Low Voltage Fault Attacks to AES

Alessandro Barenghi*, Guido M. Bertoni†, Luca Breveglieri*, Mauro Pellicioli* and Gerardo Pelosi‡
*Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano (MI), Italy
Email: {barenghi,brevegli}@elet.polimi.it, mauro.pellicioli@mail.polimi.it
†STMicroelctronics, 20041 Agrate Brianza (MB), Italy
Email: guido.bertoni@st.com
‡Dipartimento di Ingegneria dell'Informazione e Metodi Matematici
Università degli Studi di Bergamo, 24044 Dalmine (BG), Italy
Email: gerardo.pelosi@unibg.it

***Abstract*—This paper presents a new fault based attack on the Advanced Encryption Standard (AES) with any key length, together with its practical validation through the use of low voltage induced faults. The CPU running the attacked algorithm is the ARM926EJ-S: a 32-bit processor widely deployed in computer peripherals, telecommunication appliances and low power portable devices. We prove the practical feasibility of this attack through inducing faults in the computation of the AES algorithm running on a full fledged Linux 2.6 operating system targeted to two implementations of the ARM926EJ-S on commercial development boards.**

I. Introduction

The security of devices running cryptographic primitives not only depends on the underlying mathematical assumptions, but also strongly relies upon a sound implementation of the target hardware or software platform. Fault attacks aim at inducing errors during the computation of a cryptographic primitive by altering either the control flow or the internal-state data of the cryptographic algorithm implemented on the target device. Depending on the fault model, the erroneous cryptograms may leak sufficient information to extract secret key material from the target implementation. This paper proposes a new attack on a software implementation of AES, able to recover the secret key of the cipher regardless of key length, key scheduling strategy or number of rounds, under the assumption of a known plaintext attack. This assumption is realistic, since in many cases security devices embed cryptographic keys for Digital Rights Management (DRM) verification inside the chip in a non-readable manner, while the attacker is free to choose the plaintext to encrypt. The fault model assumed to carry out the attack is a single byte transient fault occurring once throughout the execution of the algorithm. More precisely, we will focus on exploiting a fault injected in the internal state of the cipher, instead of exploiting a fault in the key schedule such as in [1]. This allows us to attack implementations of AES where the key schedule has been precomputed. To achieve this, we employ a non-invasive fault model based on underfeeding a general purpose CPU, which has been first described in [2], [3] and successfully used in [4] where the authors detailed a fault-attack on a FPGA based SPARC system and demonstrated how voltage regulations can be exploited to target a RSA signature algorithm implementation included into the OpenSSL library. The methodology involves keeping the level of voltage supplied to the processor constantly lower than the nominal one during the whole computation, without any particular timing, thus requiring only a very simple and cheap workbench. Since our attack technique is able to distinguish exploitable faults from non exploitable ones, we are able to discard the faults which happen in the non sensitive parts of the algorithm from the faults used to recover the key. The remainder of the paper is organized as follows. Section II describes the method employed to inject the faults in the computation. Section III presents the attack technique and Section IV reports on the experimental evaluation. Finally, Section V summarizes our conclusions.

II. Target Architecture and Fault Injection Method

This section describes the target hardware architecture employed, in order to provide grounds for reproducibility and delineates a realistic target against which the attack has been carried out.

Target Architecture The target architecture is an ARM926EJ-S [5] CPU: a 32-bit RISC Harvard architecture CPU with 16 general purpose registers and a 5-stage pipeline. The ARM^{TM} processor has a full MMU, separate data and instruction caches each 16KB wide, and is endowed with a memory write-back buffer. Two different systems-on-chip (SoC) containing the ARM926EJ-S CPU were chosen, one oriented to the computer peripherals, the other to the telecommunications market, implemented in two different silicon technologies, in order to validate the applicability of the fault induction technique against a wide range of targets. Both CPUs are embedded in a SoC mounted on a development board, which is equipped with DDR DRAM, on-board Flash storage, USB and Serial RS-232 interfaces, and a 100Mbps Ethernet network card. The telecommunication oriented SoC is also endowed with a level 2 shared data and instruction cache 128KB wide, and supports a working frequency up to 266MHz, twice that of the SoC intended for computer peripherals. The system is endowed with a U-Boot embedded bootloader, which can load the binary to be run via the TFTP protocol. All the attacks on cryptosystems were performed using a Linux 2.6.15 kernel (DENX distribution) employing an NFS partition as the root file-system. All the binaries were compiled with the GCC 3.4 based tool chain for ARM9^{TM}

into regular ELF binaries, which were run starting from the shell.

