Fault Analysis of DPA-Resistant Algorithms

Frederic Amiel[1], Christophe Clavier[1] and Michael Tunstall[2]

[1] Gemalto, Security Labs,
Avenue des Jujubiers, La Ciotat, F-13705, France.
{frederic.amiel, christophe.clavier}@gemalto.com
[2] Smart Card Centre, Information Security Group,
Royal Holloway, University of London,
Egham, Surrey TW20 0EX, UK.
m.j.tunstall@rhul.ac.uk

Abstract. In this paper several attacks are presented that allow information to be derived on faults injected at the beginning of cryptographic algorithm implementations that use Boolean masking to defend against Differential Power Analysis (DPA). These attacks target the initialisation functions that are used to enable the algorithm to be protected, allowing a fault attack even in the presence of round redundancy. A description of the experiments leading to the development of these attacks is also given.

1 Introduction

The use of collisions to find and exploit a fault at the beginning of an algorithm has appeared in several papers. In [7] a method of exploiting faults in the early rounds of a DES implementation is described. This detailed a complex attack where faults were injected in the early rounds of DES, and the fault injected was then derived by finding a message that would naturally give the same ciphertext. This information was then used to derive information on the key.

A trivial case of this type of attack is given in [3] where a known bit of the first XOR in AES is assumed to be forced to zero. If the ciphertext changes then this bit would have been a 1; if the ciphertext remains the same then the bit is a 0. This would break an AES implementation with a mere 128 executions with successful fault injections. However, modifying bits in such a manner requires too much precision to be practical. We refer to this type of attack as Collision Fault Analysis (CFA).

Some variations of these types of attack will be presented that can be implemented against AES on embedded devices. A simple byte-wise implementation of the attack presented in [3] will be described. Several other, more complex, attacks that take advantage of DPA countermeasures to derive information on the key are also detailed. Implementations of some of these attacks, conducted under controlled conditions, are described. A similar attack will also be described, using the DES as an example, where the initialisation of randomised S-boxes are faulted. It will be demonstrated that the simplest version of this attack is by

combining the modification of S-box values with differential fault analysis [2]. The experiments implementing these attacks using glitches on the power supply or clock are also given.

The paper is organised as follows. Section 2 discusses an attempt at implementing a bytewise version of the attack described in [3], and the changes to the attack that needed to be made. Section 3 describes the most popular method for protecting algorithms against DPA attacks. Section 4 describes a possible attack against the first XOR in the AES. Section 5 gives an attack against the key masking process used to initialise the key. Section 6 describes the first attack found using this method. Section 7 details how the countermeasure proposed for the attack described in Section 6 can be circumvented, and proposes a different attack. More complete countermeasures are given in Section 8, followed by the conclusion.

2 Another Trivial Case

Another trivial CFA based attack is a bytewise implementation of the attack given in [3]. If a fault is injected so that a byte of the output becomes zero during the first AddRoundKey function a similar attack can be implemented. All the possible combinations of message bits corresponding to the modified byte can be tested and the algorithm executed again for each value. This process is stopped once a collision is found with the faulty ciphertext. This will give a message with an intermediate state where the fault was injected that is naturally equal to 0. This means that the message byte found is equal to 0 after being XORed with the corresponding key byte. This message byte will therefore be equal to the corresponding key byte.

This requires 16 faulty ciphertexts to be generated, and a search of 2^8 with the targeted device to find each key byte giving a total search time of 2^{12}.

This attack was attempted with several different microprocessors with different methods of fault injection. Varying from glitches on the Vcc to laser light injection. No successful implementation of this attack was achieved.

In [11] faults are demonstrated that enable a **for** loop to be terminated before it has finished all of its iterations. If the memory where the result of the XOR between the message and the key is stored has not been used, it has a high chance of being 00 or `FF` depending on the logical representation of the physical state. If this attack is applied to the key XOR an attack can be implemented by changing one byte of the output to zero and generating the corresponding ciphertext. Then two bytes etc. as in Table 1, as originally proposed in [2].

This can be achieved with 15 successful faults. The first byte of the key can then be found by searching through the 2^9 (i.e. 2^8 values for `XX` and 2 possible values for the rest of the bits depending on the logical representation of the physical state) possible key values that could produce C_{15}. Once the first key byte is known the second key byte can be found with a further 2^8 AES executions using C_{14}. This can be continued with 2^8 AES executions for each subsequent byte, for a total of 2^{13} AES executions to derive the entire key.

