Fault Tolerant Infective Countermeasure for AES

Sikhar Patranabis, Abhishek Chakraborty, and Debdeep Mukhopadhyay

Department of Computer Science and Engg.
IIT Kharagpur, India
sikharpatranabis@gmail.com, abhishek_cky@yahoo.co.in, debdeep@cse.iitkgp.ernet.in

Abstract. Infective countermeasures have been a promising class of fault attack countermeasures. However, they have been subjected to several attacks owing to lack of formal proofs of security and improper implementations. In this paper, we first provide a formal information theoretic proof of security for one of the most recently proposed infective countermeasures against DFA, under the assumption that the adversary does not change the flow sequence or skip any instruction. Subsequently, we identify weaknesses in the infection mechanism of the countermeasure that could be exploited by attacks which change the flow sequence. We propose suitable randomizations to reduce the success probabilities of such attacks. Furthermore, we develop a fault tolerant implementation of the countermeasure using the x86 instruction set to make such attacks which attempt to change the control flow of the algorithm practically infeasible. All the claims have been validated by supporting simulations and real life experiments on a SASEBO-W platform. We also compare the performance and security provided by the proposed countermeasure against that provided by the existing scheme.
Keywords: Infective Countermeasure, AES, Randomization, Instruction Skip, Fault Attack, Fault Tolerant

1 Introduction

The demonstration of fault attacks by Dan Boneh *et.al* [1] on the RSA cryptosystem has triggered an extensive study of fault analysis with respect to all popular cryptosystems, including symmetric key systems such as the DES and the AES. Active Fault Analysis (FA) involves injection of faults into cryptographic systems and analysis under different fault models to retrieve the secret key. A multitude of fault attacks have been proposed in recent literature - some may require a pair of faulty and fault-free ciphertexts [2–7] while others may require only faulty ciphertexts [8]. Differential Fault Analysis (DFA) of AES exploits the relation between faulty and fault-free ciphertext pairs, and may require as few as a single fault injection to recover the entire key [9].

With fault attacks now being an established threat to the security of cryptosystems, sound countermeasures are needed to protect them. Recent research has demonstrated two major flavors of countermeasures - detection and infection. Detection countermeasures such as time and hardware redundancy [10,

11], that use the duplicate and compare principle, are vulnerable to attacks to the comparison step itself. Infective countermeasures avoid the use of comparison by diffusing the effect of the fault to render the ciphertext unexploitable. However, deterministic diffusion based infective countermeasures are vulnerable to attacks as demonstrated by Lomné *et.al* [12]. A random variation of the infective countermeasure was proposed by Gierlichs *et.al* [13]. However, the infection method employed by this countermeasure has a number of shortcomings, as demonstrated by Battistello and Giraud [14], and in greater detail by Tupsamudre *et.al* [15]. Tupsamudre *et.al* have also proposed an improved infective countermeasure that avoids all the pitfalls of [13] and thwarts DFA. However, no formal proof of security has been provided for the proposed scheme. Moreover, fault attacks that allow an adversary to change the flow sequence of an algorithm by methods such as instruction skips have also not been considered.

Recent research on microcontrollers and embedded processors has revealed that a fault model in which an attacker can skip an instruction is practically observable on various architectures [16, 17] using different fault injection techniques [18–20]. Hence, such a fault model is a realistic threat to embedded applications. We demonstrate in this paper that the instruction skip fault model reveals a major drawback of the infective countermeasure scheme proposed in [15]. In particular, it allows an adversary to disturb the number of cipher rounds being effectively executed. This reveals outputs of intermediate rounds, allowing easy key recovery. Thus, it is important to make any implementation of the infective countermeasure immune to instruction skip fault attacks. A formally verified software countermeasure against instruction skip attacks was proposed by Heydemann *et. al* [21] that provides a fault-tolerant replacement sequence for several instructions of the Thumb2 instruction set. The countermeasure scheme assumes that it is difficult to skip two instructions separated by few clock cycles.

Contribution: In this paper, we present a formal information theoretic proof of the security for the infective countermeasure scheme proposed by Tupsamudre *et.al* [15], under the assumption that an adversary cannot change the flow sequence or skip instructions. We then investigate in detail the threats posed to this countermeasure by the instruction skip fault model and incorporate necessary randomization in the existing algorithm to reduce the probability of an instruction skip fault attack. Finally, we propose a secured implementation of the augmented infective countermeasure scheme using x86 instructions that replaces compiler generated instruction sequences with fault tolerant ones. All the claims have been validated by supporting simulations and real-life experiments on a SASEBO-W platform that compare the different versions of the infective countermeasures both in terms of performance and security.

Table 1: Notations Used

$RoundFunction$	The round function of AES128 block cipher which operates on a 16 byte state matrix and 16 byte round key
S	The SubByte operation in the $RoundFunction$
SR	The ShiftRow operation in the $RoundFunction$
MC	The MixColumn operation in the $RoundFunction$
n	The total number of computation rounds (n = 11 for AES128)
t	The total number of rounds for the infective algorithm
I^i	The 16 byte input to the ith round of AES128, where $i \in \{0, \ldots, 10\}$
K	The 16 byte secret key used in AES128
k^j	The 16 byte matrix that represents $(j-1)^{th}$ round key, $j \in \{1, \ldots, 11\}$, derived from the main secret key K
β	The 16 byte secret input to the dummy round
k^0	The 16 byte secret key used in the computation of dummy round
$rstr$	A 't' bit random binary string, consisting of $(2n)$ 1's corresponding to AES rounds and $(t-2n)$ 0's corresponding to dummy rounds
$BLFN$	A Boolean function that maps a 128 bit value to a 1 bit value (0 input is mapped to 0; all other inputs are mapped to 1)
γ	A one bit comparison variable to detect fault injection in AES round
δ	A one bit comparison variable to identify a fault injection in dummy round
$\cdot$	A multiplication operation
$\wedge$	A bitwise logical AND operation
$\vee$	A bitwise logical OR operation
$\neg$	A bitwise logical NOT operation
$\oplus$	A bitwise logical XOR operation

2 Preliminaries

2.1 Notations Used

Table 1 summarizes the notations used in the rest of this paper. For the description of the infective countermeasure proposed by Tupsamudre *et.al* [15], we use the same notations used in the original paper.

The following points are to be noted :

1. In a $RoundFunction$, the SubByte, ShiftRow and MixColumn transformations are applied successively on the state matrix, followed by the KeyXor operation. AES128 has 10 rounds in addition to the initial Key Whitening step, which we refer to as the 0th round.
2. The 16 bytes $(m_0 \ldots m_{15})$ of the state and key matrices are arranged in 4×4 arrays and follow a column major order.
3. $BLFN$ maps 0 to 0 and all other nonzero inputs to 1.

We next explain in brief the countermeasure for AES128 proposed in [15].

2.2 The Infective Countermeasure

Algorithm 1 depicts the infective countermeasure proposed in [15] for AES128. In the event of a fault in any of the computation rounds (redundant or cipher), the algorithm detects the difference in values of R_0 and R_1 during the execution

of the cipher round. The value of R_0 is then set to R_2 as described in step 11 of the algorithm. If, on the other hand, the adversary attacks the dummy round, $(R_2 \oplus \beta)$ evaluates to 1 and R_0 is once again set to R_2. In the event of undisturbed execution, the algorithm outputs the correct ciphertext.