Fault Injection Technique The fault injection technique is the constant underfeeding of the computing circuit as described in [3]. The effect of a constant underfeeding of the circuit is a rise in the setup time needed for the logic gates to switch into a stable state. Thus, if the clock speed is kept the same as suggested for the working conditions of the correctly fed circuit, and the feeding voltage level is gradually lowered, some of the slower logic paths will fail to setup properly. In a scenario where frequency scaling of the chip is implemented, it is always possible to induce setup time violations through lowering the voltage until the failures start happening even when the chip is running at the lowest scaling frequency supported. The experimental workbench employed to induce setup time violations is comprised of: the device under attack; a precision power supply unit (PSU); an Agilent 34420A voltmeter; and a PC running Linux, in order to provide the board both an NFS root partition via Ethernet and an easy way to control it via a serial terminal. The employed PSU has precision of 0.01V in the output tuning, which was further enhanced to 1mV by employing a resistive voltage divider. In order to decouple the voltage divider from the actual load, we employed an operational amplifier in source follower configuration. The equipment can be easily found at a low price, thus also enabling attackers with a small budget to induce exploitable faults. The characterization of the fault model induced on the chip led us to obtain the same results as the ones exposed in [3]: the only operations affected by the underfeeding of the chip are the `LOAD` operations. This can be ascribed to the fact that the `LOAD` instructions involve very long data paths, which are the most likely to show bit setup failures due to the lower transition rate caused by the underfeeding. The `STORE` instructions are not affected since the off-chip memory is separately fed and the CPU is endowed with a memory writeback buffer which considerably shortens the critical paths on the lines. Arithmetical and logical operations do not exhibit faulty behavior either: this is to be ascribed to the high level of optimization of the design of the functional units, which attempts to route very fast lines for them. All the faulty loads are characterized by bit flip-down errors: no events of flip-up ever occur. In the voltage range useful to induce exploitable errors in the computation, there is only a single bit stuck at zero for a single `LOAD` operation, and its position is fixed. This voltage range can be experimentally determined through undervolting the actual chip sample until the faults that happen are sparse enough, since it depends on the working voltage and clock frequency of the chip. The effects of a faulty `LOAD` instruction on the computation may concern either the value of the computed data or the control flow of the algorithm, thus causing different outcomes depending on whether the load is related to an instruction fetch (*instruction swapping error*) or to a data access (*data load error*). Data load errors are represented as a transient change in the value of a t-bit wide variable c during the execution of a software routine. The faulty value $\widetilde{c}$ equals the correct value c minus a power of two 2^{ε}, where ε is the position of the fault. Possible values of ε are expressed in the form $\varepsilon{=}k\,w{+}i$ with w equal to the word length, $i{\in}[0,\ w-1]$ and $k{\in}[0,\ \frac{t}{w}]$. Instruction swapping errors may occur in the case the faulty bit is part of either the instruction opcode fetched (in which case, the instruction changes to a different one) or if it is part of the condition field (in which case, a conditioned instruction is executed upon a different condition).

III. Attacks to AES

A. Overview of the AES Block Cipher

AES is an iterated block cipher that corresponds to a block size restricted version of the Rijndael [6], and can encrypt and decrypt 128-bit wide plaintext blocks using a 128-bit, 192-bit or 256-bit key. In software, AES can be implemented with a fully symmetric structure using only bitwise `xor` operations, table-lookups and 1-byte shifts [6]. The cipher is designed to execute a number of round transformations on the input plaintext, where the output of each round is the input to the next one. The number of rounds r is determined by the key length: 128-bit,192-bit and 256-bit keys use 10, 12 and 14 rounds, respectively. Each round is composed by the same steps, except for the initial where an extra addition of a round key is inserted, and for the final round where the last step (MixColumns) is skipped. Each step operates on 16 bytes of data (referred as the internal *state* of the cipher) generally viewed as a 4×4 matrix of bytes or viewed as an array of four 32-bit words, where each word corresponds to a column of the *state* table. The four round steps are: SubBytes (byte substitution by an S-box, i.e. a lookup table for enforcing a non linear function), ShiftRows (cyclical shifting of bytes in each row for realizing a inter-word byte diffusion), MixColumns (linear transformation which mixes column state data for intra-word inter-byte diffusion), and AddRoundKey (`xor` addition of a scheduled round subkey, for blending together the key and the state). The specification of the AES algorithm includes the description of a KeySchedule procedure which is responsible for computating of each 16-byte round key $k^{(i)}$, $0 \le i \le r$, given the global input key k, thus expanding the original key material in $r{+}1$ subkeys. The AES key scheduling process expands the cipher key k into a total of $4(r{+}1)$ 32-bit words with $r \in \{10, 12, 14\}$ according to whether the cipher key length s is equal to 4, 6 or 8 words, respectively. The resulting key schedule consists of a linear array of 32-bit words, denoted $W[0, \ldots, 4(r{+}1){-}1]$. The first s words of W are loaded with the user supplied key. The remaining words of W are updated according to the following rule:

```
for $i = s, \ldots, 4(r+1)-1$ do
    if $i \equiv 0 \mod s$ then
        $W[i] = W[i-s] \oplus S[W[i-1] <<< 8]) \oplus RCON[i/s]$
    else if $s = 8$ and $i \equiv 4 \mod s$
        $W[i] = W[i-s] \oplus S[W[i-1]]$
```

