

## “A DFA attack against the AES key schedule”

David Peacham & Byron Thomas

SiVenture

[david.peacham@siventure.com](mailto:david.peacham@siventure.com)

[byron.thomas@siventure.com](mailto:byron.thomas@siventure.com)

### Abstract

In this white paper we present a new form of DFA attack against the AES key schedule for 16-byte keys. We can perform a key retrieval using no more than 12 pairs of correct and faulty ciphertexts, where groups of bytes in the penultimate round subkey are corrupted. Compared to other AES key schedule attacks using DFA (such as [1] and [2]), our attack requires fewer faults, requires less computation, and uses a different fault model which applies in a different set of circumstances. Our fault model assumes each fault occurs during a single iteration of the key scheduling process, such that the fault propagates to bytes generated by subsequent iterations, including other bytes in the same round and all bytes of the final round subkey. The main advantage of this attack over the DFA attack of [3] (which corrupts the state, not the key schedule) is that it can defeat some fault-protected AES implementations where the keys are not rescheduled prior to doing a decipherment check.

### 1. Introduction

In this paper, we present a Differential Fault Analysis (DFA) attack against the AES key scheduling process. DFA attacks have been widely studied in the open literature: see for example [3] and [5]. More recently they have been applied to implementations of AES, as in [3], [1] and [2]. Of these, the attack by Dusart et al [3] is arguably the most practical since it allows a fault affecting any intermediate byte in the penultimate round to leak information about the key. A different form of DFA attack, targeting the AES key schedule was introduced by Giraud in [1] and improved upon by Chen & Yen in [2]. Like those two attacks, we target the AES key schedule for 16-byte key lengths which provides the simplest case for analysis. All the descriptions given in this paper apply specifically to the case of a 16-byte key and 16-byte block, and not necessarily to the other key and block sizes available in AES and Rijndael.

Our attack differs from the other AES DFA attacks in a number of aspects. Firstly, our attack is capable of revealing an entire 16-byte AES key in no more than twelve pairs of correct and faulty ciphertexts. With some additional processing, as few as four pairs may be sufficient. By contrast, Chen & Yen's improved key schedule attack requires one uncorrupted ciphertext and an average of 22 faulty ciphertexts, all of which correspond to the same computation. Although the total number of ciphertexts they require is thus one less than for our attack (which requires a maximum of 12 pairs), our attack requires fewer faults, which will be more efficient for the attacker, since uncorrupted ciphertexts are far easier to generate.

Our attack also differs in the amount of computation time required and the fault model. Giraud's attack requires 2 brute-force searches across 32-bit values. Although this is a feasible amount of computation, it is slow: he quotes a runtime of 2 days in his original

paper. Chen & Yen managed to decrease this to a 24-bit search which takes less than a minute. Our attack reduces the complexity further to a few searches across 8-bit and 16-bit values. The original key schedule fault model used in both the previous papers assumes that faults occur on round key bytes whilst they are stored between key scheduling steps. Our new fault model assumes the faults occur during the execution of the key scheduling process.

The rest of this paper is organised as follows. In Section 2, we describe our fault model and contrast it with the model introduced by Giraud. Our analysis methodology is presented in Sections 3 to 8. Section 9 discusses how we confirmed that our proposed fault model is plausible in practice. Section 10 discusses countermeasures, with the conclusion given in Section 11.

## 2. Fault model

Similarly to the attacks of Giraud and Chen & Yen, our attack targets the key scheduling process. We first remind the reader of the AES key scheduling algorithm before describing the fault model. The AES key scheduling process is iterative with each iteration generating 32 bits of a round key. The 16-byte key is first split into four 32-bit words labelled  $w_0, w_1, w_2, w_3$ , where  $w_0$  holds the most significant 32 bits of the key,  $w_1$  the next most significant bits, and so on. The values of  $w_i$  encode the entire key schedule for  $i$  in  $0..43$ , such that the whitening key is  $w_0 \parallel w_1 \parallel w_2 \parallel w_3$ , the round 1 key is  $w_4 \parallel w_5 \parallel w_6 \parallel w_7$ , and so on. Each iteration takes the result of the previous iteration as input, and iterations are grouped in sets of four: one non-linear transformation followed by three linear steps. So, for example, to generate  $w_4, w_3$  is first rotated one byte left, then looked up using the S-box, then XOR'd with the round 1 key schedule constant ( $R_1$ ) and then XOR'd with  $w_0$ . For the remaining three linear steps,  $w_n$  is generated by XORing  $w_{n-1}$  with  $w_{n-4}$ . The process is illustrated in Figure 1 below.

