# Fault Attack on AES with Single-Bit Induced Faults

Alessandro Barenghi*, Guido M. Bertoni†, Luca Breveglieri*, Mauro Pellicioli* and Gerardo Pelosi*
*DEI–Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano (MI), Italy
Email: {barenghi,brevegli,pelosi}@elet.polimi.it, mauro.pellicioli@mail.polimi.it
†STMicroelctronics, 20041 Agrate Brianza (MB), Italy
Email: guido.bertoni@st.com

***Abstract*—This work presents a differential fault attack against AES employing any key size, regardless of the key scheduling strategy. The presented attack relies on the injection of a single bit flip, and is able to check for the correctness of the injection of the fault a posteriori. This fault model nicely fits the one obtained through underfeeding a computing device employing a low cost tunable power supply unit. This fault injection technique, which has been successfully applied to hardware implementations of AES, receives a further validation in this paper where the target computing device is a system-on-chip based on the widely adopted ARM926EJ-S CPU core. The attack is successfully carried out against two different devices, etched in two different technologies (a generic 130nm and a low-power oriented 90nm library), running a software implementation of AES-192 and AES-256 and has been reproduced on multiple instances of the same chip.**

***Index Terms*—Fault injection attacks, Side channel attacks.**

## I. Introduction

In the recent years, the increasing adoption of embedded and portable systems, has bred a wide interest in both academical and industrial communities towards security-enhancing features of digital devices. The security of cryptographic algorithms is based both on the strength of the underlying mathematical problem and on a properly engineered implementation. Fault attacks alter the execution of a cryptographic algorithm in order to collect faulty ciphertexts which leak information related to the secret key. This work provides new insights on the security of the AES implementation, with any key scheduling strategy, and in particular on the versions employing keys larger than 128 bits. The attack proposed in this paper relies on a transient single-bit flip, practically induced through a constant underfeeding of the computing circuit, which has been proven feasible employing only a very cheap and simple apparatus [1]–[3]. Depending on the device, it is quite easy to recognize a supply voltage region where this kind of faults appear in rising quantity throughout the whole computation [2], [4]. These faults are induced with no synchronization with respect to the execution, thus not all of them will provide a useful ciphertext in order to lead the attack. All the non exploitable faults are discarded properly by our algorithm, which provides a way to recognize which ones are respecting the hypotheses needed to infer parts of the secret key, regardless of the timing and position of the errors. This leads to a greater ease in the attack setup, since no ad-hoc timing devices for an execution synchronous underfeeding of the circuit are required. Performance figures show that it is possible to successfully break the AES for every key length within minutes of work encompassing both the fault collection phase and the computational post processing of the results. The remainder of the paper is organized as follows: Section II summaries the related work on the single-bit fault model, Section III briefly revises the AES design, Section IV introduces the proposed attack, and Section V reports methodology for injecting faults and describes experimental results. Finally, Section VI summarizes our conclusions and points towards future research directions.

## II. Related Work

Recently, the attention of researchers towards fault attacks against AES with any key size has incremented. Three recent papers [4]–[6] presented attacks similar to the one presented in this work. In [4], the authors introduce a single-byte attack which, keeping the plaintext fixed, retrieves an intermediate state in the penultimate round and uses it to find the penultimate round key. Takahashi et al. [6] are able to obtain 192-bit keys using 3 pairs of correct and faulty ciphertexts, and the 256-bit key using 2 pairs of correct and faulty ciphertexts and 2 pairs of correct and faulty plaintexts, with a single byte fault model. The attack is cheap in terms of number of faults but it requires to target specifically one byte in rounds 8, 9 for the AES-192 and round 11 for the AES-256. This in turn implies that the attacker must have a precise control on the timing of the attack. Moreover, a distinguishing criteria for the useful faults is not provided and, it has not been practically validated by the authors on a real platform. Moreover, for the AES-256 case, it is required to inject faults during decryption, which is not always possible, since the attacker may not be granted access to a device able to decrypt the ciphertext. Li et al. [5] assume the same two fault models of [7]. The first one covers differences of up to 3 bytes while the second requires an entire 32-bit column of the state being different from the correct one. Albeit the assumed fault models are encompassing very wide faults, the attack requires a considerable number of faulty ciphertexts to be collected: 6 for the first fault model and around 1500 in the second, in order to retrieve a single column of the key. Thus, around 7×4 and 1500×4 faults are needed to carry

out the attack employing the first and second fault model respectively. A further downside of these technique are the steep requirements, both in terms of memory footprint and computing power, on the computing platform needed to analyze the results. In fact, when employing the second fault model, the number of candidates is reduced very slowly, thus requiring the memorization of huge sets during the pruning phase. The attack also needs to keep the plaintext fixed during the whole attack and the authors do not provide a practical validation of the technique against a real world device.