else $W[i] = W[i-s] \oplus W[i-1]$

$RCON[\ldots]$ is an array of 32-bit constants, which is useful to eliminate symmetry or similarity amid the generated subkeys in each round: $RCON[j]{=}\langle RC[j], 0, 0, 0\rangle$ where the byte values $RC[j]{=}2{*}RC[j{-}1]$ with $j{=}0, \ldots, 9$, $RC[0]{=}1$, and the multiplication defined over $\mathbb{F}_{2^8}$. $S[\ldots]$ is the array of precomputed constants corresponding to the substitution map of the cipher, and $<<< 8$ denotes a 1-byte left rotation.

B. Low Voltage Induced Errors on AES

Given the error model on the loaded data presented in Section II, we may expect that the errors induced during the execution of an AES software implementation affect the results through alterations in the values loaded during each memory lookup. In particular, we drive the feeding voltage of the device in such a way that only a single bit of the state of the cipher is affected by a lone fault during each encryption. The presented attack works under the assumption of a single byte error, which, due to the byte oriented design of the cipher, leaks information on the internal state precise enough to be exploited. Since the practically induced faults affect only a single bit at once, it is practically viable to employ the fault injection technique presented in Section II to conduct the new attack successfully.

We now introduce a new generalized attack technique to recover any round subkey from the AES cipher, regardless of both the key scheduling algorithm (i.e. regardless of the fact that the round subkeys are computed through the standardized KeySchedule algorithm or filled completely with a much longer cipher key), and the key length or the number of rounds (even if exceeding the number of rounds set by the standard). This is the first result concerning the practical implementation of an attack to a general AES instance instead of the largely scrutinized AES-128. The effectiveness of the algorithmic routines described in the following, is independent from the various optimizations and time-memory tradeoffs employed to speed up the AES execution (see Section IV). For the sake of clarity, the proposed attack can be followed referring to the plain AES implementation previously summarized. The attack focuses on the encryption primitive and requires a fault free ciphertext and a small number of faulty ciphertexts generated from the same plaintext. The goal is either to derive the s bytes composing the cipher key k, by recovering enough key material from the last round subkeys $k^{(r)}, k^{(r-1)}, \ldots$, or to retrieve the subkeys of each round. Note that the standard AES specification does not allow the reconstruction of the cipher key by knowing fewer than s consecutive bytes of the round subkeys (f.i., it is necessary to recover the last two subkeys to retrieve the cipher key of a standard AES-256).

Piret and Quisquater in [7] introduced a differential fault attack technique working against Substitution-Permutation Networks, under the hypotheses of a single byte error occurring between the last and last-but-one linear diffusion operation of the cipher, and the availability of pairs of faulty and fault free ciphertexts corresponding to the same plaintext. The attack in [7] provides a distinguishing criteria for the useful faults, using the diffusion property of the last MixColumns in order to determine whether an erroneous ciphertext was caused by a single byte difference between the last and last-but-one MixColumns operation. 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. Consider a generic instance of the AES cipher with r rounds, keylength of s words and 16-byte input/output blocks. 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 $\widetilde{c}_u, c_u$, $0{\le}u{\le}3$, are 32-bit words. 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. Indeed, 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.: $\Delta{=}\{\langle\delta_0, 0, 0, 0\rangle,\ \langle 0, \delta_1, 0, 0\rangle,\ \langle 0, 0, \delta_2, 0\rangle,\ \langle 0, 0, 0, \delta_3\rangle\}$,