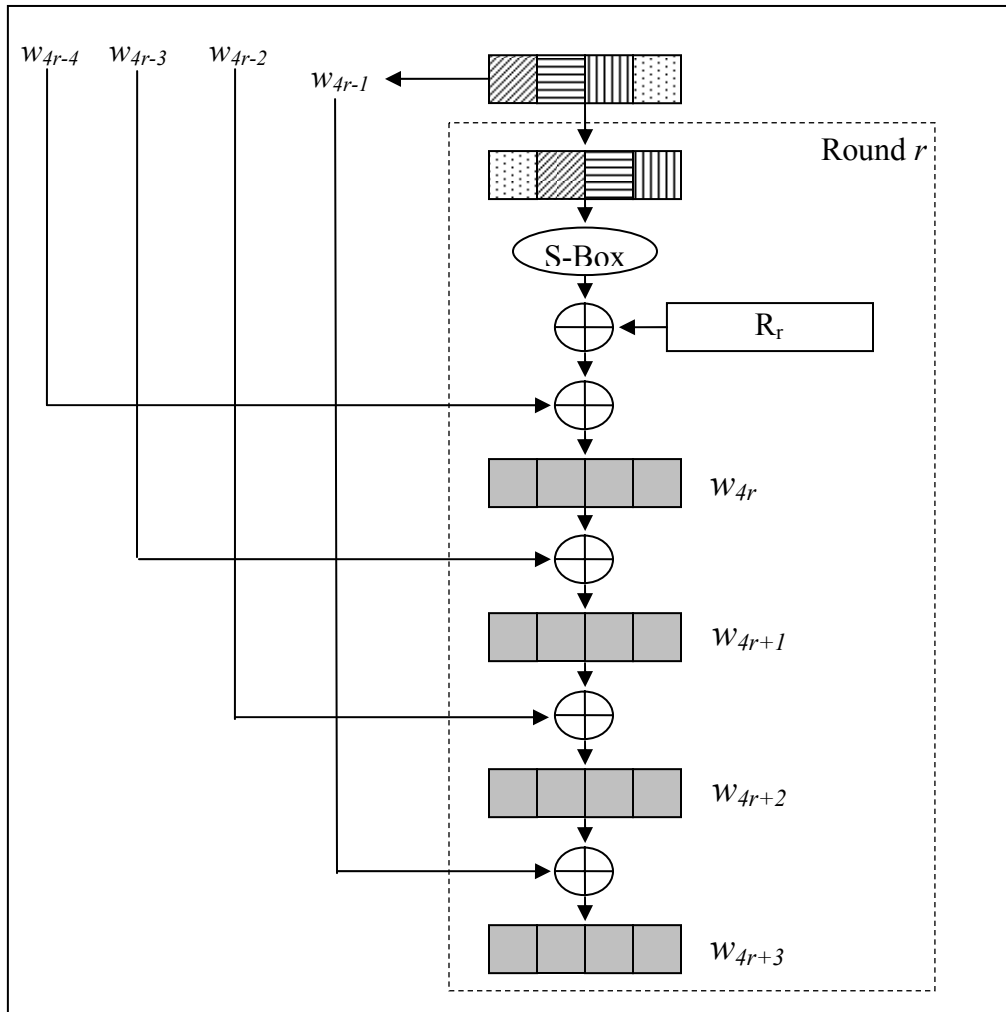


Figure 1 – Key schedule for round  $r$  subkey

### 2.1 Giraud's model

Both Giraud and Chen & Yen assume that one byte of the penultimate (or antepenultimate) round key could be corrupted whilst in storage between rounds, i.e. the entire round key is first generated correctly by the key scheduling process, used in the round key addition, and then corrupted before it is used as an input to the scheduling of the next round key. This model applies only to corruptions whilst keys are statically held in RAM. The fault model does not apply if the byte was corrupted whilst it was calculated, since then the corrupted value would be used in the round and the fault would propagate to later bytes in the same round key. Similarly, if the byte was read in wrongly from RAM during scheduling of the next round key, it would have to be read incorrectly twice in the same way. This is because the last word of a round key is used twice when calculating the next round key. Since most smartcard chips do not have sufficient registers to hold two round keys, the last word of the previous round key is read from RAM twice.

Giraud's model also does not apply when the round keys are scheduled all-at-once prior to the start of encipherment. If the key schedule is calculated this way, the corruption could happen after use of the key byte, satisfying one aspect of the model, but then the fault would not propagate to the succeeding round key as it has already been calculated. For the fault to propagate as required, the corruption would have to occur during the key scheduling process, but since this happens before the key byte gets used, the corrupted byte would then be used, again breaking an assumption of the model.

## 2.2 Our model

To overcome the restrictions of Giraud's fault model, we assume that faults can be introduced into the execution of the key scheduling process. That is, the fault is present in the key byte when it is used, and also propagates to all further iterations of the key schedule, including those which calculate later bytes of the same round key. This means that the fault model applies both to the all-at-once key scheduling process and to the inline round-at-a-time key schedule. The assumption that faults can affect the execution of the key scheduling process seems reasonable, since our experience shows that faults can affect calculations just as well as they can affect bytes held statically in RAM.

A further assumption our fault model introduces is that an entire word (32 bits) of the scheduled key is corrupted, rather than just one byte. This change in assumptions serves two purposes. Firstly, it reduces the number of faults required. Secondly, it allows any fault which affects an entire iteration to be analysed, relaxing the constraint on affecting one specific byte. Our fault model assumes that the attacker can choose which word of a round key is corrupted with a reasonable probability of success, but that they cannot choose the corrupted values which are essentially random.