Table 1. The Biham-Shamir Attack

Input	AES Key																Output
$M \rightarrow$	$K_0 =$	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	$\rightarrow C_0$
$M \rightarrow$	$K_1 =$	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	00	$\rightarrow C_1$
$M \rightarrow$	$K_2 =$	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	00	00	$\rightarrow C_2$
$M \rightarrow$	$K_3 =$	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	00	00	00	$\rightarrow C_3$
$\vdots$	$\vdots$																	$\vdots$
$M \rightarrow$	$K_{14} =$	XX	XX	00	00	00	00	00	00	00	00	00	00	00	00	00	00	$\rightarrow C_{14}$
$M \rightarrow$	$K_{15} =$	XX	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	$\rightarrow C_{15}$

The basic attack (memory content set to '0') was been implemented on an 8-bit smart card microprocessor using a glitch as fault injector by scanning the entire loop that copies the AES key from Non-Volatile Memory (NVM) to a temporary working RAM buffer at the beginning of AES execution (just before the AddRoundKey function). This produced 127 different faulty ciphertexts (whereas 15 different ones were expected), giving as a result the 22 possible keys listed below in base 16 by searching though the results in a bytewise fashion:

```
00000000000000000000000000000000
FE000000000000000000000000000000
FEDC0000000000000000000000000000
FEDCBA0000000000000000000000000000
FEDCBA9800000000000000000000000000
FEDCBA9876000000000000000000000000
FEDCBA9876540000000000000000000000
FEDCBA9876543200000000000000000000
FEDCBA9876543210000000000000000000
FEDCBA9876543210010000000000000000
FEDCBA9876543210012300000000000000
FEDCBA9876543210012345000000000000
FEDCBA9876543210012345670000000000
FEDCBA9876543210012345678900000000
FEDCBA98765432100123456789ABCDEF
FEDCBA98765432100123456789ABCD00
FEDCBA98765432100123456789ABCDCD
FEDCBA98765432100123456789AB0000
FEDCBA98765432100123456789AB00EF
FEDCBA98765432100123456789AB00AB
FEDCBA98765432100123456789ABEF00
FEDCBA98765432100123456789AB5300
```

Due to desynchronisation effects and the exhaustive scanning of the copy loop, some unexpected faulty ciphertexts are produced. In the possible keys it can be seen that there are several possible values for the last two bytes, which

makes the attack slightly more complicated than originally supposed but it still remains practical. The correct key during these experiments was:

FEDCBA98765432100123456789ABCDEF

3 Secure Algorithm Implementations

Implementations are made secure against DPA [9] and related attacks by masking the data being manipulated with a random value. The data is then manipulated in such a way that the value present in memory is always masked with the same random. An example of this sort of implementation can be found in [1], based on ideas proposed in [4].

The size of the random is generally limited as S-boxes need to be randomised before the execution of the random so that the input and output values of the S-box leak no information. This is done using an algorithm such as Algorithm 1, where the notation $(\cdot)_x$ denotes values of base x i.e. $(s_0, s_1, s_2, \ldots, s_n)_x$ containing S broken into words of size x.

Algorithm 1: Randomising S-Box Values

Input: $S = (s_0, s_1, s_2, \ldots, s_n)_x$ containing the S-box, $\mathbf{R}$ a random $\in [0, n]$, and r a random $\in [0, x)$.

Output: $RS = (rs_0, rs_1, rs_2, \ldots, rs_n)_x$ containing the randomised S-box.

for $i \leftarrow 0$ **to** n **do**
  $rs_{(i \oplus \mathbf{R})} \leftarrow s_i \oplus r$
end

return RS

As shown, the random used for masking the input data can be no larger than n, and the random used for the output value can be no larger that x. In the case of AES both $\mathbf{R}$ and r will be one byte, which means that the random mask used during the calculation is likely to be one byte. This is more problematic for DES as the input and output have a different size, and the bitwise permutations add complexity, but the principal remains the same.

4 Attacking the First XOR

At the beginning of AES the algorithm conducts an XOR between the message and the key before the first ByteSub function. This will happen as shown in Algorithm 2.