Function III.1: `GetLastRoundKeyWord(` j `)`

```
Input  : $j \in \{0,1,2,3\}$, word index of the last round subkey
Output : $k_j^{(r)}$, $j$-th word of the last round subkey $k^{(r)}$
Data   : $\Delta{=}\{\langle\delta_0,0,0,0\rangle, \langle 0,\delta_1,0,0\rangle, \langle 0,0,\delta_2,0\rangle, \langle 0,0,0,\delta_3\rangle\}$,
            $\delta_u \in \{x,\ x\,2^8,\ x\,2^{16},\ x\,2^{24} \mid 0{\le}x{<}2^8\}$, $0{\le}u{\le}3$,
            $|\Delta| = 255 \times 4 \times 4$;
         $\Delta'{=}\{\, d \mid d \leftarrow \textsc{MixColumns}(\delta),\ \forall\delta \in \Delta\,\}$;
         $\zeta$ 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 $\zeta_i,\ 0 \le i \le 3$
1  Generate a random plaintex $p$
   /* Record a faulty and fault-free ciphertext                    */
2  $c^{(r)} \leftarrow$ AES_Encrypt$_k(p)$ /* k: unknown cipher key              */
3  $\widetilde{c}^{(r)} \leftarrow$ Faulted_AES_Encrypt$_k(p)$ /* k: unknown cipher key    */
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      $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \leftarrow \langle 0,0,0,0\rangle$
8      $\zeta_j \leftarrow \textsc{InvSubBytes}(w \oplus \bar{k}) \oplus \textsc{InvSubBytes}(\widetilde{w} \oplus \bar{k})$
9      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     Generate a random plaintext $p$
       /* Record a faulty and fault-free ciphertext                */
12     $c^{(r)} \leftarrow$ AES_Encrypt$_k(p)$ /* k: unknown cipher key          */
13     $\widetilde{c}^{(r)} \leftarrow$ Faulted_ AES_Encrypt$_k(p)$
14     Carve words $\widetilde{w}, w$ both according to $j$ and taking into
       account the last ShiftRows operation
15     foreach $\bar{k} \in L$ do
16         $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \leftarrow \langle 0,0,0,0\rangle$
17         $\zeta_j \leftarrow \textsc{InvSubBytes}(w \oplus \bar{k}) \oplus \textsc{InvSubBytes}(\widetilde{w} \oplus \bar{k})$
18         if $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \notin \Delta'$ then $L \leftarrow L \setminus \{\bar{k}\}$
19 return $k_j^{(r)}$ /*   $L = \{\bar{k}\}$,  $\bar{k} = k_j^{(r)}$                     */
```

$\delta_u \in \{x,\ x\,2^8,\ x\,2^{16},\ x\,2^{24} \mid 0 \le x < 2^8\},\ 0 \le u \le 3$. 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 Δ: $\Delta^{'} = \{\, d \mid d \leftarrow \text{MixColumns}(\delta),\ \forall\ \delta \in \Delta \,\}$.