An example of our fault model is illustrated in Figure 2 below. We induce a fault into the 3<sup>rd</sup> iteration, so  $w_{38}' = w_{38} \text{ XOR } \varepsilon$ . The 4<sup>th</sup> iteration calculates  $w_{39}' = w_{38}' \text{ XOR } w_{35}$ , so  $w_{39}' = w_{39} \text{ XOR } \varepsilon$  (it contains the same error, represented by the same hatching pattern). The first iteration in the final round takes  $w_{39}' \text{ XOR } \varepsilon$  and rotates it by one byte, then passes it through the S-Box and XORs the output with the round constant  $R_{10} \text{ XOR } w_{36}$ . Therefore, the word produced by this step is  $w_{40}' = w_{40} \text{ XOR } \delta$ , i.e. due to the non-linear S-Box lookup it contains a different error. This is shown with a different hatching pattern in the diagram. Now,  $w_{41}' = w_{40}' \text{ XOR } w_{37}$ , so  $w_{41}' = w_{41} \text{ XOR } \delta$ , the same fault as in the previous word. In the 3<sup>rd</sup> iteration of the final round,  $w_{42}' = w_{41}' \text{ XOR } w_{38}'$ , so since  $w_{38}' = w_{38} \text{ XOR } \varepsilon$ ,  $w_{42}' = w_{42} \text{ XOR } \delta \text{ XOR } \varepsilon$ . A new hatching pattern represents this cumulative error. In the final step,  $w_{43}' = w_{42}' \text{ XOR } w_{39}' = (w_{42} \text{ XOR } \delta \text{ XOR } \varepsilon) \text{ XOR } (w_{39} \text{ XOR } \varepsilon) = w_{43} \text{ XOR } \delta$  (so the final word contains the same error as  $w_{41}'$  and  $w_{40}'$ ).

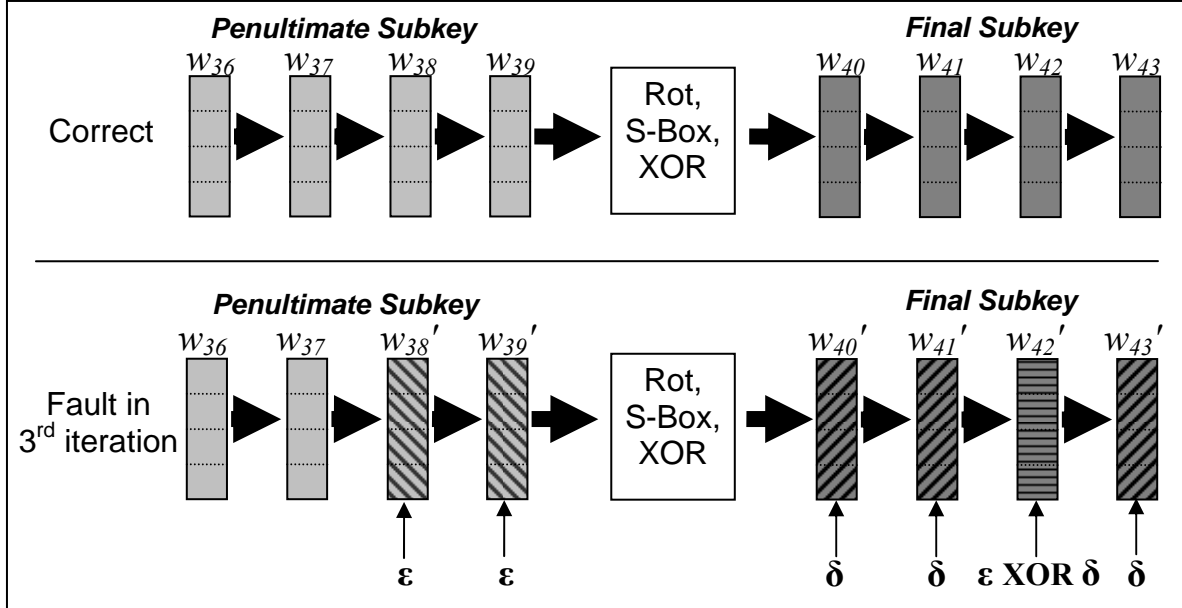


Figure 2 – Illustrated example of the fault model

### 3. Assumptions and notation for the attack

We assume that the device is performing an AES encipherment using a 16-byte key and 16-byte block size. We assume that the device will process the same input twice and will allow the attacker to examine all 16 bytes of the output, so that correct and incorrect outputs may be compared. The inputs need not be chosen, nor even known to the attacker.

We denote the 16 bytes of correct output by  $Z_0$  to  $Z_{15}$ . We denote the XOR between the correct and incorrect outputs by  $z_0$  to  $z_{15}$ . (In other words, the incorrect output is  $Z_0 \wedge z_0$  to  $Z_{15} \wedge z_{15}$ , where  $\wedge$  denotes XOR.) We denote the 16 bytes of the correct round key of the penultimate round by  $P_0$  to  $P_{15}$ , corresponding to key schedule outputs  $w_{35}$  to  $w_{39}$ . We denote corruptions induced in these bytes by  $p_0$  to  $p_{15}$ . We denote the round constant of the final round of key scheduling by  $R_{10}$ . We denote the S-Box lookup by  $S[\cdot]$  and its inverse by  $S^{-1}[\cdot]$ .