## III. Preliminaries on AES

The AES cipher is a subset of Rijndael [8], it processes 128-bit wide plaintext blocks using a key, whose size may be selected among 128-bit, 192-bit or 256-bit. Longer key sizes correspond to higher security, obtained at the cost of a longer processing time. The AES is an iterated block cipher, where the *round* is the transformation that elaborates 128 bits key of cipher state and 128 bit of round key. The number of rounds is determined by the key size, and is equal to 10, 12 and 14 rounds for 128-bit, 192-bit and 256-bit keys, respectively. For the sake of clarity in the description, the cipher state can be seen as a 4×4 matrix, where each element of the matrix is a byte. The round is composed by 4 basic transformations: SubBytes (a non linear function, usually in the form of a lookup table, applied byte-wise), ShiftRows (the rows of the matrix are shifted by a constant amount), MixColumns (the cipher state is multiplied column-wise by a constant matrix, this correspond to a mixing of bytes within a column), and AddRoundKey (`xor` addition of the words of the expanded key). All the rounds are equal, except for the last one, which is missing the MixColumns, and the first one which has an additional key addition before its start. The secret key is expanded through the KeySchedule algorithm: depending on the key size, the number of rounds $r$ to be applied is defined, and consequently the length of the expanded key, the cipher key $k$ is transformed in $4(r+1)$ 32-bit words, where $r \in \{10, 12, 14\}$ while key length $s$ is equal to 4, 6 or 8 words. Usually the expanded key is represented as an array of 32-bit words, $W[0, \ldots, 4(r+1)-1]$. The first $s$ words of $W$ are filled with the secret key itself, while the others are obtained through linear combination of the first four words, interleaving, every $s$ words, a non linear step [8]. In order to speed up the execution of the algorithm on small computing platforms, it is possible to merge together the operations of SubBytes and MixColumns through precomputing larger lookup tables (known as T-Tables) which act as the functional composition of the two operations.

## IV. Single-Bit Fault Attack to AES

Underfeeding the power supply enables a *graceful degradation* of the functioning of the device, making it realistic to induce single-bit error in the computation. In particular, the cryptographic device is underfed in such a way that each encryption is characterized by a lone single-bit fault. The new attack exploits the single-bit errors located in the penultimate round and describes a method to distinguish the faults that are potentially useful to reduce the candidate key space, allowing the application of the attack even when the attacker has no control over the fault injection timing. The new technique is able to recover any round subkey from the AES cipher, regardless of both the key scheduling algorithm or the number of rounds, thus encompassing future modifications. Our proposal is the first attack to AES-192 and AES-256 reporting a practical result in an experimental campaign with a cheap workbench, able to inject a single bit flip fault. For the sake of clarity, the proposed attack will be presented while referring to the plain AES implementation previously summarized.

The purpose of the new attack is to retrieve the penultimate round subkey $k^{(r-1)}$ in the case of AES-256 and the second half of it in the case of AES-192. The inverse key scheduling process can then be applied to find the cipher key $k$, using the method described in [9].

A differential fault attack working against Substitution-Permutation Networks, is introduced in [10] under the hypothesis of a single-byte change between the last and the last-but-one diffusion layers (MixColumns in the case of AES). The application to AES enables the attacker to recover the last round subkey $k^{(r)}$, working independently on different words. The attack in [10] provides also a distinguishing criteria to identify exploitable faults, relying on the fact that only one 32-bit column of the erroneous ciphertext $\widetilde{c}^{(r)}$ is different from the correct one of $c^{(r)}$. However, the proposed algorithm is not extended to recover subkeys other than the last. In the following we describe an algorithm able to pierce successfully a regular round of the AES cipher (i.e. one including the MixColumns), thus obtaining a method able to roll back the whole cipher and retrieve all the round subkeys regardless of their mutual relations or the number of rounds.

