rounds.

From the properties of the DES key scheduling, the authors analyzed how a modification of the sequence of rounds alters the corresponding sequence of round keys actually used during the encryption. They give an example of the exploitation of a pair of faulty executions which led to close sequences of round keys. A detailed analysis of how to exploit these faults is given in [92], where how to recover the DES key is described for each possible case where the two sequences of round keys differ from each other by suffixes of length of at most 2. A significant number of the faulty pairs obtained fell into these exploitable cases. While the alarm cells of the hardened version actually detected most of the alterations of the counter block, a few executions with a modification of the round sequence remained undetected.

2.3.2 Corrupting the Randomization of a DES S-box

Before each execution of a DES protected by Boolean masking, all substitution tables are first randomized so that they comply with the original S-box tables in an implementation where all intermediate data are masked. This randomized S-box precomputation is typically performed as described in Algorithm 2.2. Assuming that an attacker is able to modify some S-box entry during this randomization phase, the subsequent DES execution could then be altered. Amiel et al. [12] precisely studied

Algorithm 2.2: S-box randomization

Input: The original S-box table $S = (s_0, \ldots, s_{63})_{16}$
The input and output masks $R \in (0, \ldots, 63)$ and $r \in (0, \ldots, 15)$
Output: The randomized S-box table $\tilde{S} = (\tilde{s}_0, \ldots, \tilde{s}_{63})_{16}$

for $i \leftarrow 0$ **to** 63 **do**
  $\tilde{s}_{i \oplus R} \leftarrow s_i \oplus r$
end
return $\tilde{S}$

how to exploit a modification of an S-box entry in both cases, where the index of the corrupted value is known and unknown to the attacker.[15]

2.3.2.1 Modifying Known S-box Values

When all eight S-Boxes are randomized as in Algorithm 2.2 an attacker can precisely know which S-box entry it is modifying. For some fixed arbitrary plaintext, it is possible to scan the randomization loop of some attacked S-box, induce a fault at each successive iteration, and observe whether the ciphertext has been corrupted or not. This results in the list of indices of all entries that are used for some S-box in one round or another of this execution. Such list contains an average of 14.255 indices which can all be taken as possible candidates for the S-box entries used in the first round. From the relevant bits of the plaintext, each index value can then be turned into a hypothesis on the S-box first round subkey. The different lists of candidates for each subkey can be further reduced by repeating the attack with one or more other plaintexts. The expected number of remaining candidates for the first round key K_1 is $2^{30.7}$ (or $2^{15.4}$) when the attack exploits one plaintext (or two plaintexts), which results in $2^{38.7}$ (or $2^{23.4}$) candidates for the whole key K.

In practice, embedded implementations of DES are unlikely to have the 512 four-bit S-box values written as separated bytes in memory. To avoid this waste of memory space, DES S-Boxes are usually compressed by storing the data into four 64-byte tables where the odd-numbered S-Boxes are stored in the high nibbles and the even-numbered S-Boxes are stored in the low nibbles.[16]

The number of key hypotheses generated by this attack against a DES using compressed S-Boxes is shown in Table 2.1 for different numbers of plaintexts. Since the number of faults required with two plaintexts when S-Boxes are compressed is the same as those required with only one plaintext when they are not, one can notice that the attack is more efficient on compressed S-Boxes. As an example, with 512

[15] Note that some figures given in [12] happen to be imprecise. The figures we give in the sequel to this section are borrowed from [92], where they have been more rigorously computed.

[16] While there are a few other ways in which the S-box data could be compressed, we do not consider this as something an attacker must previously know since he can conduct his attack under all possible compression methods until the correct one is found.