### 4. Preparing for the attack: state expressions

The AES state immediately before the Key Add operation of the penultimate round is given by the following 16 expressions, which give the values of each byte of the state in terms of the outputs  $Z_0$  to  $Z_{15}$  and the penultimate round key  $P_0$  to  $P_{15}$ . We give these expressions ordered by the  $Z_i$  to which they relate; the presence of a Shift Row operation in the final round means that this is different from their order in the state. We denote their position in the state by zero-based column and row numbers.

$$\text{Column 0, row 0: } P_0 \wedge S^{-1}[P_0 \wedge R_{10} \wedge S[P_{13}] \wedge Z_0]$$

$$\text{Column 1, row 1: } P_5 \wedge S^{-1}[P_1 \wedge S[P_{14}] \wedge Z_1]$$

$$\text{Column 2, row 2: } P_{10} \wedge S^{-1}[P_2 \wedge S[P_{15}] \wedge Z_2]$$

Column 3, row 3:  $P_{15} \wedge S^{-1}[P_3 \wedge S[P_{12}] \wedge Z_3]$

Column 1, row 0:  $P_4 \wedge S^{-1}[P_0 \wedge P_4 \wedge R_{10} \wedge S[P_{13}] \wedge Z_4]$

Column 2, row 1:  $P_9 \wedge S^{-1}[P_1 \wedge P_5 \wedge S[P_{14}] \wedge Z_5]$

Column 3, row 2:  $P_{14} \wedge S^{-1}[P_2 \wedge P_6 \wedge S[P_{15}] \wedge Z_6]$

Column 0, row 3:  $P_3 \wedge S^{-1}[P_3 \wedge P_7 \wedge S[P_{12}] \wedge Z_7]$

Column 2, row 0:  $P_8 \wedge S^{-1}[P_0 \wedge P_4 \wedge P_8 \wedge R_{10} \wedge S[P_{13}] \wedge Z_8]$

Column 3, row 1:  $P_{13} \wedge S^{-1}[P_1 \wedge P_5 \wedge P_9 \wedge S[P_{14}] \wedge Z_9]$

Column 0, row 2:  $P_2 \wedge S^{-1}[P_2 \wedge P_6 \wedge P_{10} \wedge S[P_{15}] \wedge Z_{10}]$

Column 1, row 3:  $P_7 \wedge S^{-1}[P_3 \wedge P_7 \wedge P_{11} \wedge S[P_{12}] \wedge Z_{11}]$

Column 3, row 0:  $P_{12} \wedge S^{-1}[P_0 \wedge P_4 \wedge P_8 \wedge P_{12} \wedge R_{10} \wedge S[P_{13}] \wedge Z_{12}]$

Column 0, row 1:  $P_1 \wedge S^{-1}[P_1 \wedge P_5 \wedge P_9 \wedge P_{13} \wedge S[P_{14}] \wedge Z_{13}]$

Column 1, row 2:  $P_6 \wedge S^{-1}[P_2 \wedge P_6 \wedge P_{10} \wedge P_{14} \wedge S[P_{15}] \wedge Z_{14}]$

Column 2, row 3:  $P_{11} \wedge S^{-1}[P_3 \wedge P_7 \wedge P_{11} \wedge P_{15} \wedge S[P_{12}] \wedge Z_{15}]$

### 5. Attack phase 1 – corrupt the 4<sup>th</sup> iteration in the penultimate round

Let us suppose we induce a random error in the calculation of the last 32-bit word of P. This means that  $P_0$  to  $P_{11}$  will be correct, but  $P_{12}$  to  $P_{15}$  will take the corrupted values  $P_{12} \wedge p_{12}$  to  $P_{15} \wedge p_{15}$ .

The state will be correct up to the Key Add of the penultimate round. The right-most column of the state will be corrupted by the addition of the corrupt bytes of P. Moreover, the corruption in P will cause all sixteen bytes of the final round key to be calculated incorrectly, so the observed output will be incorrect in every byte.

We have seen that column 0, row 0 of the state before the Key Add of P is given by:

Column 0, row 0:  $P_0 \wedge S^{-1}[P_0 \wedge R_{10} \wedge S[P_{13}] \wedge Z_0]$

In the presence of corruptions  $p_{12}$  to  $p_{15}$ , causing output corrupted by  $z_0$  to  $z_{15}$ , the state is given by:

Column 0, row 0:  $P_0 \wedge S^{-1}[P_0 \wedge R_{10} \wedge S[P_{13} \wedge p_{13}] \wedge Z_0 \wedge z_0]$

But these two expressions must be equal, because the state is correct before the addition of P. Equating the two we have:

$$P_0 \wedge S^{-1}[P_0 \wedge R_{10} \wedge S[P_{13}] \wedge Z_0] = P_0 \wedge S^{-1}[P_0 \wedge R_{10} \wedge S[P_{13} \wedge p_{13}] \wedge Z_0 \wedge z_0]$$