### A. Retrieval of the Last Round Subkey, $k^{(r)}$

The first step of the attack is to retrieve the last round subkey in a word oriented fashion, using the method described in [10]. The attack in [10] can easily fit a single bit fault model, since it is an actual restriction of the required fault hypothesis (single byte fault). Thus, it is possible to use it in order to find the last key $k^{(r)}$, as detailed in Function IV.1 (GetLastSubKeyWord). Let us denote a faulty ciphertext as $\widetilde{c}=\langle \widetilde{c}_0, \widetilde{c}_1, \widetilde{c}_2, \widetilde{c}_3 \rangle$, and a fault-free one as $c=\langle c_0, c_1, c_2, c_3 \rangle$, where $\widetilde{c}_u, c_u$, $0 \leq u \leq 3$, are the columns of the matrices. The evaluation of possible differences between $\widetilde{c}_u$ and $c_u$ (caused by a single byte fault between the last and the last-but-one MixColumns step) adds up to $255 \times 4 \times 4$ different values. Such values can be listed through enumerating all the state tables resulting from changing a single fixed-position byte value, and then repeating the change for each one of the 16 bytes composing the state table, i.e.:

**Function IV.1**: `GetLastRoundKeyWord(`$j$`)`

**Input** : $j \in \{0,1,2,3\}$, word index of the last subkey
**Output**: $k_j^{(r)}$, $j$-th word of the last subkey $k^{(r)}$
**Data** : $\Delta = \{\langle \delta_0,0,0,0\rangle, \langle 0,\delta_1,0,0\rangle, \langle 0,0,\delta_2,0\rangle, \langle 0,0,0,\delta_3\rangle\}$, $\delta_u \in \{x,\ x\,2^8,\ x\,2^{16},\ x\,2^{24} \mid 0 \le x < 2^8\}$, $0 \le u \le 3$, $|\Delta| = 255 \times 4 \times 4$; $\Delta' = \{\, d \mid d \leftarrow \text{MixColumns}(\delta), \forall \delta \in \Delta\,\}$; $\zeta$ denotes a matrix storing a differential value of the AES state matrix; $\langle \zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle$ specifies the state matrix as the tiling of four 32-bit words

1 Generate a random plaintext $p$
`/* Record a faulty and fault-free ciphertext */`
2 $c^{(r)} \leftarrow \text{AES\_Encrypt}_k(p)$ `/* k: unknown cipher key */`
3 $\widetilde{c}^{(r)} \leftarrow \text{Faulted\_AES\_Encrypt}_k(p)$
4 Carve words $\widetilde{w}, w$ both according to $j$ and taking into account the last ShiftRows operation
5 $L \leftarrow \emptyset$ `/* Set up of Candidate-Words List */`
6 **foreach** $\bar{k} \in \{0,\ldots,2^{32}-1\}$ **do**
7 $\quad \langle \zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \leftarrow \langle 0,0,0,0\rangle$
8 $\quad \zeta_j \leftarrow \text{InvSubBytes}(w \oplus \bar{k}) \oplus \text{InvSubBytes}(\widetilde{w} \oplus \bar{k})$
9 $\quad$ **if** $\langle \zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \in \Delta'$ **then** $L \leftarrow L \cup \{\bar{k}\}$
`/* Word Selection Phase */`
10 **while** $|L| > 1$ **do**
11 $\quad$ Generate a random plaintext $p$
$\quad$ `/* Record a faulty & fault-free ciphertext */`
12 $\quad c^{(r)} \leftarrow \text{AES\_Encrypt}_k(p)$ `/* k: unknown cipher key */`
13 $\quad \widetilde{c}^{(r)} \leftarrow \text{Faulted\_AES\_Encrypt}_k(p)$
14 $\quad$ Carve words $\widetilde{w}, w$ both according to $j$ and taking into account the last ShiftRows operation
15 $\quad$ **foreach** $\bar{k} \in L$ **do**
16 $\quad\quad \langle \zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \leftarrow \langle 0,0,0,0\rangle$
17 $\quad\quad \zeta_j \leftarrow \text{InvSubBytes}(w \oplus \bar{k}) \oplus \text{InvSubBytes}(\widetilde{w} \oplus \bar{k})$
18 $\quad\quad$ **if** $\langle \zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \notin \Delta'$ **then** $L \leftarrow L \setminus \{\bar{k}\}$
19 **return** $k_j^{(r)}$ `/*` $L = \{\bar{k}\},\ \bar{k} = k_j^{(r)}$ `*/`