As in Section 2 faults are used that enable a **for** loop to be ended before it would normally do so. If Algorithm 2 is attacked in this way so that the loop only runs to 14, rather than 16, two bytes will not be written to KM i.e. two

Algorithm 2: The First XOR

Input: $M = (m_1, m_2, m_3, \ldots, m_{16})_{256}$ containing the message,
$K = (k_1, k_2, k_3, \ldots, k_{16})_{256}$ containing the key masked with $\mathbf{R}$ a random byte.

Output: $KM = (km_1, km_2, km_3, \ldots, km_{16})_{256}$ also masked with $\mathbf{R}$.

for $i \leftarrow 1$ **to** 16 **do**
  $km_i \leftarrow m_i \oplus k_i$
end

return KM

bytes will be untouched. This means that physically these bytes will be set to 0, but the algorithm will take the value as $\mathbf{R}$ due to the masking.

By searching through the 2^{16} possible values for m_{15} and m_{16}, it will be possible to find a collision where $k_{15} \oplus m_{15} \oplus \mathbf{R} = 0$ and $k_{16} \oplus m_{16} \oplus \mathbf{R} = 0$ i.e. $m_{15} \oplus \mathbf{R} = k_{15}$ and $m_{16} \oplus \mathbf{R} = k_{16}$, and therefore $m_{15} \oplus m_{16} = k_{15} \oplus k_{16}$. It is not important what values km_{15} and km_{16} become, but they do need to be the same value. If the value of the memory becomes `FF`, for example, it can still be assumed to be 00 and the error can be taken up in the value of $\mathbf{R}$.

This in itself only reduces the keyspace from 2^{128} to 2^{120}. The attack can be continued by repeating the attack with three bytes of the key being left uninitialised by Algorithm 2. The same method can then be used to derive $k_{14} \oplus k_{15}$ with the same amount of work as $k_{15} \oplus k_{16}$.

For each fault injected the attacker needs to search through 2^{16} different values to determine the message values that enable information on the key to be derived. This needs to be done with the smart card under attack so the attack process will be lengthy.

In DPA-resistant algorithms it is usual to do as much as possible in a random order, as this is an additional countermeasure to that described in Section 3. The loop given in Algorithm 2 would therefore take one of the 120, i.e. $\binom{16}{2}$, different orders possible to XOR the message with the key. This is so that the data being manipulated from one execution to another will occur at different points in time which is primarily a DPA countermeasure.

If a random order is implemented and the last two bytes are not assigned these will be two random bytes in the buffer. To find a collision an attacker will have to search through the 2^{23} possible messages. The random involved will be different each time so two pairs, e.g.

$$k_i \oplus m_i \oplus \mathbf{R},\ k_j \oplus m_j \oplus \mathbf{R}, \text{ and}$$
$$k_j \oplus m'_j \oplus \mathbf{R}',\ k_k \oplus m_k \oplus \mathbf{R}'$$

These pairs cannot be directly related to each other. However, if they have one message byte in common it is possible to change the random mask so that they become the same by XORing the two values together i.e.:

$$k_j \oplus m_j \oplus \mathbf{R} \oplus k_j \oplus m'_j \oplus \mathbf{R}' = m_j \oplus \mathbf{R} \oplus m'_j \oplus \mathbf{R}'$$

As m_j and m'_j are known values these values can be removed from this value with an XOR, leaving $\mathbf{R} \oplus \mathbf{R}'$. This value can be applied to $k_k \oplus m_k \oplus \mathbf{R}'$ so that it becomes $k_k \oplus m'_k \oplus \mathbf{R}$. This then gives:

$$m_i \oplus \mathbf{R} = k_i$$
$$m_j \oplus \mathbf{R} = k_j$$
$$m_k \oplus \mathbf{R} = k_k$$

This process can then be repeated until information is derived on every byte of the key. This then leaves an exhaustive search of 2^8 to find the value of $\mathbf{R}$ and therefore the key.

In order to be able to find collisions the attacker needs to generate a dictionary of 2^{23} entries. These values depend on the key so they need to be generated with the device under attack, which is prohibitively large for devices such as smart cards that use relatively slow communication protocols. To form an idea of the amount of time required to create a dictionary a smart card with a DPA-resistant AES was timed. It took the smart card approximately 149 milliseconds to create one dictionary entry. The whole dictionary will therefore require around 14.5 days to create.