This simplifies to:

$$S[P_{13}]^{\wedge}S[P_{13}^{\wedge}p_{13}]^{\wedge}z_0 = 0$$

We can apply similar reasoning to the other 15 bytes of the state, though not all of them simplify to such an extent. The resulting 16 equations are as follows:

$$S[P_{13}]^{\wedge}S[P_{13}^{\wedge}p_{13}]^{\wedge}z_0 = 0$$

$$S[P_{14}]^{\wedge}S[P_{14}^{\wedge}p_{14}]^{\wedge}z_1 = 0$$

$$S[P_{15}]^{\wedge}S[P_{15}^{\wedge}p_{15}]^{\wedge}z_2 = 0$$

$$S^{-1}[P_3^{\wedge}S[P_{12}]^{\wedge}Z_3]^{\wedge}S^{-1}[P_3^{\wedge}S[P_{12}^{\wedge}p_{12}]^{\wedge}Z_3^{\wedge}z_3]^{\wedge}p_{15} = 0$$

$$S[P_{13}]^{\wedge}S[P_{13}^{\wedge}p_{13}]^{\wedge}z_4 = 0$$

$$S[P_{14}]^{\wedge}S[P_{14}^{\wedge}p_{14}]^{\wedge}z_5 = 0$$

$$S^{-1}[P_2^{\wedge}P_6^{\wedge}S[P_{15}]^{\wedge}Z_6]^{\wedge}S^{-1}[P_2^{\wedge}P_6^{\wedge}S[P_{15}^{\wedge}p_{15}]^{\wedge}Z_6^{\wedge}z_6]^{\wedge}p_{14} = 0$$

$$S[P_{12}]^{\wedge}S[P_{12}^{\wedge}p_{12}]^{\wedge}z_7 = 0$$

$$S[P_{13}]^{\wedge}S[P_{13}^{\wedge}p_{13}]^{\wedge}z_8 = 0$$

$$S^{-1}[P_1^{\wedge}P_5^{\wedge}P_9^{\wedge}S[P_{14}]^{\wedge}Z_9]^{\wedge}S^{-1}[P_1^{\wedge}P_5^{\wedge}P_9^{\wedge}S[P_{14}^{\wedge}p_{14}]^{\wedge}Z_9^{\wedge}z_9]^{\wedge}p_{13} = 0$$

$$S[P_{15}]^{\wedge}S[P_{15}^{\wedge}p_{15}]^{\wedge}z_{10} = 0$$

$$S[P_{12}]^{\wedge}S[P_{12}^{\wedge}p_{12}]^{\wedge}z_{11} = 0$$

$$S^{-1}[P_0^{\wedge}P_4^{\wedge}P_8^{\wedge}P_{12}^{\wedge}R_{10}^{\wedge}S[P_{13}]^{\wedge}Z_{12}]^{\wedge}S^{-1}[P_0^{\wedge}P_4^{\wedge}P_8^{\wedge}P_{12}^{\wedge}R_{10}^{\wedge}S[P_{13}^{\wedge}p_{13}]^{\wedge}Z_{12}^{\wedge}p_{12}^{\wedge}z_{12}]^{\wedge}p_{12} = 0$$

$$S[P_{14}]^{\wedge}S[P_{14}^{\wedge}p_{14}]^{\wedge}p_{13}^{\wedge}z_{13} = 0$$

$$S[P_{15}]^{\wedge}S[P_{15}^{\wedge}p_{15}]^{\wedge}p_{14}^{\wedge}z_{14} = 0$$

$$S[P_{12}]^{\wedge}S[P_{12}^{\wedge}p_{12}]^{\wedge}p_{15}^{\wedge}z_{15} = 0$$

## 6. Attack phase 2 - solving the state equations

We can solve some of these equations in order to obtain bytes of P. For example, we have:

$$S[P_{13}]^{\wedge}S[P_{13}^{\wedge}p_{13}]^{\wedge}z_0 = 0$$

$$S[P_{14}]^{\wedge}S[P_{14}^{\wedge}p_{14}]^{\wedge}z_1 = 0$$

$$S[P_{14}]^{\wedge}S[P_{14}^{\wedge}p_{14}]^{\wedge}p_{13}^{\wedge}z_{13} = 0$$

We can eliminate  $S[P_{14}] \wedge S[P_{14} \wedge p_{14}]$  between the last two of these equations, giving:

$$p_{13} \wedge z_1 \wedge z_{13} = 0$$

We know  $z_1$  and  $z_{13}$ , so we can find  $p_{13}$ , the XOR between the correct and incorrect values of  $P_{13}$ . We then substitute this value of  $p_{13}$  into the first equation and solve by exhaustive search to obtain  $P_{13}$ . It may be possible to derive  $P_{13}$  by mathematical analysis without exhaustive search but we did not investigate this since a search over 8 bits is very cheap and is also straightforward to understand and implement. We obtain two candidate values for  $P_{13}$ , one of which is the correct value and the other the corrupted value.

In order to distinguish the two candidates, we merely have to repeat the calculation for another correct-incorrect pair. Assuming the corruptions are random, we shall obtain one value for  $P_{13}$  that is the same as one found from the first correct-incorrect pair, and one which is different. The value that appears both times is the correct one.

We can apply exactly the same reasoning to obtain  $P_{14}$  and  $P_{15}$ .

We cannot obtain any of the other 13 bytes of  $P$  directly. However, we can obtain information about them equivalent to 3 further bytes, as follows.