$\Delta = \{\langle \delta_0,0,0,0\rangle,\ \langle 0,\delta_1,0,0\rangle,\ \langle 0,0,\delta_2,0\rangle,\ \langle 0,0,0,\delta_3\rangle\}$, $\delta_u \in \{x,\ x\,2^8,\ x\,2^{16},\ x\,2^{24} \mid 0 \le x < 2^8\}$, $0 \le u \le 3$. The inter-byte diffusion operated by the last MixColumns maps bijectively each difference value into another thus obtaining another set of differential state tables with the same cardinality of $\Delta$: $\Delta' = \{\, d \mid d \leftarrow \text{MixColumns}(\delta), \forall\ \delta \in \Delta\,\}$. The first step of the algorithm is to generate a random plaintext $p$ (line 1) and to encrypt it both in a correct and in a fault inducing condition, outputting $c$ and $\widetilde{c}$ respectively (lines 2–3). The next step extracts the word $w$ from $c$ and the word $\widetilde{w}$ from $\widetilde{c}$ according to the positions in which $c$ is different from $\widetilde{c}$. In lines 6–9 each of the candidate values for the target word of the last subkey is checked for compatibility with the fault model. Keeping $\bar{k}$ as the current key candidate, lines 7–8 build the matrix $\zeta$, which represents the differential value before the last SubBytes step. If the difference $\zeta$ is compatible with the set $\Delta'$, meaning that it may have been originated by a single byte difference before MixColumns in the $(r-1)$-th round, then the current candidate is inserted in $L$, otherwise it is discarded. Lines 10–18 simply repeat this process considering a new couple of correct and faulty ciphertext and using as key candidates only the elements already in set $L$. If an element that was previously accepted does not satisfy condition in line 18 then it is removed from $L$ (line 18). At the end, $L$ will contain a single value for $k_j^{(r)}$.

**Function IV.2**: `GetPotentialFault(`$j$, $k^{(r)}$`)`

**Input** : $k^{(r)}$, last round subkey,
$j \in \{0,1,2,3\}$, word index of the state in the penultimate round
**Output**: $(c_j^{(r-1)}, \widetilde{c}_j^{(r-1)})$,
$c_j^{(r-1)} = \langle c_{j,0}^{(r-1)}, c_{j,1}^{(r-1)}, c_{j,2}^{(r-1)}, c_{j,3}^{(r-1)}\rangle$,
$\widetilde{c}_j^{(r-1)} = \langle \widetilde{c}_{j,0}^{(r-1)}, \widetilde{c}_{j,1}^{(r-1)}, \widetilde{c}_{j,2}^{(r-1)}, \widetilde{c}_{j,3}^{(r-1)}\rangle$
**Data** : $\Gamma = \{\langle \gamma_0,0,0,0\rangle, \langle 0,\gamma_1,0,0\rangle, \langle 0,0,\gamma_2,0\rangle, \langle 0,0,0,\gamma_3\rangle\}$, $\gamma_u \in \{\text{SubBytes}(\alpha) \oplus \text{SubBytes}(\tilde{\alpha}),\ \alpha \oplus \tilde{\alpha} \in \{2^i,\ 0 \le i \le 7\}\}$, $0 \le u \le 3$, $|\Gamma| = 163 \times 4$

1 Generate a random plaintext $p$
2 $c^{(r)} \leftarrow \text{AES\_Encrypt}_k(p)$ `/* k: unknown cipher key */`
3 $c^{(r-1)} \leftarrow \text{InvSubBytes}(\text{InvShiftRows}(c^{(r)} \oplus k^{(r)}))$
4 **repeat**
5 $\quad \widetilde{c}^{(r)} \leftarrow \text{Faulted\_AES\_Encrypt}_k(p)$
6 $\quad \widetilde{c}^{(r-1)} \leftarrow \text{InvSubBytes}(\text{InvShiftRows}(\widetilde{c}^{(r)} \oplus k^{(r)}))$
7 **until** $(\exists\,!\ j' \in \{0,1,2,3\} \mid j' = j \wedge c_j^{(r-1)} \neq \widetilde{c}_j^{(r-1)}) \wedge$
8 $\quad (\text{InvMixColumns}(c_j^{(r-1)} \oplus \widetilde{c}_j^{(r-1)}) \in \Gamma)$
9 **return** $\langle c_j^{(r-1)}, \widetilde{c}_j^{(r-1)}\rangle$