The diffusion layer (MixColumns) is not perfect and only spreads a single bit difference on a quarter of the inner state (i.e. diffuses a single byte change over a single column (word) of the inner state). The exploitation of this peculiarity of the AES diffusion layer allows to conceive a 32-bit word based implementation of the attack, which retrieves the whole last round subkey $k^{(r)}$ in four passes, sweeping a candidate space of 2^{32} values at most, for each word $k_j^{(r)}$, $j{\in}\{0,1,2,3\}$. Function III.1 details the procedure to recover one of the last subkey words and takes as input the list $\Delta^{'}$ of all the differences that may occur just after the last MixColumns operation (in the $(r{-}1)$-th round), and the position j corresponding to the target word of the last subkey $k_j^{(r)}$ it is intended to retrieve. As a first step (lines 3–6), Function III.1 records a faulty $\widetilde{c}$ and fault-free ciphertext c, carves words $\widetilde{w}$, w according to the target position index j and taking into account the last ShiftRows operation. Then, for each possible value of the j-th word of the last subkey, $k^{(r)}$, it computes the difference ζ_j between the state tables corresponding to c and $\widetilde{c}$ just after the last MixColumns operation: $\zeta_j \leftarrow \text{InvSubBytes}(w \oplus \bar{k}) \oplus \text{InvSubBytes}(\widetilde{w} \oplus \bar{k})$. If ζ_j is included in the set Δ' then the value of the corresponding subkey word $\bar{k}$ is inserted in a list L of candidate words. Subsequently (lines 7–12), until L contains only a single value, another pair of faulty and fault-free ciphertext is collected. This usually requires only one extra faulty and fault free ciphertext to narrow down the key candidates to a single one, as it is easily verifiable in practice. Then, for each candidate key in L the differential value corresponding to the new faulty and fault-free ciphertexts is computed in order to verify that such value is effectively included in $\Delta^{'}$. If the value $\bar{k}$ is not included in Δ', the candidate is removed from the list L. At the end of this sieving phase, L will contain a single value for $k_j^{(r)}$. Algorithm III.3 rolls back both the last (r-th) and the last-but-one (($r{-}1$)-th) rounds of the AES cipher, thus retrieving the last two subkeys: $k^{(r)}$ and $k^{(r-1)}$. The implemented method allows to circumvent the security criteria in the design of the AES cipher, thus highlighting a new serious vulnerability even when employing the parameters recommended for maximum security of the cipher (256-bit key). In order to recover the last subkey $k^{(r)}$, the algorithm (lines 4–5) iterates Function III.1 for each of the four words: $\langle k_0^{(r)}, k_1^{(r)}, k_2^{(r)}, k_3^{(r)} \rangle$. In order to retrieve the $(r{-}1)$-th subkey, $k^{(r-1)}$, we collect erroneous ciphertexts resulting from a single byte fault that occurred between the last-but-one MixColumns in the $(r{-}2)$-th round, and the last-but-two MixColumns operation in the $(r{-}3)$-th round. This kind of fault will result in a complete corruption of the state $\widetilde{c}^{(r-1)}$ by the end of the last-but-one round. In order to distinguish the induced errors which respect this hypothesis from the non useful ones, we arranged Function III.2 (GetDifferential) to cope with the diffusing effect of the last MixColumns operation and to bypass the obfuscation provided by the $(r{-}1)$-th AddRoundKey. Given a random plaintext p and the last subkey $k^{(r)}$, Function III.2 returns the differential between the corresponding j-th words of the faulty free state matrix and the faulty state matrix after the SubBytes of the last-but-one round ($j{\in}\{0,1,2,3\}$). Function III.2 repeatedly computes a new faulted ciphertext $\widetilde{c}^{(r)}$ and derives the corresponding output of the last-but-one round, $\widetilde{c}^{(r-1)}$ in order to consider the differential value $\widetilde{c}^{(r-1)}{\oplus}c^{(r-1)}$. This value (line 6) can be safely transformed through an InvMixColumns since the operation is linear w.r.t the `xor`, and subsequently passed through a InvShiftRows primitive to realign the bytes, obtaining the alleged $(r{-}1)$-th SubBytes output as $\langle \delta_0, \delta_1, \delta_2, \delta_3 \rangle$. A useful fault for our purposes is recognized checking whether there is a single non-zero word δ_j. In case the fault is not useful, the function discards the faulty ciphertext and starts examining a new one. Once a useful fault has been found, the function GetDifferential returns the non zero word differential, δ_j, and its relative position within the state matrix (j), together with the corresponding values $c^{(r-1)}$, $\widetilde{c}^{(r-1)}$. Algorithm III.3 uses Function III.1 to extract the last round subkey $k^{(r)}$ (line 2), generates a further random plaintext (line 4) and searches for convenient faulty ciphertexts (with a single byte fault between the last-but-one and the last-but-two MixColumns) until it finds a set of candidate words for the intermediate value of the correct ciphertext at the output of the SubBytes stage (in the $(r{-}1)$-th round) (lines 5–12). Each candidate word for the alleged output of the $(r{-}1)$-th SubBytes is stored in a