Algorithm 1 Infective Countermeasure [15]

Inputs : P, k^j for $j \in \{1, \ldots, n\}$, (β, k^0), $(n = 11)$ for AES128
Output : C = BlockCipher(P, K)

1. State $R_0 \leftarrow P$, Redundant state $R_1 \leftarrow P$, Dummy state $R_2 \leftarrow \beta$
2. $i \leftarrow 1$, $q \leftarrow 1$
3. $rstr \leftarrow \{0,1\}^t$ // $\#1(rstr) = 2n, \#0(rstr) = t - 2n$
4. while $q \leq t$ do
5. $\lambda \leftarrow rstr[q]$ // $\lambda = 0$ implies a dummy round
6. $\kappa \leftarrow (i \wedge \lambda) \oplus 2(\neg\lambda)$
7. $\zeta \leftarrow \lambda \cdot \lceil i/2 \rceil$ // ζ is actual round counter, 0 for dummy
8. $R_\kappa \leftarrow RoundFunction(R_\kappa, k^\zeta)$
9. $\gamma \leftarrow \lambda(\neg(i \wedge 1)) \cdot BLFN(R_0 \oplus R_1)$ // check if i is even
10. $\delta \leftarrow (\neg\lambda) \cdot BLFN(R_2 \oplus \beta)$
11. $R_0 \leftarrow (\neg(\gamma \vee \delta) \cdot R_0) \oplus ((\gamma \vee \delta) \cdot R_2)$
12. $i \leftarrow i + \lambda$
13. $q \leftarrow q + 1$
14. end
15. return(R_0)

In the following section, we present a information theoretic proof of the fact that, given the adversary cannot alter the number of executed rounds via instruction skip or affecting the state of the variables.

3 Information Theoretic Evaluation of the Infective Countermeasure

In Differential Fault Analysis (DFA), the adversary compares the response of a cipher with and without fault injection by obtaining both faulty and fault-free ciphertexts. The basic assumption underlying the DFA principle is that the differential of the correct and faulty ciphertext must contain some information about the secret key used in the algorithm. The adversary then infers the key by analyzing the fault propagation under the assumption of a fault model. However, if the differential provides no additional information about the key, then obtaining the faulty ciphertext does not give the adversary any advantage at all. Thus, the capability of a countermeasure scheme to thwart DFA, can be evaluated by the extent of **mutual information** between the differential and the key. The lesser the mutual information, the stronger is the countermeasure scheme.

In this section, we first describe in greater detail the aforementioned information theoretic security evaluation methodology for countermeasures. We then evaluate the infective countermeasure depicted in algorithm 1 using this framework and verify that the countermeasure indeed thwarts DFA successfully. Table 2 summarizes some of the notations used in this section.

Table 2: Notations Used

X	A discrete random variable
x_i	A specific value that X may take
$Pr(X = x)$ or $Pr(x)$	The probability that a random variable X takes a value x
$Pr(x\|y)$	The conditional probability that $X = x$ given $Y = y$
$H(X)$	The entropy of random variable X
$H(X\|Y)$	The conditional entropy of X given Y
$I(X\|Y)$	The mutual information of random variables X and Y
K	The secret key used by AES
Δ	The differential of the fault-free and faulty ciphertexts
N	The total number of possible values for K and Δ
$\{k_1, k_2, \cdots, k_N\}$	The sample space from which K takes its values
$\{\Delta_1, \Delta_2, \cdots, \Delta_N\}$	Sample space from which Δ can take its value

3.1 The Evaluation Methodology

Definition 1 *In information theory, the mutual dependency of two random variables can be measured using the concept of* ***mutual information****. The mutual information of two discrete random variables X and Y is defined as:*

$$I(X;Y) = H(X) - H(X|Y) \tag{1}$$

where $H(X)$ denotes the entropy of the random variable X and $H(X|Y)$ denotes the conditional entropy of X given Y .

Again, entropy and conditional entropy are represented using the following formulations:

$$H(X) = \sum_{i=1}^{N} Pr(x_i)\log(Pr(x_i)) \tag{2}$$

$$H(X) = \sum_{i=1}^{N}\sum_{j=1}^{N} Pr(y_j)Pr(x_i \mid y_j)\log(Pr(x_i \mid y_j)) \tag{3}$$

Using this information theoretic measure, we can compute how much mutual information a differential fault attacking technique provides in revealing the key, for a given fault model.

Let Δ is the fault observed at the output of a block cipher and K is the key used in the encryption. Then $I(K;\Delta)$ provides the mutual information between Δ and K. Using formulations 1, 2 and 3, we have :

$$I(K;\Delta) = \sum_{i=1}^{N}\sum_{j=1}^{N} Pr(\Delta_j)Pr(k_i \mid \Delta_j)\log Pr(k_i \mid \Delta_j) - \sum_{i=1}^{N} Pr(k_i)\log Pr(k_i) \tag{4}$$

where Δ and K can take values from the sets $\{\Delta_1, \Delta_2, \cdots, \Delta_N\}$ and $\{k_1, k_2, \cdots, k_N\}$ respectively.

Further, using Bayes' Theorem, we have :

$$Pr(k_i \mid \Delta_j) = \frac{Pr(\Delta_j \mid k_i)Pr(k_i)}{Pr(\Delta_j)} \tag{5}$$

For details of how to compute $Pr(k_i \mid \Delta_j)$ via simulation for a given fault model, please refer Appendix A.

Thus, using the information theoretic measure, one can evaluate the security of a countermeasure scheme against DFA. We next perform this analysis for the infective countermeasure in the forthcoming discussion.

3.2 Evaluating the Security of the Infective Countermeasure against DFA

Assumptions about the fault model: In the information theoretic evaluation of the security of the infective countermeasure, we make the following assumptions about the fault model:

1. The flow sequence of the algorithm, that is, the order in which the redundant, cipher and dummy computations are executed for various rounds is determined solely by the sequence of bits in $rstr$ and does not change during the course of execution of the algorithm via instruction skip or any other methodology.
2. The number of rounds of execution of the algorithm is not in any way affected, that is, we have exactly 11 pairs of redundant and cipher computations, with the redundant computation always preceding the cipher computation.
3. The values of internal variables and registers other than the state registers R_0, R_1 and R_2 are not updated except as required by the algorithm.

We now use the mutual information formalism to evaluate the security of the infective countermeasure proposed by Tupsamudre *et.al.* Before delving into a rigorous analysis, we make an important observation about the algorithm.

Observation 1 *In the event of a fault injection into a single round of the algorithm, the entire cipher state is affected and is in fact replaced by a random matrix* β *which is entirely independent of the key* K*.*

A single fault injection could occur in either a redundant computation round, or a cipher computation round, or a dummy round. The correctness of observation 1 can be easily verified by considering each scenario individually.

1. **Redundant round affected:** In this case, R_1 stores the faulty output after the redundant round computation. When the computation of the corresponding cipher round takes place, R_0 stores a value different from the current content of R_1. Hence $R_0 \oplus R1$ evaluates to 1 in step 9 of the algorithm. Hence γ is 1 and R_0 is replaced by β.
2. **Cipher round affected:** In this case, R_0 stores the faulty output after the original round computation, while R_1 stores the correct output. Hence $R_0 \oplus R1$ evaluates to 1 in step 9 of the algorithm. Hence γ is 1 and R_0 is replaced by β.

3. **Dummy round affected:** In this case, R_2 stores the faulty output after the original round computation, which is different from β. Hence $R_2 \oplus \beta$ evaluates to 1 in step 9 of the algorithm. Hence δ is 1 and R_0 is once again replaced by β.

In discussing the outcome of fault injection in all of the above scenarios, we have assumed single fault injection. Additionally, even if the adversary were to inject multiple faults, it could only go undetected if the same fault was introduced in a redundant-cipher round pair. However, the presence of random intermediate dummy rounds implies that even if the adversary had the ability to inject the same fault twice, she could never be sure which rounds to inject the faults into. This makes such an attack probability very low. It is also interesting to note that while fault injections in the cipher and dummy rounds are detected immediately in the same round itself, a fault injection in the redundant round is detected subsequently, during the execution of the corresponding cipher round.