### B. Exploitable Fault Selection

Our extension, from now on, works under a known ciphertext assumption with no particular requirements on the enciphered plaintexts, other than having pairs of faulty and fault free ciphertexts obtained from the same plaintext. In order to perform the retrieval of the key $k^{(r-1)}$ we assume that an erroneous ciphertext is the result of a single bit fault occurred just before the SubBytes operation in the $(r-1)$-th round. This fault will result in a single byte corruption of the state just before the MixColumns in the $(r-1)$-th round. Due to the position and shape of these faults, they may also be employed to retrieve the last round subkey as described in Function IV.1, enabling the attacker to further reduce the number of faults he needs to inject. As a first step, we describe the function able to detect the useful faults and discard the un-exploitable ones. Function IV.2 takes as input a copy of the last round subkey $k^{(r)}$, and returns a pair of faulty-free ($c_j^{(r-1)}$) and faulty ($\widetilde{c}_j^{(r-1)}$) columns of the penultimate round output, together with their position ($j \in \{0,1,2,3\}$) in the state matrix. In order to distinguish exploitable faults from non exploitable ones, the effect of the S-box on a difference between two inputs must be investigated. Through exhaustive computation, we determined that, given a single bit difference between two byte values $\alpha$ and $\tilde{\alpha} = \alpha \oplus 2^i$, $0 \le i \le 7$, the possible differences $\text{SubBytes}(\alpha) \oplus \text{SubBytes}(\tilde{\alpha})$ are only 163. The size of the set $\Gamma$ of all the possible word-wise differences, which may happen after the ShiftRows operation, adds up to $163 \times 4$, since the single bit fault may happen in any of the four bytes involving a word of the state matrix. Once this set has been precomputed and stored in memory, the function GetPotentialFault considers a fixed plaintext (line 1), removes from the

corresponding correct ciphertext $c^{(r)}$ the effect of the last round, producing the correct $c^{(r-1)}$ output of the penultimate round (lines 2–3). An analogous procedure is employed to produce the faulty output $\widetilde{c}^{(r-1)}$ of the penultimate round (lines 5–6), the only difference is that now a fault is injected during the computation (line 5). After computing $c^{(r-1)}$ and $\widetilde{c}^{(r-1)}$, the algorithm, acting word-wise, computes the difference between the correct and faulty state and inverts the MixColumns operation on it, checking if only one resulting word belongs to $\Gamma$ and the other ones are identically zero. If this is the case, it returns the pair of correct and faulty words, together with their position $j \in \{0,1,2,3\}$, otherwise the algorithm goes back to consider a new faulty ciphertext and performs the analysis again. We want to stress that the GetPotentialFault only represents a good heuristic in order to determine if a faulty ciphertext has been generated by a single bit fault, since multi-bit faults occurring in a byte, $\alpha$, may generate a value $\widetilde{\alpha}$ such that $\gamma_u = (\text{SubBytes}(\alpha) \oplus \text{SubBytes}(\widetilde{\alpha}))$, $u \in \{0,1,2,3\}$ ( $\Gamma = \{\langle\gamma_0,0,0,0\rangle, \langle 0,\gamma_1,0,0\rangle, \langle 0,0,\gamma_2,0\rangle, \langle 0,0,0,\gamma_3\rangle\}$ ). Viceversa, the key observation which enables us to reduce the keyspace is that no other values but the ones in $\Gamma$ may be generated, at the output of the S-box, by a single bit difference. Since the faults are equally distributed over the cipher states, the probability that a fault hits a word of the state under attack is $\frac{1}{r}$, thus resulting in an average requirement of $\frac{r}{2}$ trials to obtain an exploitable fault. Assuming that the attacker is able to inject a single bit fault with probability $p_s$, the average number of faults which must be injected raises to $\frac{r}{2p_s}$. As reported in [2], it is possible to tune correctly the workbench, so that $p_s$ is very close to 1. Taking into account the possible double injection of the same fault, and denoting the probability of injecting a different fault as $p_d$ (the actual $p_d$ depends on the hardware fault model), the average number of faults to be collected to obtain distinct, exploitable faults raises to $\frac{r}{2p_s p_d}$. In case the attacker is not able to selectively inject only single bit faults, the checking against the $\Gamma$ set provides a method to raise the probability of returning an exploitable fault from $\frac{8}{255}$ (accepting any single byte fault as possible single bit) to $\frac{8}{163}$, a twofold improvement.

### C. Single bit Attack

Algorithm IV.3 describes the new attack technique to retrieve the last-but-one round subkey $k^{(r-1)}$ employing Function IV.2 and the last round subkey $k^{(r)}$ obtained through Function IV.1 (lines 1–3). This algorithm takes into account the fact that the heuristic in GetPotentialFault may yield non exploitable faults, thus it is repeated until all the four key words of $k^{(r-1)}$ are successfully recovered. In order to keep track of which ones have already been extracted, a set of indexes $J = \{0,1,2,3\}$ is initialized at line 4. The body of the while loop (lines 5–25) acts in two distinct phases: at first it builds four different lists $L_j$, one for each candidate word of the