Function III.2: `GetDifferential`$(p, k^{(r)})$

```
Input  : $p$, random plaintext;
         $k^{(r)}$, last round key;
Output: $\langle c^{(r-1)}, \widetilde{c}^{(r-1)}, \delta_j, j \rangle$,
         $c^{(r-1)}$: output of the last-but-one round in a fault
               free computation
         $\widetilde{c}^{(r-1)}$: output of the last-but-one round in a faulty
               computation
         $\delta_j$: one word difference between faulty and faulty free
               state after the SubBytes of the $(r{-}1)$-th;
         $j \in \{0,1,2,3\}$: position of the only non-zero word in
               the aforementioned difference
1 $c^{(r)} \leftarrow$ AES_Encrypt$_k(p)$ /* k: unknown cipher key */
  /* fault-free last-but-one round output */
2 $c^{(r-1)} \leftarrow$ InvSubBytes (InvShiftRows ($c^{(r)} \oplus k^{(r)}$))
3 repeat
4     $\widetilde{c}^{(r)} \leftarrow$ Faulted_ AES_Encrypt$_k(p)$ /* k: unknown cipher key */
5     $\widetilde{c}^{(r-1)} \leftarrow$ InvSubBytes (InvShiftRows($\widetilde{c}^{(r)} \oplus k^{(r)}$))
      /* MixColumns in the $(r-1)$-th round */
6     $\langle \delta_0, \delta_1, \delta_2, \delta_3 \rangle \leftarrow$ InvShiftRows(InvMixColumns($\widetilde{c}^{(r-1)} \oplus c^{(r-1)}$))
7 until $\exists\,!\ j \in \{0,1,2,3\} \mid \delta_j \neq 0$
  /* Here $\delta_j \neq 0$, as lone non-zero differential word at the
  output of the $(r{-}1)$-th SubBytes, is due to a single byte fault
  happened between the $(r{-}2)$-th and the $(r{-}3)$-th MixColumns */
8 return $\langle c^{(r-1)}, \widetilde{c}^{(r-1)}, \delta_j, j \rangle$
```

Algorithm III.3: FAULT ATTACK TO THE AES

```
Output: $(k^{(r-1)}, k^{(r)})$, subkeys of the last two rounds;
Data: $\Delta=\{\langle\delta_0,0,0,0\rangle, \langle 0,\delta_1,0,0\rangle, \langle 0,0,\delta_2,0\rangle, \langle 0,0,0,\delta_3\rangle\}$ ,
      $\delta_u \in \{x,\ x\,2^8,\ x\,2^{16},\ x\,2^{24} \mid 0 \le x < 2^8\}$, $0 \le u \le 3$,
      $|\Delta| = 255 \times 4 \times 4$;
      $\Delta' = \{ d \mid d \leftarrow \textsc{MixColumns}(\delta),\ \forall \delta \in \Delta \}$;
      $\zeta$ 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 $\zeta_i$, $0 \le i \le 3$
1  begin
       /* $k^{(r)} = \langle k_0^{(r)}, k_1^{(r)}, k_2^{(r)}, k_3^{(r)}\rangle$ */
2      foreach $j \in \{0,1,2,3\}$ do $k_j^{(r)} \leftarrow \textsc{GetLastRoundKeyWord}(j)$
       /* $L_j \leftarrow \{\ldots\}$ will include the candidate values of faulty
       free ciphertext words resulting after the SubBytes in the
       $(r-1)$-th round. */
3      foreach $j \in \{0,1,2,3\}$ do $L_j \leftarrow \emptyset$
4      Generate a random plaintext $p$
5      repeat
6          $\langle c^{(r-1)}, \widetilde{c}^{(r-1)}, \delta_j, j\rangle \leftarrow \textsc{GetDifferential}(p, k^{(r)})$
7          foreach $w \in \{0, \ldots, 2^{32}-1\}$ do
               /* Guess on the values of the $j$-th word of a
               candidate faulty-fault free pair of state matrices
               after the $(r-1)$-th SubBytes */
8              $\widetilde{w} \leftarrow w \oplus \delta_j$
9              $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \leftarrow \langle 0,0,0,0\rangle$
10             $\zeta_j \leftarrow \textsc{InvSubBytes}(\widetilde{w}) \oplus \textsc{InvSubBytes}(w)$
               /* $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle$: differential value at the output
               of the $(r-2)$-th round */
11             if $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \in \Delta'$ then $L_j \leftarrow L_j \cup \{w\}$
12     until $\forall\, j \in \{0,1,2,3\}, L_j \neq \emptyset$
13     while $\exists\, j \in \{0,1,2,3\},\ |L_j| > 1$ do
14         $\langle c^{(r-1)}, \widetilde{c}^{(r-1)}, \delta_j, j\rangle \leftarrow \textsc{GetDifferential}(p, k^{(r)})$
15         foreach $w \in L_j$ do
16             $\widetilde{w} \leftarrow w \oplus \delta_j$
17             $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \leftarrow \langle 0,0,0,0\rangle$
18             $\zeta_j \leftarrow \textsc{InvSubBytes}(\widetilde{w}) \oplus \textsc{InvSubBytes}(w)$
               /* $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle$: differential value at the output
               of the $(r-2)$-th round */
19             if $\langle\zeta_0,\zeta_1,\zeta_2,\zeta_3\rangle \notin \Delta'$ then $L_j \leftarrow L_j \setminus \{w\}$
       /* $L_0 = \{\bar{w}_0\}$, $L_1 = \{\bar{w}_1\}$, $L_2 = \{\bar{w}_2\}$, $L_3 = \{\bar{w}_3\}$ */
20     $k^{(r-1)} \leftarrow c^{(r-1)} \oplus \textsc{MixColumns}(\textsc{ShiftRows}(\langle \bar{w}_0, \bar{w}_1, \bar{w}_2, \bar{w}_3\rangle))$
21     return $(k^{(r-1)}, k^{(r)})$
22 end
```