Since the outcome of fault injection in any of the rounds is thus essentially the same, we present a common analysis for all three scenarios. *Assuming that the adversary cannot affect the number of rounds of computation*, a fault injection must be detected and the infection will occur. Consequently, the output differential is of the form $\Delta = C \oplus \hat{\beta}$, for fault injection into either the redundant, cipher or dummy rounds, C being the fault free ciphertext output and $\hat{\beta}$ being a random 128 bit matrix. Since $\hat{\beta}$ and K are independent random variables, we have:

$$Pr(\hat{\beta} = \hat{\beta}_k \mid K = k_i) = Pr(\hat{\beta} = \hat{\beta}_k) \tag{6}$$

Consequently , the conditional probability $Pr(\Delta_j \mid k_i)$ takes the form :

$$\begin{aligned} Pr(\Delta_j \mid k_i) &= Pr(\hat{\beta} = \Delta_j \oplus C \mid k_i) \\ &= Pr(\hat{\beta} = \Delta_j \oplus C) \\ &= Pr(\Delta_j) \end{aligned} \tag{7}$$

Formulations 5 and 7 together establish the conditional independence of K and Δ as:

$$Pr(k_i \mid \Delta_j) = Pr(k_i) \tag{8}$$

Substituting $Pr(k_i)$ in equation 4 yields the following:

$$\begin{aligned} I(K;\Delta) &= \sum_{i=1}^{N}\sum_{j=1}^{N} Pr(\Delta_j)Pr(k_i \mid \Delta_j)\log Pr(k_i \mid \Delta_j) - \sum_{i=1}^{N} Pr(k_i)\log Pr(k_i) \\ &= \sum_{i=1}^{N}\sum_{j=1}^{N} Pr(\Delta_j)Pr(k_i)\log Pr(k_i) - \sum_{i=1}^{N} Pr(k_i)\log Pr(k_i) \\ &= \sum_{i=1}^{N} Pr(k_i)\log Pr(k_i) - \sum_{i=1}^{N} Pr(k_i)\log Pr(k_i) \\ &= 0 \end{aligned} \tag{9}$$

Thus the countermeasure scheme ensures that the mutual information of the differential and the key is 0 and thus the adversary gains no information about the key once the infection affects the entire cipher state. However, an important assumption for this analysis was that *the adversary cannot in any way disturb*

the order in which the rounds are executed. If the adversary chooses to mount an attack that, instead of affecting the cipher state, *disturbs the round counter and prevents the infection from affecting the cipher state*, she could gain access to intermediate cipher state values and exploit it to decipher the key. In the next section, we show that the instruction skip fault model indeed allows such an attack that exploits the vulnerability of the round counter.

4 Threats to the Infective Countermeasure

We now look in detail at the threat posed to Algorithm 1 by the instruction skip fault model. We begin by looking at possible threats to the infective countermeasure other than traditional DFA attacks, on of which is the instruction skip attack. We next introduce in brief the instruction skip fault model, followed by a description of how the instruction skip fault model could be used by the adversary to disturb the number of executed rounds in Algorithm 1. Finally we focus on the loopholes in the algorithm that allow the adversary to mount such an attack.

4.1 Possible Attacks on the Infective Countermeasure : Affecting Flow Sequence

The formal proof of security of the infective countermeasure 1, presented in section 3, makes some assumptions about the fault model of the adversary. One of these is that the number of rounds and the order of their execution are not affected by the adversary. However, in a practical implementation of the infective countermeasure, the adversary could attack the round counter itself to try and upset the normal execution of the algorithm. As demonstrated in [22, 23], round reduction and fault round modification allow the adversary to obtain the key with a relatively small number of computations. Thus, although Algorithm 1 thwarts traditional DFA, it could be vulnerable to this flavor of attacks where the adversary could play around with the number of effectively executed rounds.

One of the major drawbacks of the infective countermeasure depicted in Algorithm 1 is that there is no validation check for the round counters q and i. If the value of either of these counters is affected by the adversary, the algorithm would not be able to detect the fault, which in turn would affect the order of round execution. There are many ways in which the adversary could inject such a fault. One approach is to affect the state of either the counter variables q and i, or other variables affecting them, such as λ. The stuck-at fault model makes such attacks practically feasible. Alternatively, the adversary could choose to simply skip the round counter updation step(s), that is, steps 12 and/or 13 of Algorithm 1. Such attacks come under the purview of the *instruction skip fault model*. In the forthcoming discussion, we look in greater detail at the threat posed by the instruction skip fault model, as well the loopholes in the infective countermeasure scheme that make such an attack possible.

4.2 The Instruction Skip Fault Model

The instruction skip fault model is a subset of the more general instruction replacement fault model, in which the adversary is able to replace one instruction by another. Previous research has shown that it is possible to perform instruction replacement on embedded processors by a variety of fault injection means [24, 18]. However, precise control over instruction replacement demands very accurate fault injection means and is not of much practical significance. However, a specific category of instruction replacement is the instruction skip fault model, in which the adversary replaces an instruction by another one that does not affect any useful register [21] and has the same effect as a NOP. Instruction skips have been achieved by a number of fault injection schemes on a variety of architectures - via clock glitches [16, 18] and electromagnetic glitches [19] on 8-bit AVR microcontroller, via voltage glitches on a 32-bit ARM9 processor [17] and via laser shots on a 32-bit ARM Cortex-M3 processor [20]. Hence, instruction skips are considered as a practically achievable fault model and have been used for cryptanalysis in recent research [25, 26].

We now look into how the adversary may use the instruction skip fault model to attack Algorithm 1.

4.3 Instruction Skip Attack on the Infective Countermeasure

The attack presented here exploits the fact that a redundant round computation in Algorithm 1 does not involve any infection to the state of the cipher R_0. The adversary targets skipping instruction 12 of algorithm 1 after the execution of the last redundant round. As a result of this attack, the final cipher round is replaced by another redundant computation round. Since a redundant round does not involve any infection and does not affect the output register R_0, the algorithm simply returns the output of the penultimate cipher round, that is the output of round 9. The adversary can then exploit this faulty ciphertext to recover the key by making hypotheses over each key byte.

Let e be the event that the adversary performs a succesful instruction skip in the nth redundant round by attacking the qth loop of algorithm 1. The probability $Pr(e)$ is thus the probability that the bit string $rstr$ has $(2n-2)$ positions set to 1 among the first $q-1$ positions, has the qth bit set and exactly 1 more bit set among the remaining $t-q$ bits, given by $\frac{\binom{q-1}{2n-2}\binom{t-q}{1}}{\binom{t}{2n}}$. Moreover, $Pr(e)$ could be further augmented to $Pr(e,r)$ by repeating the fault injection experiment independently r times, such that $Pr(e,r) = 1-(1-Pr(e))^r$. Note that $Pr(e,r)$ is essentially the probability of obtaining at least one useful faulty ciphertext in r fault injections.