This is an advantage over the version that does not use a random order, as the dictionary can be generated once and used numerous times. As the fault injected will not always do what is expected this may help speed up the overall attack process. In the first version of this attack each search of 2^{16} will require around 3 hours with the test card used.

The implementation conditions of the attack are the same as those described in Section 2 with two exceptions. The implementation used was DPA resistant and the timing of the glitch injection was fixed as the random order provided the temporal variation. The key was known *a priori* so the dictionary was generated with a computer rather than with the smart card.

By conducting faults using glitches on the Vcc for less than one hour, 118 faulty ciphertexts were obtained. Amongst these, 72 unique collisions were extracted. The information in these 72 collisions was compiled to find 31 unique keys i.e. no linear relationship between any of the 31 keys. This process took approximately 10 minutes on a standard PC.

Only one 16-byte key candidate was expected, but 31 were produced showing that some fault injections have produced some collisions not related to the key values. However, the number of keys produced can easily be tested to determine the correct key, requiring $31 \times 2^8 = 2^{12}$ AES executions.

As mentioned above, generating 2^{23} ciphertexts with a smart card takes a prohibitively long time to generate. This can be reduced by only generating a certain amount of the dictionary and conducting more fault attacks to generate the data required to derive the key.

For example, if a dictionary of size 2^{19} was generated, which would take around 21 hours with the test card, faulty ciphertexts could be generated until a collision was found. It would be expected that 1 in 2^4 faulty ciphertexts would be present in the dictionary. An attacker would therefore need 16 times as much

data compared to the attack described above, but as only a few faulty ciphertexts are required to realise the attack this is an efficient option.

5 Attacking the Key Masking

Before the key can be used by the algorithm in the fashion described in Section 3 the random value, **R**, that is applied to the S-boxes needs to be applied to the key. The key values are usually stored XORed with a random value of the same size as the key. This is because this random is a static value for one card as it is stored in the EEPROM and is diversified from one card to another. This mask needs to be removed, and replaced with **R** without the key being manipulated. This process is shown in Algorithm 3. Again, this happens in a random order rather than as shown.

Algorithm 3: Masking the Key

Input: $KR = (kr_1, kr_2, kr_3, \ldots, k_{16})_{256}$ masked with a random $R = (r_1, r_2, r_3, \ldots, r_{16})_{256}$, **R** a random byte.
Output: $K = (k_1, k_2, k_3, \ldots, k_{16})_{256}$ masked with **R**.

for $i \leftarrow 1$ **to** 16 **do**
  $k_i \leftarrow kr_i \oplus \mathbf{R}$
  $k_i \leftarrow k_i \oplus r_i$
end

return K

A similar attack can be envisaged against this process to that shown in Section 4. If only one byte is initialised by this loop the the resulting memory will be predominately set to zero. Again, the algorithm will take this value as being **R**, which gives 2^{20} possible ciphertexts that need to be generated before the attack is conducted. As this attack changes key bytes this dictionary can be generated by a PC as the values are not dependent on the rest of the key.

This attack can be repeated until enough information is derived about the key to enable an exhaustive search to take place. The expected number of faulty ciphertexts needed to derive each byte in this way can be calculated by using the coupon collectors test given in [8]. In the case of AES this would require 50 ciphertexts to derive the whole key.

The implementation conditions of the attack are the same as that described in Section 2. The same software implementation was run on the same smart card.

The precomputation of the dictionary was generated in a matter of minutes on a standard PC. Unlike the attack described in Section 2 the dictionary does not dependent on the value of the secret key used, so the dictionary generated would be valid for any secret key value.

After attacking the implementation for approximately one hour around 60 collisions were generated from faulty ciphertexts. After acquiring these ciphertexts the *a posteriori* processing was trivial as no incorrect hypotheses were produced by the collisions found.

6 Modifying Known S-Box Values

If the S-box values are created as shown in Algorithm 1, the order in which the S-box is constructed is therefore known (as i is incremented from 0 to 63). A fault attack can then be constructed around the modification of known S-box values.