different set L_j, $j \in \{0,1,2,3\}$, depending on the position index of the corresponding 32-bit word into the state matrix. Once all the lists L_j are filled with at least a single candidate word, a pruning phase takes place (lines 13–19). This second phase aims at reducing the number of candidates contained in each list, through further sieving of non-compatible candidates. It is also possible to skip this pruning phase altogether if the candidate lists are small enough to allow a brute force search over all the possible keys. The pruning phase is performed generating new faulty ciphertexts and checking which key candidates are still valid through iterating the same criterion used to include the guesses in the candidate lists L_j. In the case a candidate word does not pass the check, it is removed from the list (lines 16–19). The existence of a correct key candidate and the correctness of the fault hypothesis ensure that the lists L_j will never be emptied. After obtaining a single candidate for each of the four words at the output of $(r-1)$-th SUBBYTES, it is possible

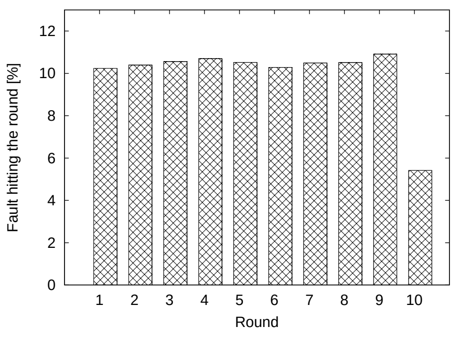

Figure 1. Distribution of single byte faults over the rounds of an AES-128 implementation

to apply SHIFTROWS and MIXCOLUMNS operations in order to find the correct value of the state matrix in input to the last-but-one ADDROUNDKEY. In order to retrieve the $(r-1)$-th round subkey $k^{(r-1)}$, it suffices to compute a further `xor` with $c^{(r-1)}$ (line 20). The procedure described by Algorithm III.3 can be easily adapted to fully invert the effect of any round of the AES algorithm by removing the rounds one by one.

IV. EXPERIMENTAL RESULTS

We now provide experimental evidence of the practicality of the algorithmic techniques exposed in Section III, and report the results of conducting them on an ARM9$^{\text{TM}}$ CPU. We report figures of merit for both the attack strategies proposed in [7] and the framework introduced in the previous sections, which allows us to attack any number of rounds of any AES cipher. All the the CPU caches (both L1 and L2 when available) were *enabled* during the experiments and the frequency set to the maximum one supported, in order to evaluate the attack in unsimplified working conditions. The enciphering procedure of AES is amenable to several software implementations which trade-off memory and computational resources [6]. We considered three different implementations: the first one combines the different steps of the round transformation in a set of four lookup tables (named T-tables), which is also the reference implementation used in OpenSSL project; the other two implementations use a single T-table and the standard S-box, respectively. Since the CPU caches can have a mitigating effect on the injection of faults, the time-memory tradeoffs exposed by these three implementations allow to precisely ascertain the impact of the ARM9$^{\text{TM}}$ data caching policies on the effectiveness of the attack. The feasibility of the attack is dependent on the injection of one byte faults in a specific word of a round of the cipher. To this end, it is important to estimate the statistical distribution of faults over the cipher states. Figure 1 depicts the faults spread on the first 10 rounds of the AES-128 algorithm, obtained through collecting 100K faults and classifying them by the round they hit. This was done through inverting the faulty ciphertexts with the known key and calculating the differences between each state of the correct and the erroneous runs until the single byte difference was found.

The depicted data refer to the 4 T-table implementation of AES; the results about the other implementations are quite similar. As the figure shows, the faults are almost equally distributed on the first $r-1$ rounds of the cipher, except for the last one which has a sensibly lower probability to be hit. The estimated fault distributions for AES-192 and AES-256 algorithms are analogous to the reported one except for the larger number of rounds.