**Algorithm IV.3: AES Single-Bit Attack**

```
   Output: k^(r-1), penultimate round subkey
1  begin
       /* k^(r)=⟨k_0^(r), k_1^(r), k_2^(r), k_3^(r)⟩                    */
2      foreach i ∈ {0,1,2,3} do
3          k_i^(r) ← GetLastRoundKeyWord(i)
       /* k^(r-1)=⟨k_0^(r-1), k_1^(r-1), k_2^(r-1), k_3^(r-1)⟩          */
4      J ← {0,1,2,3}
5      while J ≠ ∅ do
           /* List Filling Phase                                       */
6          foreach j∈J do
7              (w, w̃) ← GetPotentialFault(j, k^(r))
               /* w ← c_j^(r-1); w̃ ← c̃_j^(r-1);                        */
8              L_j ← ∅
9              foreach v ∈ {0,...,2^32 − 1} do
10                 ω←InvSubBytes(InvMixColumns(w⊕v))
11                 ω̃←InvSubBytes(InvMixColumns(w̃⊕v))
12                 if (ω⊕ω̃) ∈ {2^i, 0 ≤ i < 32} then
13                     L_j ← L_j ∪ { v }
           /* List Pruning Phase                                       */
14         foreach j∈J do
15             while |L_j| > 1 do
16                 (w, w̃) ← GetPotentialFault(j, k^(r))
17                 foreach v ∈ L_j do
18                     ω←InvSubBytes(InvMixColumns(w⊕v))
19                     ω̃←InvSubBytes(InvMixColumns(w̃⊕v))
20                     if (ω⊕ω̃) ∈ {2^i, 0 ≤ i < 32} then
21                         L_j ← L_j \ { v }
22             if |L_j|=1 then
23                 k_j^(r-1) ← v̄ /*   L = {v̄}                          */
24                 J ← J \ { j }
           /* Whether J ≠ ∅ (∃ j ∈ {0,1,2,3} | |L_j|=0),
              go back to line 5                                        */
25 end
26 return k^(r-1)
```

subkey to be found (lines 5–13), then it prunes them until a single candidate for each list is left (or until a brute force search over the reduced keyspace is feasible) (lines 14–24). The first phase, repeated once for every subkey word to be found, starts obtaining the $j$-th word of an exploitable correct-faulty ciphertext pair through the GetPotentialFault function: $w \leftarrow c_j^{(r-1)}; \widetilde{w} \leftarrow \widetilde{c}_j^{(r-1)}$ (line 7). In order to fill the list of candidates for the corresponding word of the subkey, the two state values are added to the key, passed through an InvMixColumns and an InvSubBytes function, obtaining $\omega$ and $\widetilde{\omega}$ respectively (lines 10–11). If the difference (computed through `xor`) between $\omega$ and $\widetilde{\omega}$ is non zero in a single bit, the key candidate is deemed compatible with the current state pair and added to the list (lines 12–13). Note that, the algorithm does not consider the effect of the InvShiftRows operation, which is present between the InvSubBytes and InvMixColumns steps. This is not an issue since the InvShiftRows will move a byte from each word into each other. This results into having a single faulty byte per word which allows the attacker to infer the correct position of the fault. It is then possible to deduce the correct word which has been affected with the fault from the position of the faulty byte in the shifted state and the column affected

by the fault. Now, all the four lists $L_j$, $j \in \{0,1,2,3\}$, have been filled with candidate keys. In the event a brute force search over all the possible cipher subkeys $k^{(r-1)}$, which can be built through the combination of all the words in the four lists, is not directly possible or not desired, the algorithm starts a list pruning phase. The pruning phase follows the same procedure of the filling phase for each list $L_j$ (line 14), the only difference is that instead of adding a new key candidate word $v$ to $L_j$, it checks if a key candidate word in $v \in L_j$ passes the check against a new correct-faulty state pair, and removes it in case the check is missed (lines 15–21). At the end of the pruning phase the list $L_j$ will either contain a single candidate or be empty. In case a single candidate is present (line 22), this value is assigned to the subkey word $k_j^{(r-1)}$ and the index $j$ is extracted from the set $J$ (lines 22–24). The case of a list being empty is generated by a non single-bit fault misleading the GetPotentialFault function, and generating an unsolvable set of constraints for the candidate keys. In this case, the algorithm simply resumes, from the filling phase, the search for the candidate key word (i.e., it goes back to line 5). It is not possible for the multiple-bit faults to lead the algorithm to compute a wrong key candidate word, since the checks to fill in (lines 12–13) and spill from (lines 20–21) the candidate lists are done against the single-bit fault hypothesis, thus leading to a contradictory set of constraints in case of a multi-bit fault. After the execution of the algorithm, we have retrieved the whole round subkey $k^{(r-1)}$ without considering any relation with the KeySchedule algorithm, thus we can roll back one more round. If the algorithm under attack is either AES-192 or AES-256, it is already possible to compute the whole key schedule since we are in possess of enough key material to invert it and derive the cipher key $k$.