4.4 The Loopholes in the Infective Countermeasure : A Closer Look

There are two major loopholes in the countermeasure that allow the adversary to mount the aforementioned attack :

Table 3: Computation of Algorithm 1

Step	Redundant Round	Cipher Round	Dummy Round
5.	$\lambda = 1$, i is odd	$\lambda = 1$, i is even	$\lambda = 0$
6.	$\kappa \leftarrow 1$	$\kappa \leftarrow 0$	$\kappa \leftarrow 2$
7.	$\zeta \leftarrow \lceil i/2 \rceil$	$\zeta \leftarrow \lceil i/2 \rceil$	$\zeta \leftarrow 0$
8.	$R_1 \leftarrow RoundFunction(R_1, k^\zeta)$	$R_0 \leftarrow RoundFunction(R_0, k^\zeta)$	$R_2 \leftarrow RoundFunction(R_2, k^0)$
9.	$\gamma \leftarrow 0$	$\gamma \leftarrow BLFN(R_0 \oplus R_1)$	$\gamma \leftarrow 0$
10.	$\delta \leftarrow 0$	$\delta \leftarrow 0$	$\delta \leftarrow BLFN(R_2 \oplus \beta)$
11.	$R_0 \leftarrow R_0$	$R_0 \leftarrow (\neg(\gamma) \cdot R_0) \oplus ((\gamma) \cdot R_2)$	$R_0 \leftarrow (\neg(\delta) \cdot R_0) \oplus ((\delta) \cdot R_2)$
12.	$i \leftarrow i + 1$	$i \leftarrow i + 1$	$i \leftarrow i + 0$
13.	$q \leftarrow q + 1$	$q \leftarrow q + 1$	$q \leftarrow q + 1$

1. An inherent drawback of the infective countermeasure is the inability to immediately detect a fault injection in the redundant round. The algorithm must wait until the corresponding cipher round in order to detect the presence of the fault. On the other hand, a fault injection in a cipher round or a dummy round is detected immediately. After a faulty redundant round, R_0 still contains the output of the previous round and is not infected. The phenomenon is made clear in the highlighted row of Table 3 that captures the execution flow of the algorithm in the redundant, cipher and dummy rounds respectively.
2. The execution of the redundant round is merely decided by the fact that the variable i is odd and $\lambda = 1$. There is no way to verify if the redundant round being executed is indeed a valid one. This makes the round counter vulnerable to attacks by a malicious agent who can manipulate the value of the internal variables, as done in the aforementioned attack via an instruction skip, and trick the algorithm into believing that the round to be executed is a redundant one. This allows the adversary to skip the final cipher round, and thus avoid fault detection and infection altogether.

An interesting observation about Algorithm 1 is that the order of the redundant and cipher rounds is fixed. For a given round, the redundant computation always precedes the cipher computation. This is because both the redundant and the cipher rounds are denoted by a set bit in the bit vector $rstr$ and are distinguished by the oddity of i. The randomness is thus limited to the occurrence of the dummy rounds in between, represented by the 0's in $rstr$. Thus the adversary is guaranteed to obtain the output of the penultimate round if she can skip the last cipher round and replace it by a redundant round. If, on the other hand, the relative ordering of the redundant and cipher computations corresponding to a single round could also be randomized and the output masked, then the adversary would have to perform additional instruction skips to get the unmasked output of the penultimate round. In the following section, we present a modified version of Algorithm 1 that achieves this randomization.

5 A Modified Infective Countermeasure

In this section we present a modified infective countermeasure algorithm. The idea is to reduce the probability of the instruction skip attack by making it more

uncertain as to whether the fault is introduced in the redundant or cipher round of computation. Unlike in the original scheme where the redundant round always precedes the corresponding cipher round, in the modified version, the order of the redundant and cipher rounds is scrambled and is encoded by an additional bit string *cstr* of length $2n$. Each 1 bit in *cstr* corresponds to a redundant round and each 0 bit corresponds to a cipher round. Since cipher and redundant pair of computations are still necessary for each round, *cstr* is a sequence of $(1, 0)$ and $(0, 1)$ pairs. The *cstr* vector may be populated by randomly filling out the odd positions with 0 or 1 and then setting each even position to the negation of its preceding odd position. Additionally, in the modified algorithm, both R_0 and R_1 are masked at the end of each odd computation round and unmasked at the beginning of the corresponding even computation round. The mask m is a 128 bit vector and is generated randomly at the beginning of each odd computation round. Algorithm 2 details the steps of the modified countermeasure, while Table 4 summarizes the functioning of algorithm 2.The major differences between algorithms 1 and 2 are summarized below:

1. In algorithm 1 no infection occurs during the redundant round since the redundant round always occurs prior to the cipher round. On the other hand, in algorithm 2, the infection occurs (upon fault detection) in the round that occurs later, which may be either cipher or redundant, depending on the content of *cstr*. This makes the treatment of the redundant and cipher rounds more symmetric.
2. In algorithm 2 both R_0 and R_1 are masked in the end of an odd round and unmasked in the beginning of the corresponding even round computations. This ensures that neither R_0 nor R_1 exposes the output of the previous round after the end of an odd computation round.
3. In algorithm 1 between each pair of redundant and cipher computations, R_0 retains the unmasked output of the previous round, which could be exploited by an adversary. On the other hand, in algorithm 2, R_0 holds the masked output of a previous round only if the bit pair in *cstr* corresponding to the current round is $(1, 0)$, the probability of which is $\frac{1}{2}$.

Note: The formal proof of security presented for algorithm 1 in Section 3 also holds good for algorithm 2 under the same assumptions that the attacker cannot alter the flow sequence or skip instructions.

A security analysis for the bit string *cstr* is presented in Appendix B. We now analyze the impact of the instruction skip fault model on algorithm 2 as well as corresponding attack probabilities.

5.1 Instruction Skip Attack on the Modified Algorithm

We now analyze in greater detail the probability that the adversary can still mount the same instruction skip attack on algorithm 2 and obtain the output of the penultimate round. Note that the adversary would have to skip step 13 of algorithm 2, which corresponds to the increment of the variable i. Since the

Algorithm 2 Modified Infective Countermeasure

Inputs : P, k^j for $j \in \{1, \ldots, n\}$, (β, k^0), $(n = 11)$ for AES128
Output : C = BlockCipher(P, K)

1. State $R_0 \leftarrow P$, Redundant state $R_1 \leftarrow P$, Dummy state $R_2 \leftarrow \beta$
2. $i \leftarrow 1$, $q \leftarrow 1$
3. $rstr \leftarrow \{0,1\}^t$ // $\#1(rstr) = 2n, \#0(rstr) = t - 2n$
4. $cstr \leftarrow \{0,1\}^{2n}$ // $\#1(cstr) = n, \#0(cstr) = n$
5. while $q \leq t$ do
6. $\quad\lambda \leftarrow rstr[q]$ // $\lambda = 0$ implies a dummy round while $\lambda = 1$ implies a computation round
7. $\quad\kappa \leftarrow (\lambda \cdot cstr[i]) \oplus 2(\neg\lambda)$ // $\kappa = 0$ or 1 depending on $cstr[i]$
8. $\quad\zeta \leftarrow \lambda \cdot \lceil i/2 \rceil$ // ζ is actual round counter, 0 for dummy
9. $\quad m \leftarrow (\neg(\lambda) \cdot m) \oplus (\lambda \cdot (((\neg(i \wedge 1)) \cdot m) \oplus ((i \wedge 1).RAND())))$ // new m if λ is 1 and i is odd
10. $\quad R_\kappa \leftarrow RoundFunction(R_\kappa \oplus (\neg(i \wedge 1) \cdot m), k^\zeta)$ // unmask if i is even
11. $\quad\gamma \leftarrow \lambda(\neg(i \wedge 1)) \cdot BLFN(R_0 \oplus R_1 \oplus (\neg(i \wedge 1) \cdot m))$ // unmask if i is even
12. $\quad\delta \leftarrow (\neg\lambda) \cdot BLFN(R_2 \oplus \beta)$
13. $\quad R_0 \leftarrow (\neg(\gamma \vee \delta) \cdot (R_0 \oplus ((i \wedge 1) \cdot m))) \oplus ((\gamma \vee \delta) \cdot R_2)$ // mask if i is odd
14. $\quad R_1 \leftarrow (\neg(\lambda) \cdot R_1) \oplus (\lambda \cdot (R_1 \oplus ((i \wedge 1) \cdot m)))$ // mask if i is odd and λ is 1
15. $\quad i \leftarrow i + \lambda$
16. $\quad q \leftarrow q + 1$
17. end
18. return(R_0)