Table I
Percentages of faults hitting each word of the state at the $(r-1)$-th round, over 50k injected faults, reported for O1/O2/O3 gcc optimization levels

State	Faults hitting count [%]		
Word	4 T-tables	1 T-table	S-box
1st	25.1/24.4/24.6	25.0/25.0/25.5	24.8/24.3/20.7
2nd	24.7/25.0/24.1	25.6/25.0/25.1	25.1/25.8/19.2
3rd	24.6/25.1/25.9	23.9/24.3/24.7	24.8/26.2/20.6
4th	25.5/25.3/25.3	25.3/25.5/24.6	25.1/23.6/39.4

A more precise analysis takes into account the fault distribution over the state of a single round considering also the effect of the optimization strategies employed by the compiler on the cipher code. This is mandated by the fact that aggressive optimizations may employ the coalesced instructions of the ARMv5TE architecture which may alter the fault spread over the words of the state. Table I reports the fault spread over the words of the state matrix at the output of the penultimate round of AES-128, averaged over 50K faults for each implementation and evaluated for each optimization level to which the GCC compiler was set. The reported results depict a uniform spread of the faults over all the four words of the inner state of the cipher (regardless of the implementation or the optimization grade of the binary). Therefore, even if the injection of faults cannot be time-driven, it is sufficient to collect more ciphertexts to obtain exploitable faults. We compared the performance of Algorithm III.3 versus the method described in [7]. Thus, in order to further validate the applicability of the presented under-voltage fault model, we collected 50K faulty ciphertexts from 2000 different plaintexts and processed them offline on an Intel$^{®}$ Core^{TM} i7–920 (over-)clocked at 4.0GHz and running Ubuntu Linux 9.04. The algorithms are implemented in `C++` using POSIX standard threads in order to split the load on the four cores of the machine. Table II summarizes the performances of the two attacks for all key lengths of the AES.

Table II
Attack Performances while processing 50K faults

	Alg. [7]	Alg. III.3		
Key Size [bit]	128	128	192	256
Execution Time	$1'$	$1'$	$2'$	$2'21''$
Mem. Footprint [kB]	480	480	500	605
N. of useful faults	8	8	12	16
Avg. N. injected faults	84	84	106	252

The generalized attack to AES (Algorithm III.3) has the same performance of [7] as far as the AES–128 case goes (since it actually uses it), whilst requiring 12 faults for AES–192 (eight for the last subkey and four for the last-but-one subkey), and 16 faults for AES–256 (eight for the last round subkey and eight for the last-but-one subkey). The last row of Table II reports that, on average, 106 and 252 faulty ciphertexts are respectively enough to obtain the 16 correct ones required to retrieve either the AES–192 or the AES–256 cipher key. The measured CPU time, employed to run Algorithm III.3, amounts some minutes with a memory footprint of about 500KB, thus well within the reach of a common desktop. Indeed, it is possible to successfully break the AES cipher through executing on average 100K encryptions with different plaintexts (used to retrieve the r-th subkey) and 2000K encryptions with the same plaintext ($(r-1)$-th subkey). This quantity of ciphertexts can be easily collected by an attacker in posses of the physical device, regardless of the mode of operation of the cipher since the system on chip may be re-set, thus leading to the re-encryption of the same plaintext at will.

V. Conclusion

In this paper we present a new attack to AES, able to break the cipher regardless of the keylength, the number of rounds or the key schedule choice in only a few minutes. The attack proposed has been practically carried against a widely deployed CPU, the ARM926EJ-S, embedded in a realistic, production grade, environment employing a very simple and low cost workbench and not leaving any evidence of tampering with the device. Countermeasures may be based on the introduction of redundancy in the calculations and the appropriate choices are demanded to future researches.

Acknowledgment

This work was partially supported by MIUR in the framework of the PRIN SESAME project.

References

[1] C. H. Kim and J.-J. Quisquater, "New Differential Fault Analysis on AES Key Schedule: Two Faults Are Enough," in *CARDIS*, ser. Lecture Notes in Computer Science, G. Grimaud and F.-X. Standaert, Eds., vol. 5189. Springer, 2008, pp. 48–60.

[2] 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 Computer Society, 2008, pp. 91–96.

[3] 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.

[4] A. Pellegrini, V. Bertacco, and T. Austin, "Fault-Based Attack of RSA Authentication," in *DATE 2010: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, March 2010.

[5] Acorn Risc Machine, "ARM9 Family of General-Purpose Microprocessors, ARM926EJ-S Technical Reference Manual." [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0198e/DDI0198E_arm926ejs_r0p5_trm.pdf

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

[7] 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, C. D. Walter, Çetin Kaya Koç, and C. Paar, Eds., vol. 2779. Springer, 2003, pp. 77–88.