## V. Experimental Results

### A. Experimental Settings

In this section the setup of the attack environment, used to demonstrate the feasibility of the attack in a realistic environment, is described. This provides insights on a low cost fault injection technique, not invasive, not destructive and easily reproducible. This represents a practical attack against a software execution of the AES algorithm, but this technique is not limited to software implementations. A practical proof of feasibility on a secure hardware implementations of AES on a tamper proof device (smart card) is given in [1]. The overall architecture is the same as that used in [2].

Device Under Attack The devices attacked are system-on-chips based on the general purpose 32-bit CPU ARM926EJ-S[1]. This CPU is based on a RISC Harvard architecture CPU, with 16 general purpose registers and a 5-stages pipeline. We had the possibility to test the attack on two different devices, both equipped with the same CPU though implemented in different silicon technologies. One of these devices is optimized for mobile applications adopting a low power technology, while the other is a general purpose device used for different applicative targets like computer peripherals or telecommunication systems, and is implemented using a general purpose silicon technology. Both devices are endowed with a split data/instruction level 1 cache, 16KB wide, while the first one also has a common level 2 cache, 256KB wide. Both devices are equipped with different peripherals: USB, Ethernet, and connections for different types of RAM and Flash memories. In both cases common commercial development boards were used, without having the need to design a custom one. Both systems are endowed with an U-Boot[2] embedded bootloader, able to load the binary to be run via TFTP[3] protocol. During all the attack campaigns, the target application was run on top of a Linux 2.6.15 kernel (DENX distribution) employing an NFS[4] partition as root file-system. To compile our application, we used a cross-compile toolchain based on GCC 3.4 for ARM9 provided by Codesourcery[5].

Fault Injection Methodology Transient faults are injected through the constant underfeeding of the system-on-chip as described in [2]. The manipulation of the power supply causes faults due to the raise of the setup time needed for the logic gates to switch into the correct state: this phenomenon affects in particular high capacitance paths, which are the slowest lines of the circuit. It is interesting to note how the position of the failing line is rather stable and bound to the device sample. To manipulate the power supply, we have detached the on-board power supply and replaced it with an external one. We have used an Agilent precision power supply unit, namely the Agilent 3631A[6]. Note that this is the most expensive part of the setup equipment needed to perform the attack. From a quick browse on the web, we have seen that is possible to buy a pre-owned one for about 1000 USD. Cheaper, and thus less precise, power supply units are suitable as well, provided that an output resolution improvement circuit (which can be built for less than 5 USD) is employed as described in [4]. Before running the attack, we made a brief characterization of the fault model, which confirmed the results presented in [2]. The fault is single-bit flip down, with the position of the flipped bit is rather stable for the each device sample. Only the `load` operations from the main memory are affected by faults. A faulty `load` from memory can affect either a value of the algorithm or an

[1] ARM926EJ-S Technical Reference Manual: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198e/DDI0198E_arm926ejs_r0p5_trm.pdf

[2] Das U-boot Bootloader, http://www.denx.de/wiki/U-Boot

[3] RFC 1350, http://tools.ietf.org/html/rfc1350

[4] RFC 1813, http://tools.ietf.org/html/rfc1813

[5] GNU Toolchain for ARM Processors, http://www.codesourcery.com/sgpp/lite/arm

[6] E3631A 80W Triple Output Power Supply Datasheet, http://www.home.agilent.com/upload/cmc_upload/All/663xseries_datasheet_Jan06.pdf

instruction. In practice, instructions are rarely affected, since in our case the instruction cache achieves a very low miss rate due to the very low number of branches in the algorithm [4]. This results in a very regular behavior of the chip which, provided the voltage tuning is properly done, does not crash while all the needed faults are collected. Therefore, we exploit only the faults that affect data `load`s from the main memory.