Table 2.1 Number of key hypotheses generated by attacking compressed S-Boxes

Plaintexts	Hypotheses per S-box pair	Hypotheses for the first round key	Whole key space
1	25.3	$2^{37.3}$	$2^{45.3}$
2	10.8	$2^{27.4}$	$2^{35.4}$
3	5.29	$2^{19.2}$	$2^{27.2}$
4	3.23	$2^{13.5}$	$2^{21.5}$

Algorithm 2.3: S-box randomization in random order

Input: The original S-box table $S = (s_0, \ldots, s_{63})_{16}$
The input and output masks $R \in (0, \ldots, 63)$ and $r \in (0, \ldots, 15)$

Output: The randomized S-box table $\tilde{S} = (\tilde{s}_0, \ldots, \tilde{s}_{63})_{16}$

for $i \leftarrow 0$ **to** 63 **do**
  $\tilde{s}_i \leftarrow s_{i \oplus R} \oplus r$
end
return $\tilde{S}$

faults, the whole key space is reduced to $2^{35.4}$ keys with compressed S-Boxes instead of $2^{38.7}$ with uncompressed S-Boxes.

Note also that the attack can be further slightly optimized by observing that, based on the faulty ciphertext, it is possible to identify cases where an S-box entry has been used only in one of the last two rounds. In these cases, the S-box entry should not be considered as a candidate for the first round subkey.

A countermeasure for this specific attack is to randomize the order in which the S-Boxes are randomized. This applies to the order in which the S-Boxes are treated, and to the order in which the S-box elements are masked. The data masking can be done as shown in Algorithm 2.3. The counter i is XORed with a random number before being used so the order in which the S-box elements are treated is unknown.

When only the order in which the S-Boxes are treated is randomized, one can still attack by searching for S-box indices that never change the ciphertext when modified. If the same S-box index is repeatedly changed, but the ciphertext never changes after numerous executions with the same plaintext, it can reasonably be assumed that this index value does not represent a key hypothesis for any part of the first round key. Faulting 22 times on the same index with no ciphertext modification ensures a probability of non-detection as small as 0.01. The expected number of indices that are used for at least one S-box in at least one round is 55.5. For each of them four faults are enough on average to show that this index is used. For the others the fault must be repeated 22 times. Thus an average of 409 fault injections are required for each plaintext. Table 2.2 presents the complexity of the resulting exhaustive search for different numbers of plaintexts.

Table 2.2 Number of key hypotheses generated by attacking compressed S-Boxes initialized in random order

Plaintexts	Hypotheses per S-box pair	Hypotheses for the first round key	Whole key space
1	55.5	$2^{46.4}$	$2^{54.4}$
2	48.2	$2^{44.7}$	$2^{52.7}$
3	42.1	$2^{43.2}$	$2^{51.2}$
5	32.5	$2^{40.2}$	$2^{48.2}$
10	18.5	$2^{33.7}$	$2^{41.7}$
15	12.4	$2^{29.1}$	$2^{37.1}$
20	9.71	$2^{26.4}$	$2^{34.4}$

2.3.2.2 Modifying Unknown S-box Values

If the attacker does not know which S-box index he has modified (for instance, if Algorithm 2.3 is actually used), then the previous attack does not work. In that case it is possible to derive an attack based on the exploitation of the event that the modified S-box value has been used only in the fifteenth round. This kind of event is easily identifiable from the normal and faulty ciphertexts by analyzing the differential at the end of the round.[17] Each time this event occurs it can be exploited by simply using the classical DFA method [49]. The probability of such event is 0.0123 when the S-Boxes are not compressed, and 0.0192 when they are. The number of faults needed for this attack is thus significantly larger than for a DFA where the fault directly targets the fifteenth round, but the advantage of the present attack is that it is not prevented by the computation verification countermeasure unless the S-box randomization is performed again between the two DES computations.

[17] It may happen that a corrupted S-box value used only in the fourteenth round is misinterpreted as being used only in the fifteenth round. We refer to [92] for a detailed analysis of these false positives which, as mentioned in [163], do not have much impact on the success of the attack.

http://www.springer.com/978-3-642-29655-0

Fault Analysis in Cryptography
Joye, M.; Tunstall, M. (Eds.)
2012, XVI, 356 p., Hardcover
ISBN: 978-3-642-29655-0