The equation for  $Z_6$  is:

$$S^{-1}[P_2 \wedge P_6 \wedge S[P_{15}] \wedge Z_6] \wedge S^{-1}[P_2 \wedge P_6 \wedge S[P_{15} \wedge p_{15}] \wedge Z_6 \wedge z_6] \wedge p_{14} = 0$$

But we know  $P_{15}$ , and we know  $p_{14}$  and  $p_{15}$  for each correct-incorrect pair. Therefore we can solve this equation by exhaustive search to obtain  $P_2 \wedge P_6$ .

Similarly, we have:

$$S^{-1}[P_1 \wedge P_5 \wedge P_9 \wedge S[P_{14}] \wedge Z_9] \wedge S^{-1}[P_1 \wedge P_5 \wedge P_9 \wedge S[P_{14} \wedge p_{14}] \wedge Z_9 \wedge z_9] \wedge p_{13} = 0$$

We can solve this to obtain  $P_1 \wedge P_5 \wedge P_9$ .

Lastly, we have:

$$S^{-1}[P_3 \wedge S[P_{12}] \wedge Z_3] \wedge S^{-1}[P_3 \wedge S[P_{12} \wedge p_{12}] \wedge Z_3 \wedge z_3] \wedge p_{15} = 0$$

$$S[P_{12}] \wedge S[P_{12} \wedge p_{12}] \wedge z_7 = 0$$

From these we can eliminate  $S[P_{12} \wedge p_{12}]$  so as to obtain:

$$S^{-1}[P_3 \wedge S[P_{12}] \wedge Z_3] \wedge S^{-1}[P_3 \wedge S[P_{12}] \wedge Z_3 \wedge z_3 \wedge z_7] \wedge p_{15} = 0$$

We can solve this so as to obtain  $P_3 \wedge S[P_{12}]$ , everything else is known.

Hence we have obtained the equivalent of six bytes of  $P$ .

### 7. Attack phase 3 - corrupt the 4<sup>th</sup> iteration in the penultimate round

We now have the equivalent of 6 bytes of P. This leaves 10 bytes unknown, which is far beyond exhaustive search. However, we can obtain the remaining bytes by collecting further correct-incorrect pairs, in which we have corrupted earlier words of P.

Let us suppose that we can induce a random corruption in  $P_8$  to  $P_{11}$ . We denote the corruptions by  $p_8$  to  $p_{11}$ . These corruptions will appear directly in  $P_{12}$  to  $P_{15}$ , which will be corrupted to  $P_{12} \wedge p_8$  to  $P_{15} \wedge p_{11}$ , and will affect the final round key as well. We rewrite our state expressions and equations to take account of the new values of  $P_8$  to  $P_{15}$ .

We can deduce that:

$$S^{-1}[P_2 \wedge S[P_{15}] \wedge Z_2] \wedge S^{-1}[P_2 \wedge S[P_{15} \wedge p_{11}] \wedge Z_2 \wedge z_2] \wedge p_{10} = 0$$

We have already determined  $P_{15}$ . Moreover, we can calculate  $p_{10}$  and  $p_{11}$  for each correct-incorrect pair by noting that, for example:

$$S[P_{14}] \wedge S[P_{14} \wedge p_{10}] \wedge z_1 = 0$$

Therefore we can solve for  $P_2$ . We already know  $P_2 \wedge P_6$ , so now we can calculate  $P_6$ .

Further, we can deduce that:

$$S^{-1}[P_1 \wedge P_5 \wedge S[P_{14}] \wedge Z_5] \wedge S^{-1}[P_1 \wedge P_5 \wedge S[P_{14} \wedge p_{10}] \wedge Z_5 \wedge z_5] \wedge p_9 = 0$$

We can solve this for  $P_1 \wedge P_5$ . We already know  $P_1 \wedge P_5 \wedge P_9$ , so we can calculate  $P_9$ .

Further, we can deduce that:

$$S^{-1}[P_3 \wedge P_7 \wedge P_{11} \wedge P_{15} \wedge S[P_{12}] \wedge Z_{15}] \wedge S^{-1}[P_3 \wedge P_7 \wedge P_{11} \wedge P_{15} \wedge S[P_{12} \wedge p_8] \wedge Z_{15} \wedge z_{15}] \wedge p_{11} = 0$$

$$S[P_{12}] \wedge S[P_{12} \wedge p_8] \wedge z_7 = 0$$

From these we can eliminate  $S[P_{12} \wedge p_8]$ :

$$S^{-1}[P_3 \wedge P_7 \wedge P_{11} \wedge P_{15} \wedge S[P_{12}] \wedge Z_{15}] \wedge S^{-1}[P_3 \wedge P_7 \wedge P_{11} \wedge P_{15} \wedge S[P_{12}] \wedge Z_{15} \wedge z_7 \wedge z_{15}] \wedge p_{11} = 0$$

Hence we can calculate  $P_3 \wedge P_7 \wedge P_{11} \wedge S[P_{12}]$ . We already know  $P_3 \wedge S[P_{12}]$ , so we can calculate  $P_7 \wedge P_{11}$ .