Table 4: Computation of Algorithm 2

Step	Computation Round 1	Computation Round 2	Dummy Round
6.	$\lambda = 1$, i is odd	$\lambda = 1$, i is even	$\lambda = 0$
7.	$\kappa \leftarrow 1$	$\kappa \leftarrow 0$	$\kappa \leftarrow 2$
8.	$\zeta \leftarrow \lceil i/2 \rceil$	$\zeta \leftarrow \lceil i/2 \rceil$	$\zeta \leftarrow 0$
9.	$m \leftarrow RAND()$	$m \leftarrow m$	$m \leftarrow m$
10.	$R_{cstr[i]} \leftarrow RoundFunction(R_{cstr[i]}, k^\zeta)$	$R_{cstr[i]} \leftarrow RoundFunction((R_{cstr[i]} \oplus m), k^\zeta)$	$R_2 \leftarrow RoundFunction(R_2, k^0)$
11.	$\gamma \leftarrow 0$	$\gamma \leftarrow BLFN(R_0 \oplus R_1 \oplus m)$	$\gamma \leftarrow 0$
12.	$\delta \leftarrow 0$	$\delta \leftarrow 0$	$\delta \leftarrow BLFN(R_2 \oplus \beta)$
13.	$R_0 \leftarrow R_0 \oplus m$	$R_0 \leftarrow (\neg(\gamma) \cdot R_0) \oplus ((\gamma) \cdot R_2)$	$R_0 \leftarrow (\neg(\delta) \cdot R_0) \oplus ((\delta) \cdot R_2)$
14	$R_1 \leftarrow R_1 \oplus m$	$R_1 \leftarrow R_1$	$R_1 \leftarrow R_1$
15.	$i \leftarrow i + 1$	$i \leftarrow i + 1$	$i \leftarrow i + 0$
16.	$q \leftarrow q + 1$	$q \leftarrow q + 1$	$q \leftarrow q + 1$

order of redundant and cipher rounds is now random, we simply assume that the adversary skips instruction 15 in the penultimate computation round, which could be either a redundant or cipher round. It is to be noted that irrespective of whether a cipher or redundant round is targeted by the adversary, the value of i corresponding to this round is odd(as it is the penultimate round). So at the beginning of this round, a new random value of mask m is generated, and both R_0 and R_1 are thus masked at the end of this round. Thus, we have the following scenarios:

Scenario 1: The penultimate computation is redundant computation
If the instruction is skipped during the redundant computation, then the the last round comprises two consecutive redundant rounds. Thus, in this scenario, the adversary gets a ciphertext which is the output of the penultimate round XOR-ed with two distinct random values of m generated in the two consecutive redundant rounds. Thus, the obtained ciphertext gives the adversary no extra information about the key.

Scenario 2: The penultimate computation is cipher computation
In this scenario, the last round comprises of two consecutive cipher rounds. Thus, the obtained ciphertext is the output of an extra cipher round, but again XOR-ed with two distinct random values of m generated during the two consecutive cipher rounds. Hence, even in this scenario, the ciphertext so obtained gives the adversary no information about the key.

Note that the masking step is important otherwise in either scenario, the attacker would get the key easily, either from the output of the penultimate cipher round or the output of the additional cipher round. In the presence of the masking step,the only way for the adversary to get the output of the penultimate or the extra round is to also skip the masking step in the second redundant/cipher round. Thus, if the instruction skip attacks are now made in rounds q and $\hat{q}$, the new probability of a successful attack $Pr(\hat{e})$ becomes $\frac{\binom{q-1}{2n-2}\binom{t-q}{1}}{\binom{t}{2n}} \times \frac{\binom{\hat{q}-1}{2n-1}}{\binom{t}{2n}}$ which is less than the original attack probability. Moreover, even if the attacker skips the instructions, the output so obtained is either the output of the penultimate round or the ouput of the additional round with probability $\frac{1}{2}$. This adds to the computational complexity of the attack.

6 Instruction Level Fault Tolerant Implementation of the Infective Countermeasure

As evident from Section 5, algorithm 2 uses a combination of randomization and masking to reduce the probability of a specific instruction skip attack in which the attacker is able to skip targeted instructions, thus affecting the flow sequence of the final redundant and cipher rounds. However, it is not able to entirely obliterate the possibility of the attack. Using a stronger fault model that allows to perform stuck-at faults on consecutive bits of the bit string $cstr$, the adversary could easily set the bit pair corresponding to the last round to $(1, 0)$ with a probability of 1. In such a scenario, a single skip of the masking

instruction in the last computation round causes algorithm 2 to have precisely the same vulnerability to the instruction skip attack as 1. Moreover, while the assumed fault model focuses on only a single instance of the instruction skip attack, there are other variations of such attacks that could be mounted on the infective countermeasure. Hence, it is necessary to adopt stronger schemes that protect the countermeasure against instruction skip attacks in general.

A formal treatment of a countermeasure scheme at the machine instruction level against instruction skip fault attacks was presented by Heydemann *et. al* in [21]. The scheme is based on the assumption that while it is easy to inject identical faults in independent executions of an algorithm, introducing faults in two instructions separated by a few clock cycles is significantly harder. The scheme involves rewriting each individual instruction by a sequence of instructions that are immune to single instruction skips. The target architecture for their fault tolerant scheme is the Thumb2 instruction set which is a successor for both ARM and Thumb instruction sets. In this section we adopt a similar approach for the x86 instruction set. We briefly explain the equivalent fault tolerant scheme for x86 by first classifying the entire instruction set, and then explaining fault tolerant strategies for each category of instructions. Finally, we take a compiler generated machine level representation of the high level description of Algorithm 2 and rewrite the entire code using the fault tolerant scheme. Please note that henceforth, any reference to the x86 instruction set assumes only 32 bit instructions.

6.1 Instruction Classification

We use the classification scheme proposed in [21] to classify the entire x86 instruction set into 4 categories as follows:

1. Idempotent Instructions : Instructions that only need to be duplicated to achieve fault tolerance.
2. Separable Instructions : Instructions that can be replaced by a set of idempotent instructions followed by duplication to achieve fault tolerance.
3. Specific Instructions : Instructions that require specific replacement sequences to achieve fault tolerance.
4. Non-replaceable and Partially Replaceable Instructions : Instructions that either cannot be replaced by a fault tolerant sequence or can at best be replaced by a suitable combination of idempotent and non-idempotent instructions that is more fault tolerant than the original instruction.

Next, we present a few examples for each class of instructions from the x86 instruction set and demonstrate how each of them can be re-written to achieve fault tolerance.

Idempotent Instructions: As defined in [21], idempotent instructions have a disjoint set of source and destination operands, and the value of the destination operand after the execution of an instruction is independent of the location of

the instruction in a code. Hence, they only need to be duplicated to achieve fault tolerance. Table 5a presents a few instances of idempotent instructions from the x86 instruction set along with their corresponding fault tolerant replacement sequences. It is interesting to note here that unlike in Thumb2, the add instruction **addl %eax,%ebx** is not an idempotent instruction for the x86 instruction set. This is because, for the **addl** instruction, the destination operand is also a source operand. While this leads to an optimal usage of registers, it also leads to loss of idempotence for the **addl** instruction. For details of the **leal** instruction, please refer Appendix C.