### B. Performance Evaluation

We now present the experimental results of the new attack considering three AES software implementations [8], which differ for the various time to memory tradeoff strategy employed. In particular, the fastest of the three requires four lookup tables (called T-TABLEs) while the slowest one only one `S-box` table. The third implementation employs only one T-TABLE and obtains the values of the words from the other three tables through a rotation operation. Since the CPU caches can influence the injection of faults, the time-memory tradeoffs exposed by these three implementations allow to precisely ascertain the impact of the ARM9 data caching policies on the effectiveness of the attack. All the CPU caches (both L1 and L2 when available) were *enabled* during the experiments and the clock frequency set to the maximum one supported, in order to evaluate the attack in unsimplified working conditions. Using the under-voltage fault injection technique, we collected 100KB of faulty ciphertexts for each of the three implementations (for AES-128, -192, -256, respectively) from 2000 different plaintexts and processed them offline on an Intel Core i7–920 (over-)clocked at 4.0GHz and running Ubuntu Linux 9.04. The attack was implemented in `C++` using POSIX standard threads in order to split the load on the eight logical cores of the machine. The results

Table I
ATTACK PERFORMANCES
WHILE PROCESSING 100KB OF FAULTY CIPHERTEXTS

| | **Algorithm IV.3** | |
|---|---|---|
| **Key Size [bit]** | 192 | 256 |
| **Required Faults** | 10 | 20 |
| **Execution Time** | $2'30''$ | $3'$ |
| **Collection Time** | $20''$ | $30''$ |

of the experimental campaign are shown in Table I. The results refer only to the single T-TABLE implementation, since the device in our possess experiences only a single flip down in the 28-th bit of a word, and the single T-TABLE implementation is the only one that distributes the faults among different bytes of the word. As it can be seen from the second row of Table I, the attack requires 10 single-bit faults for the AES-192 to retrieve the second half of the penultimate round key (five for each column of the key), and 20 faults for the AES-256. The execution time row tells that, starting from the raw data produced by the device, only $2'30''$ are required to retrieve the cipher key of AES-192 and 3 minutes for AES-256. The last row of Table I reports that, on average, $20''$ and $30''$ seconds of constant underfeeding of the device suffice to produce the number of faults required to carry out the attack. In other words, the attacker needs the physical control of our device for at most 30 seconds in order to find the cipher key.

## VI. CONCLUSION

In this paper we propose a new differential fault attack against AES implementations of any key size, without being bound to any key scheduling strategy. This enables us to successfully extend the attack to any possible revision of the AES standard involving those parameters, or to attack AES implementations employing proprietary key schedules. The attack relies on the injection of single bit faults: this has been proven viable through a low cost fault injection methodology, based on underfeeding the computing device. We have practically validated the feasibility of the attack carrying a successful one against multiple instances of two chips, etched with a different technological process and based on the widely deployed ARM926EJ-S. Two possible types of countermeasures against this fault injection technique can be adopted: either employing forms of redundant computation (replicate the execution of the algorithm or insert of error correcting codes) or preventing the underfeeding through proper voltage probes inserted in the computing circuit. These methodologies imply additional costs either in computation time or in design effort of the device and thus need to be properly investigated to choose an optimal solution for the target application.

## REFERENCES


[1] N. Selmane, S. Guilley, and J.-L. Danger, "Practical Setup Time Violation Attacks on AES," in *EDCC-7'08: Proceedings of the 2008 Seventh European Dependable Computing Conference*. Washington, DC, USA: IEEE CS, 2008, pp. 91–96.

[2] A. Barenghi, G. M. Bertoni, E. Parrinello, and G. Pelosi, "Low Voltage Fault Attacks on the RSA Cryptosystem," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 23–31.

[3] F. Khelil, M. Hamdi, S. Guilley, J.-L. Danger, and N. Selmane, "Fault Analysis Attack on an FPGA AES Implementation," in *NTMS*, A. Aggarwal et al., Eds. IEEE, 2008, pp. 1–5.

[4] A. Barenghi, G. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi, "Low Voltage Fault Attacks to AES and RSA on General Purpose Processors," Cryptology ePrint Archive, Report 2010/130, 2010, http://eprint.iacr.org/.

[5] W. Li, D. Gu, Y. Wang, J. Li, and Z. Liu, "An Extension of Differential Fault Analysis on AES," *International Conference on Network and System Security*, vol. 0, pp. 443–446, 2009.

[6] J. Takahashi and T. Fukunaga, "Differential Fault Analysis on AES with 192 and 256-Bit Keys," Cryptology ePrint Archive, Report 2010/023, 2010, http://eprint.iacr.org/.

[7] A. Moradi, M. T. M. Shalmani, and M. Salmasizadeh, "A Generalized Method of Differential Fault Attack Against AES Cryptosystem," in *CHES*, ser. Lecture Notes in Computer Science,vol. 4249. Springer, 2006, pp. 91–100.

[8] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[9] P. Dusart, G. Letourneux, and O. Vivolo, "Differential Fault Analysis on A.E.S," *CoRR*, vol. cs.CR/0301020, 2003.

[10] G. Piret and J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD," in *CHES*, ser. Lecture Notes in Computer Science, vol. 2779. Springer, 2003, pp. 77–88.