Lastly, we can deduce that:

$$S[P_{12}] \wedge S[P_{12} \wedge p_8] \wedge z_7 = 0$$

$$S^{-1}[P_0 \wedge P_4 \wedge P_8 \wedge R_{10} \wedge S[P_{13}] \wedge Z_8] \wedge S^{-1}[P_0 \wedge P_4 \wedge P_8 \wedge R_{10} \wedge S[P_{13} \wedge p_9] \wedge Z_8 \wedge p_8 \wedge z_8] \wedge p_8 = 0$$

$$S^{-1}[P_0 \wedge P_4 \wedge P_8 \wedge P_{12} \wedge R_{10} \wedge S[P_{13}] \wedge Z_{12}] \wedge S^{-1}[P_0 \wedge P_4 \wedge P_8 \wedge P_{12} \wedge R_{10} \wedge S[P_{13} \wedge p_9] \wedge Z_{12} \wedge z_{12}] \wedge p_8 = 0$$

These three equations contain three unknowns,  $P_0 \wedge P_4 \wedge P_8$ ,  $P_{12}$  and  $p_8$ . We can solve them for each correct-incorrect pair as follows:

- Guess the value of  $P_{12}$ .
- Substitute the guess into the first equation and solve this to obtain  $p_8$  (no search required).
- Substitute  $p_8$  into the second equation and solve this to obtain  $P_0 \wedge P_4 \wedge P_8$  (an 8-bit search).
- Substitute  $P_0 \wedge P_4 \wedge P_8$ ,  $P_{12}$  and  $p_8$  into the third equation and determine whether the equation is satisfied (no search required). If it is not, then the guessed value of  $P_{12}$  was incorrect; try another guess.

This allows us to calculate  $P_0 \wedge P_4 \wedge P_8$  and  $P_{12}$  and amounts to an exhaustive search over  $2^{16}$  possibilities: an 8-bit search for each of the possible 8-bit values for  $P_{12}$ . We already know  $P_3 \wedge S[P_{12}]$ , so we can calculate  $P_3$ .

By analysing faults in  $P_8$  to  $P_{11}$ , we have obtained the equivalent of 5 bytes of the key, in addition to the 6 bytes found by analysing faults in  $P_{12}$  to  $P_{15}$ . This gives us a total of 11 bytes, and the remaining 5 bytes can be found by exhaustive search. Alternatively, we may find the remaining 5 bytes by two more DFA steps, as follows.

### 8. Attack phase 3 – corrupting the 1<sup>st</sup> and 2<sup>nd</sup> iterations

We induce faults on bytes  $P_4$  to  $P_7$ . We now rewrite the state expressions and equations to reflect the new values of  $P_4$  to  $P_{15}$ . We can deduce that:

$$S^{-1}[P_1 \wedge S[P_{14}] \wedge Z_1] \wedge S^{-1}[P_1 \wedge S[P_{14} \wedge p_6] \wedge Z_1 \wedge z_1] \wedge p_5 = 0$$

We know  $P_{14}$ , and we can calculate  $p_5$  and  $p_6$  for each correct-incorrect pair by similar reasoning to that given earlier. Hence we can solve for  $P_1$ . We already know  $P_1 \wedge P_5$ , so we can calculate  $P_5$ .

We deduce that:

$$S^{-1}[P_2 \wedge P_6 \wedge P_{10} \wedge P_{14} \wedge S[P_{15}] \wedge Z_{14}] \wedge S^{-1}[P_2 \wedge P_6 \wedge P_{10} \wedge P_{14} \wedge S[P_{15} \wedge p_7] \wedge Z_{14} \wedge p_6 \wedge z_{14}] \wedge p_6 = 0$$

We can solve this for  $P_{10}$ .

We deduce that:

$$S^{-1}[P_0 \wedge P_4 \wedge R_{10} \wedge S[P_{13}] \wedge Z_4] \wedge S^{-1}[P_0 \wedge P_4 \wedge R_{10} \wedge S[P_{13} \wedge p_5] \wedge Z_4 \wedge p_4 \wedge z_4] \wedge p_4 = 0$$

We can solve this for  $P_0 \wedge P_4$ . We already know  $P_0 \wedge P_4 \wedge P_8$ , so we can calculate  $P_8$ .

The faults on  $P_4$  to  $P_7$  have therefore given us 3 more bytes of  $P$ , leaving two bytes to be found by exhaustive search or by inducing faults on  $P_0$  to  $P_3$ . Faults on  $P_0$  to  $P_3$  give us:

$$S^{-1}[P_3 \wedge P_7 \wedge S[P_{12} \wedge Z_7]] \wedge S^{-1}[P_3 \wedge P_7 \wedge S[P_{12} \wedge p_0] \wedge Z_7 \wedge z_7] \wedge p_3 = 0$$

We can solve this for  $P_7$ . We already know  $P_7 \wedge P_{11}$ , so we can calculate  $P_{11}$ . Faults on  $P_0$  to  $P_3$  also give us:

$$S^{-1}[P_0 \wedge R_{10} \wedge S[P_{13} \wedge Z_0]] \wedge S^{-1}[P_0 \wedge R_{10} \wedge S[P_{13} \wedge p_1] \wedge Z_0 \wedge p_0 \wedge z_0] \wedge p_0 = 0$$

We can solve this for  $P_0$ . We already know  $P_0 \wedge P_4$ , so we can calculate  $P_4$ .

We now know all 16 bytes of  $P$ . From these it is straightforward to calculate the 16 bytes of the AES key, no further exhaustive search being required.