Separable Instructions: This category of instructions are not idempotent by themselves due to fact that for these instructions, the destination register is also a source register. But these instructions can be replaced by a sequence of idempotent instructions, which can in turn be duplicated to achieve fault tolerance. However, such a replacement warrants the availability of one or more *dead* or idle register at this location in the code [21]. Table 5b illustrates how the **addl %eax,%ebx**, the **pushl %eax** and the **popl %eax** instructions respectively can be replaced by fault tolerant instruction sequences. The **pushl %eax** is equivalent to the set of instructions **subl %esp, $4 ; movl (%esp), %eax**. On the other hand , the **popl %eax** is equivalent to the set of instructions **movl (%esp), %eax; addl %esp, $4**. When writing the replacement sequence for the **pushl %eax** instruction, we assume that a register %rx stores the value of -4 in two's complement notation.

Table 5: Instruction Replacement Sequences

a.Replacement Sequences for Idempotent Instructions

Instruction	Replacement Sequence
movl %eax,%ebx (copies eax to ebx)	movl %eax,%ebx movl %eax, %ebx
movl %eax,-4(%ebp) (stores %eax at the address ebp-4)	movl %eax,-4(%ebp) movl %eax,-4(%ebp)
movl -8(%ebp),%eax (loads the value at the address ebp-8 to eax)	movl -8(%ebp),%eax movl -8(%ebp),%eax
leal %esi, [ebx + 8*eax + 4] (stores (ebx + 8*eax + 4) in esi)	leal %esi, [ebx + 8*eax + 4] leal %esi, [ebx + 8*eax + 4]

b.Replacement Sequences for Separable Instructions

Instruction	Replacement Sequence
addl %eax,%ebx	movl %eax, %ecx movl %eax, %ecx leal %eax, [%ebx + %ecx] leal %eax, [%ebx + %ecx]

pushl %eax	movl %esp, %ebx
	movl %esp, %ebx
	leal %esp, [%ebx + %rx]
	leal %esp, [%ebx + %rx]
	movl %eax, (%esp)
	movl %eax, (%esp)

popl %eax	movl (%esp), %eax
	movl (%esp), %eax
	movl %esp, %ebx
	movl %esp, %ebx
	movl $4, %ecx
	movl $4, %ecx
	leal %esp, [%ebx + %ecx]
	leal %esp, [%bax + %ecx]

Detailed descriptions of special, non-replaceable and partly replaceable instructions are presented in Appendices D and E respectively. We also present an

analysis of the average code size increase for the countermeasure on incorporating fault tolerance in the compiler generated code in Appendix F.

7 Simulation and Experimental Results

In this section, we present results of performed instruction skip attacks mounted on the three different versions of the infective countermeasures for AES128 - algorithms 1 and 2, as well as the fault tolerant version of 2. The experiments were divided in two broad categories. The first category of experiments were performed on C implementations of the infective countermeasures, where random instruction skips were simulated on the equivalent machine level representation of each countermeasure. The second category of experiments were performed on infective countermeasure circuits implemented using a Xilinx MicroBlaze softcore processor in Spartan 6 FPGA of SASEBO-W board. For this category of experiments, the instruction skips were achieved via timing violations at high clock frequencies.

7.1 Simulation Results

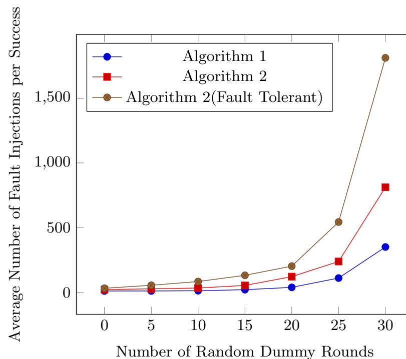


Fig. 1: Simulation Results : Impact of Number of Dummy Rounds on the Fault Attack Efficiency

The simulation experiments involved inflicting random instruction skips on $10,000$ runs of C implementations of each countermeasure scheme for fixed number of dummy rounds. The 128 bit plaintext and the 128 bit key were both randomly chosen, but the same input-key pair was used across all countermeasure schemes for normalization of results. For algorithm 1, a particular instance of the instruction skip attack was deemed to be successful if the faulty ciphertext matched with the output of the 9th round. For the naive and fault tolerant implementations of algorithm2, the instruction skip attack was deemed to be successful if the output of the countermeasure matched with the output of the

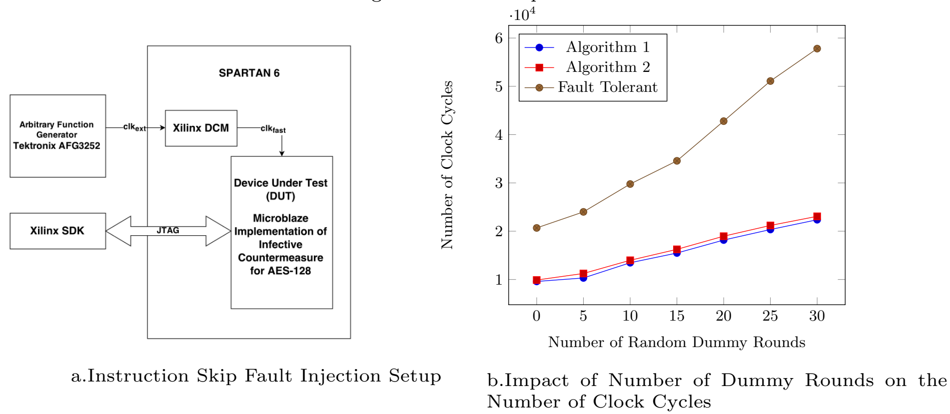
9th round or an extra 11th round. We refer to the output of a successful fault attack as a *useful* ciphertext. Finally, the average number of fault injections to get a single useful faulty ciphertext was plotted against the number of faulty bits for each scheme.

Figure 1 summarizes the results thus obtained. We note that for the same number of dummy rounds, the number of fault injections required by algorithm 2 is approximately double the number of fault injections required for algorithm 1. This is in accordance with the fact that the probability of a successful fault injection in algorithm 2 is $\frac{1}{2}$ the corresponding probability for algorithm 1. The fault tolerant implementation of algorithm 2 requires a much higher number of fault injections per success as compared to algorithm 2, and the ratio increases as the number of dummy rounds increases.

7.2 Fault Injection Set-up on SASEBO-W

Figure 3a describes our set up for instruction skip fault injection in the infective countermeasure for AES128. The set up consisted of an FPGA (Spartan-6 XC6SLX150) on a SASEBO-W platform, Xilinx SDK and an external arbitrary function generator (Tektronix AFG3252). The FPGA had a DUT (Device Under Test) block, which consisted of an infective countermeasure implementation for AES128 on a Xilinx MicroBlaze softcore processor. Instruction skip faults were injected in the DUT using clock glitches. The external clock signal clk_{ext} was supplied from the function generator. The high frequency clock signal clk_{fast} was then derived from the clk_{ext} signal via a Xilinx Digital Clock Manager (DCM) module and supplied to the DUT. Comparisons between the three different versions of infective countermeasures were made by setting the the appropriate countermeasure implementation as the DUT and subsequently collecting sets of faulty ciphertexts at different clock frequencies.

Fig. 2: Practical Experiments



a.Instruction Skip Fault Injection Setup

b.Impact of Number of Dummy Rounds on the Number of Clock Cycles

7.3 Experimental Results

We first compared the average number of clock cycles required for each version of the countermeasure against the number of dummy rounds. Figure 3b summarizes the results. It is interesting to note that there is only a slight increase in the number of cycles when one compares the performance of algorithm 2 with that of algorithm 1. The slight overhead could be attributed to the use of an extra bit string and the additional computations required for masking and to compute the values of γ. On the other hand, we observe a significant performance overhead for the fault tolerant version of algorithm 2. It is quite evident that the additional instructions necessary for achieving fault tolerance leads to a degradation in performance of the countermeasure in terms of number of clock cycles required.

Next, we inflicted instruction skip attacks on each of the three infective countermeasure implementations by causing critical path violations using clk_{fast}. Our aim was to compare the number of fault injections required per useful faulty ciphertext for each of the countermeasure schemes. We compared the results at six different clk_{fast} frequencies and for the number of the dummy rounds set at 0, 10, 20 and 30 respectively. We performed the fault injection trials in a range of clk_{fast} frequencies such that the faulty ciphertexts thus obtained were *useful* ones. Table 6 summarizes the experimental results. There are two essential observations in this regard:

1. The number of fault injections per useful ciphertext increases with an increase in the number of dummy rounds for each version of infective countermeasures.
2. The number of fault injections per useful ciphertext decreases with an increase in clk_{fast} frequency. This leads to the conclusion that a higher clock frequencies lead to greater probabilities of achieving instruction skip faults.

8 Conclusions

The paper shows that a recently proposed infective countermeasure is formally secure against DFA under the assumption that an attacker cannot subvert the control flow or skip instructions. The work identifies that such threats against the countermeasure exist because the scheme has a fixed ordering of the redundant and cipher rounds, leading to the fact that a fault in the redundant round is detected in the subsequent cipher round. This leads to the exposure of the previous round output which can lead to trivial attacks. Furthermore, the validity of a redundant round is not checked in the proposal. In order to reduce the attacker's success probability, the paper proposes suitable randomizations in the ordering of the redundant and cipher rounds, along with masking the previous round outputs. Subsequently, we implement the countermeasure using x86 instructions by rewriting the instructions in a fault tolerant manner. We propose several replacement sequences of the compiler generated code that makes use of idempotent instructions to reduce the probability of successful instruction skip attacks. Detailed simulations and real life experiments have been

Table 6: Experimental Results : Impact of Number of Dummy Rounds on the Fault Attack Efficiency

$\mathbf{clk_{fast}}$ **(MHz)**	**Number of dummy rounds**	**Fault Injections per useful faulty ciphertext**		
		Algorithm 1	**Algorithm 2**	**Algorithm 2(Fault Tolerant)**
128.0	0	150	250	2000
	10	225	600	5000
	20	400	1000	1×10^4
	30	1500	5000	4.5×10^5
128.4	0	80	150	500
	10	125	500	2500
	20	200	750	5000
	30	800	3000	1.5×10^5
128.8	0	51	92	205
	10	80	140	480
	20	120	250	750
	30	500	855	9.5×10^4
129.2	0	35	82	289
	10	65	109	368
	20	98	179	544
	30	420	635	5.2×10^4
129.6	0	23	71	185
	10	52	95	275
	20	87	145	369
	30	254	524	3.8×10^4
130.0	0	15	39	82
	10	27	56	159
	20	42	89	245
	30	159	355	1.4×10^4

performed on a MicroBlaze implementation of the countermeasure schemes on a SASEBO-W board, injected with faults via clock glitches. The experiments have demonstrated that the overall resistance to fault attacks is significantly higher for the proposed fault tolerant infective countermeasure scheme as compared to the already existing scheme.

References

1. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT 1997*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.
2. Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology – CRYPTO 1997*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.
3. Christophe Giraud. DFA on AES. In Hans Dobbertin, Vincent Rijmen, and Aleksandra Sowa, editors, *Advanced Encryption Standard – AES*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2005.
4. Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In Colin D. Walter, Çetin K. KoÇ, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.
5. Debdeep Mukhopadhyay. An Improved Fault Based Attack of the Advanced Encryption Standard. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 421–434. Springer, 2009.
6. Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In *Cryptographic Hardware and Embedded Systems-CHES 2010*, pages 320–334. Springer, 2010.
7. Chong Hee Kim. Differential fault analysis against aes-192 and aes-256 with minimal faults. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 3–9. IEEE, 2010.
8. Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault Attacks on AES with Faulty Ciphertexts Only. In Wieland Fischer and Jörn-Marc Schmidt, editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC 2013*, pages 108–118. IEEE Computer Society, 2013.
9. Michael Tunstall, Debdeep Mukhopadhyay, and Subidh Ali. Differential fault analysis of the advanced encryption standard using a single fault. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 224–233. Springer, 2011.
10. Tal G Malkin, François-Xavier Standaert, and Moti Yung. A comparative cost/security analysis of fault attack countermeasures. In *Fault Diagnosis and Tolerance in Cryptography*, pages 159–172. Springer, 2006.
11. Paolo Maistri and Régis Leveugle. Double-data-rate computation as a countermeasure against fault analysis. *IEEE Transactions on Computers*, 57(11):1528–1539, 2008.
12. Victor Lomné, Thomas Roche, and Adrian Thillard. On the Need of Randomness in Fault Attack Countermeasures - Application to AES. In Guido Bertoni and Benedikt Gierlichs, editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC 2012*, pages 85–94. IEEE Computer Society, 2012.
13. Benedikt Gierlichs, Jörn-Marc Schmidt, and Michael Tunstall. Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2012.

14. Alberto Battistello and Christophe Giraud. Fault Analysis of Infective AES Computations. In Wieland Fischer and Jörn-Marc Schmidt, editors, *Fault Diagnosis and Tolerance in Cryptography – FDTC 2013*, pages 101–107. IEEE Computer Society, 2013.
15. Harshal Tupsamudre, Shikha Bisht, and Debdeep Mukhopadhyay. Destroying fault invariant with randomization. In *Cryptographic Hardware and Embedded Systems–CHES 2014*, pages 93–111. Springer, 2014.
16. J Schmidt and Christoph Herbst. A practical fault attack on square and multiply. In *Fault Diagnosis and Tolerance in Cryptography, 2008. FDTC'08. 5th Workshop on*, pages 53–58. IEEE, 2008.
17. Alessandro Barenghi, Guido M Bertoni, Luca Breveglieri, and Gerardo Pelosi. A fault induction technique based on voltage underfeeding with application to attacks against aes and rsa. *Journal of Systems and Software*, 86(7):1864–1878, 2013.
18. Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*, pages 105–114. IEEE, 2011.
19. Amine Dehbaoui, J-M Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of aes. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2012 Workshop on*, pages 7–15. IEEE, 2012.
20. Elena Trichina and Roman Korkikyan. Multi fault laser attacks on protected crt-rsa. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*, pages 75–86. IEEE, 2010.
21. Karine Heydemann, Nicolas Moro, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. In *PROOFS 2013*.
22. Hamid Choukri and Michael Tunstall. Round reduction using faults. *FDTC*, 5:13–24, 2005.
23. J-M Dutertre, A-P Mirbaha, David Naccache, A-L Ribotta, Assia Tria, and Thierry Vaschalde. Fault round modification analysis of the advanced encryption standard. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, pages 140–145. IEEE, 2012.
24. Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88. IEEE, 2013.
25. J Schmidt and Marcel Medwed. A fault attack on ecdsa. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2009 Workshop on*, pages 93–99. IEEE, 2009.
26. Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

A Computing $Pr(k_i \mid \Delta_j)$

For a given differential value Δ_j and key hypothesis k_i for any cipher, the conditional probability $k_i \mid \Delta_j$ can be written as follows using Bayes' Theorem :

$$Pr(k_i \mid \Delta_j) = \frac{Pr(\Delta_j \mid k_i)Pr(k_i)}{Pr(\Delta_j)} \tag{10}$$

$Pr(\Delta_j \mid k_i)$ can be naively calculated by the following steps:

1. The first step is to enumerate all possible input differentials corresponding to the output differential Δ_j and key hypothesis k_i. This can be done by taking each possible input differential value and checking if the corresponding output differential, obtained using the given key k_i, matches that target output differential Δ_j.
2. Next, from the enumerated differential values, only those input differentials that are possible with respect to the fault model are retained, and the rest are discarded. For example, if the fault model is a *single bit fault model*. then a 2 bit input differential will not be considered as a possible case even if it leads to an output differential Δ_j. Let the pruned set of valid input differentials be of size D.
3. The value of $Pr(\Delta_j \mid k_i)$ is now calculated as $\frac{D}{N}$ where N is the number of possible values Δ_j can take.