## 9. Practical results

We implemented a simple software AES on a modern security chip. Our software contained no defences against fault attacks, and used the all-at-once method of scheduling the key. Using a pulsed laser, we were able to cause faults consistent with our fault model. By varying the timing of the laser we were able to fault every iteration of the penultimate key scheduling step, as required for our analysis. Although we did not always fault the iteration we targeted, we could generate a number of faults and then analyse the faulty ciphertexts in pairs. If the both the faults were in the iteration being analysed, the key bytes found by the analysis would agree, otherwise they would not agree in any of the values. By analysing our faults in pairs in this way, we could perform the analysis without knowing which specific iteration had been faulted.

## 10. Discussion and countermeasures

We have presented an attack which allows for some trade-offs between the number of faults required and the amount of exhaustive search needed. For example, it is possible to cause faults in only the last two iterations and then search for the remaining five bytes of the key. Alternatively, one can fault only the last three iterations and search for two bytes of the key. Faulting all 4 iterations allows calculation of the entire round key without further search.

Assuming the corruptions are random, the analysis rarely requires more than eight faults (two per iteration), and could be done with as few as four faults with more calculation. The trouble with using one fault per iteration is that we cannot distinguish between the true value of the key and the corrupted value. This could be solved by brute force. For example, in the faulting of the last iteration (the first phase of the attack) we obtain the equivalent of six bytes of information but we have two possibilities for each byte with only one fault. We can guess which possibility is correct for each byte and propagate these values backwards into the analysis of the fault for the previous iteration. We can continue back, continuing similarly until we have analysed faults in all iterations, we can then check whether the key we have is consistent with a known correct plaintext / ciphertext pair. If it is not, we guess again which values were correct in the last phase of the attack, and try those until we run out of possibilities and have to resort to changing our guesses in previous attack phases and working these through. This is equivalent to losing one bit of information per byte obtained, and so amounts to an extra 16 bits of

search. It could be useful in situations where the amount of faults that can be obtained is limited.

Our attack relies upon being able to corrupt specific iterations of the AES key scheduling process, so clearly it cannot be used in scenarios where the round keys are pre-scheduled and stored in the device prior to any encryptions being done. This is one potential countermeasure against the attack. The normal method to calculate the round keys on a per-encryption basis is to generate the key schedule prior to encipherment, and our attack applies in this case. Even if a decipherment check is used as a DFA countermeasure, the decipherment key schedule is normally calculated from the encipherment key schedule or is identical, and in this case decipherment will be the inverse of encipherment so the attack is not detected. The attack also applies to the case where the key schedule is calculated inline with the cipher, so that a round key is generated just prior to the round it is used in and only a single round key is held at a time. The normal way to generate the decipherment keys in this case would be to save the final round key and use it to work backwards through the key schedule. If decipherment is used as a DFA check in this case, our attack would be detected, since the fault in the final round key will propagate back to all of the round keys. This is therefore a countermeasure against our attack. Chen & Yen describe other possible countermeasures. The first is only calculating the key schedule when the key is changed and saving the schedule for use in all subsequent cipher operations. This defeats our attack since the attacker can only cause one fault per key used, too few for our analysis. The other countermeasure they suggest is parity checking the keys as they are generated: this involves treating the round key bytes as a 4x4 matrix which has a parity bit for each row and each column, [2] describes the method to predict the parities of the current round from the previous round parities. This should detect possible fault attempts and prevent faulty ciphertexts from being output, which again defeats our attack.

It appears likely that our attack can be extended to the other key and block sizes available in the AES and Rijndael. This is the subject of the future work.

## 11. Conclusion

This paper describes a new form of DFA attack against AES. Like some previous attacks, it attempts to corrupt round key values in order to derive information about the true values of round key bytes. However, the attack we present requires fewer faults than most previously considered AES fault attacks. As discussed in the preceding section, this attack is only applicable to certain implementation scenarios, although we believe that such scenarios exist in real-world systems. We have practically demonstrated the attack against a software AES implementation on a modern smartcard chip. Clearly, cryptosystem designers and implementers need to be aware of this attack when they implement in AES in devices where fault induction is a realistic threat.

## References

- [1] C. Giraud, "DFA on AES", <http://citeseer.ist.psu.edu/558158.html>
- [2] C.-N. Chen, S.-M. Yen, "Differential Fault Analysis on AES Key Schedule and Some Countermeasures", *Australasian Conference on Information Security and Privacy*

- 2003 (*ACISP 2003*), *LNCS 2727*, pp. 118–129. Springer-Verlag  
[http://www.ncu.edu.tw/~ncu7020/PHD\\_Report/92/EECS/ccn/paper.pdf](http://www.ncu.edu.tw/~ncu7020/PHD_Report/92/EECS/ccn/paper.pdf)
- [3] P. Dusart, G. Letourneux, O. Vivolo, “Differential Fault Analysis on A.E.S.”, *Cryptology ePrint Archive: Report 2003/010*. <http://www.iacr.org>
- [4] Boneh, DeMillo & Lipton, “On the Importance of Checking Cryptographic Protocols for Faults”, *Eurocrypt '97*, Springer Verlag.
- [5] Biham & Shamir, *A New Cryptanalytic Attack on DES*, pre-print, 1996.