However a more efficient technique to compute the probability is to compute $Pr(\Delta_j \mid k_i)$ for all possible output differentials Δ_j simultaneously. We consider each input differential possible under the fault model, and check the corresponding output differential value for the key hypothesis k_i. If the output differential corresponds to Δ_j, the score for Δ_j is incremented by 1. Finally the score of the output differential is normalized by N to obtain the desired probability. This takes much fewer number of computations.

The total probability $Pr(\Delta_j)$ can then be simply calculated as $\sum_{i=1}^{N} Pr(\Delta_j \mid k_i)$.

B Security of the Bit String *cstr*

Note that in algorithm 2, *cstr* is a sequence of 01 and 10 pairs, because the countermeasure must duplicate the computation of each round for fault detection. It is only the order of the redundant and cipher computations for each round that is scrambled using *cstr*. An adversary might therefore try to upset the order of round execution by performing bit flips or stuck-at fault attacks on *cstr*. *A single bit flip in cstr would result in either two consecutive redundant rounds or two consecutive cipher rounds*. In either scenario, the same register would be updated twice in both rounds - R_1 for redundant or R_0 for cipher

round respectively. The algorithm will automatically detect the fault in the second computation, as one of R_0 or R_1 will contain the output of the current round while the other still contains the output of the previous round. Thus as long as the variable i is incremented appropriately after each round computation, the algorithm is fault-tolerant to single bit upsets on *cstr*. However, the adversary could reverse the order of the redundant and original computations corresponding to a round r by flipping the $(2r-1)^{th}$ and $(2r)^{th}$ bits. In such a scenario, the algorithm cannot detect the fault. In that case, the adversary must make sure that both the bits of *cstr* that are flipped correspond to computations for the same round, which demands slightly greater precision than a naive two bit flip attack. For the naive attack, the probability of appropriate bit flips is $(n-1)/\binom{n}{2}$, which is much smaller than the single bit flip probability.

C The *leal* instruction:

The *load effective address* instruction is a 32 bit instruction of the x86 instruction set that is meant for performing memory addressing calculations without actually accessing the memory content. This instruction can be used to add not only memory addresses but also any pair of 32 bit registers. Although originally meant for directly mapping high level memory references, **leal** is essentially an arithmetic instruction with two major advantages over the **addl** instruction, namely - the ability to perform addition with either two or three operands and the ability to store the result in any register (which is not necessarily a source operand).

D Special Instructions:

A large number of instructions cannot be easily replaced by a sequence of idempotent instructions. However, some of these instructions can still be broken down into an alternative sequence of instructions, not necessarily idempotent. These instructions can still be duplicated to achieve the desire fault tolerance. Table 7 demonstrates how the **call <function>** instruction can be made fault tolerant. In x86, the function call mechanism involves pushing the return address onto the stack followed by an unconditional jump. This idea is exploited to rewrite the **call** instruction as follows. First, the return address is computed by adding 1 to the address of the **returnlabel** and is pushed onto the stack. This is followed by two successive unconditional jump instructions. It is interesting to note that although the unconditional jump is not an idempotent instruction, the sequencing of the instructions ensures that both jump instructions can never be executed sequentially even if no fault occurs. When writing the replacement sequence, we replace the **pushl %eax** instruction by its equivalent fault tolerant sequence, with the assumption that register %rx stores the value of -4 in two's complement notation.

Table 7: Replacement Sequence: **call <function>**

movl %ebx, <returnlabel>
movl %ebx, <returnlabel>
movl $1, %ecx
movl $1, %ecx
leal %eax, [%ebx + %ecx]
leal %eax, [%ebx + %ecx]
movl %esp, %ebx
movl %esp, %ebx
leal %esp, [%ebx + %rx]
leal %esp, [%bax + %rx]
movl (%esp), %eax
movl (%esp), %eax
jmp <function>
jmp <function>
returnlabel:

E Non-replaceable and Partially Replaceable Instructions in the x86 Instruction Set

There are a large number of instructions in the x86 instruction set that cannot be replaced by an equivalent fault tolerant sequence of instructions. An example of a non-replaceable instruction is the **jne** (jump if not equal) instruction. For such instructions, fault detection schemes are the only possible solution. There are also partially replaceable instructions such as the **subl** instruction that can be written by a combination of several idempotent instructions and a few non-idempotent ones. This helps reduce the probability that the attacker can skip precisely the non-idempotent instructions.

Table 8: Replacement Sequence: **subl %eax,%ebx**

Instruction	Replacement Sequence
subl %eax,%ebx	compl %eax, %rx
	movl $1, %ry
	movl $1, %ry
	leal %rz, [%rx + %ry]
	leal %rz, [%rx + %ry]
	movl %eax, %ecx
	movl %eax, %ecx
	leal %eax, [%ecx + %rz]
	leal %eax, [%ecx + %rz]

The instruction set for x86 is highly optimized with respect to the usage of registers. Consequently, for most arithmetic instructions, such as the **addl** and **subl** instructions, the destination operand is also a source operand. While the

addl instruction can be replaced by the idempotent **leal** instruction, it is not possible to make any such idempotent replacement for the *subl* instruction. For the *subl* instruction, the best we can do is replace the single instruction by a sequence of instructions, a large fraction of which are idempotent. This reduces the probability that the few remaining non-idempotent instructions are skipped by the adversary. The concept is elucidated by a possible replacement sequence for the **subl** as shown in Table 8. We first compute the two's complement of **%ebx** and add it to **%eax**. The **compl** instruction used is non-idempotent, however the probability that the attacker skips precisely this instruction among all instructions in this replacement sequence is $\frac{1}{10}$, as compared to 1 for a single **subl** instruction. Thus the idea is to reduce the proportion of non-idempotent instructions as far as possible to achieve greater fault tolerance.

F Fault Tolerant Implementation of the Infective Countermeasure for AES

Our next step is to write the infective countermeasure scheme presented in algorithm 2 in the aforementioned fault redundant framework. We first translate a high level implementation of the algorithm in C into a sequence of machine level instructions. The translation is done using a GNU GCC compiler on a 32-bit Intel processor. It generates a sequence of 32 bit x86 instructions. We then replace each instruction by a fault tolerant sequence of instructions, provided such a sequence exists for the corresponding instruction. Table 9 summarizes the various categories of instructions in the original machine level representation of algorithm 2 and also presents the average blowup due to each category.

Table 9: Fault Tolerant Infective Countermeasure : Blowup at the Instruction level

Instruction Category	**Original Number of Instructions**	**Percentage Blowup**
Idempotent	368	100.0
Separable	160	332.2
Special	68	847.4
Non-replaceable & Partially replaceable	570	50.0
Overall	1166	132.1