The first S-box value of the first S-box is modified by a fault and the algorithm executed with a message for which the ciphertext is known. If the ciphertext is not equal to the known ciphertext then this S-box value was used by the algorithm; if it stays the same the S-box value was not used anywhere in the algorithm. All 64 values of the first S-box can each be changed in this manner and the algorithm executed. After which, all the S-box entries used from the first S-box will be known for a given message.

The expected number of S-box entries used per DES execution can be calculated using the solution to the classical occupancy problem, as described in [10], giving a value of 14.3. Therefore, if the attack is repeated for each S-box a list of around 14 different values will be given for the number of entries used in each S-box.

If these values are taken as possible hypotheses for the S-box entries used in the first round, the index values of the S-box entries can be turned into hypotheses on the first subkey. To do this, the index values simply need to be XORed with the relevant message bits. This will produce slightly under 2^{31} hypotheses for the first subkey leading to a total exhaustive search of 2^{39} to find the entire DES key.

In order to reduce the size of the exhaustive search the attack can be repeated with a different message. The intersection of the two keyspaces will contain the first subkey. This provides $14.255 \times (14.255/64) = 3.18$ different hypotheses per S-box, which gives 2^{13} hypotheses for the first subkey, leading to a total exhaustive search of 2^{21} keys.

In practice, embedded implementations of DES are unlikely to have the 512 S-box values necessary for DES written separately in memory. These are generally compressed to optimise the amount of memory required by the DES implementation.

One way of achieving this is to store the data on 256 bytes where the odd numbered S-boxes are stored in the high nibbles and the even numbered S-boxes are stored in the low nibbles. This corresponds to the attack implementation detailed below and all further discussion will assume this is the case. There are several other ways in which the S-box data could be compressed, but is not considered to be something an attacker needs to know before conducting an

attack, as all the possible combinations can be attempted until the correct one is found.

The number of key hypotheses generated by implementing this attack against a DES using compressed S-boxes is shown in Table 2 for different numbers of messages used. As can be seen, this is more efficient than modifying 1 S-box value as information on 2 boxes can be gained at once i.e. less faults are required to derive the key.

Table 2. The hypotheses generated by attacking a compressed S-box.

Messages	Hypotheses per S-box pair	Hypotheses for the first round key	Total Keyspace
1	25.3	2^{37}	2^{45}
2	10.0	2^{27}	2^{35}
3	3.97	2^{16}	2^{24}
4	1.57	2^{5}	2^{13}

In attempting to implement the attacks described in [2], it was observed that when the duty cycle[3] of the clock given to the smart card was too small an incorrect ciphertext was produced. This was on a different chip to that used in the previous attacks on the initial functions of the actual algorithm. Further study revealed that if the duty cycle was below 15% data written to certain areas of the chip's memory would then be written incorrectly.

This attack could therefore be implemented against a smart card using this effect. As mentioned above, the S-boxes were written in a compressed format to save memory, so this needed to be taken into account. The attack was conducted with three different messages, followed by a small exhaustive search to find the key. The entire attack took 45 minutes using tools created specifically for this purpose.

This attack can be further optimised by analysing the faulty ciphertexts generated by the modified S-boxes. It should be apparent from the ciphertext if a faulty S-box values has been used in the fifteenth or sixteenth round. As the aim is to try and derive hypotheses on the first subkey these ciphertexts provide no information. Ciphertexts where the faulty S-box is used in the last round can be considered to be equivalent to the S-box value not being used. If the faulty value is used in the fifteenth round no information is provided as the detection of this event is subject to false positives (as described in Section 7) and a different message needs to be used to provide information on this S-box value.

The countermeasure for this specific attack is to randomise the order in which the S-boxes are randomised. This applies to the order in which the S-boxes are

[3] The duty cycle is the amount of time that a voltage is applied to the clock pin compared to the time no voltage is applied e.g. a standard clock will have a duty cycle of 50%.

treated, and to the order in which the S-box elements are masked. The data masking can be done as shown in Algorithm 4, which adds no extra time to the algorithm implementation. The counter i is XORed with a random before being used so the order in which the S-box elements are treated is unknown.

Algorithm 4: Randomising DES S-Box Values

Input: $S = (s_0, s_1, \ldots, s_{63})_{16}$ containing the S-box, $\mathbf{R}$ a random $\in [0, 63]$, and r a random $\in [0, 15]$.

Output: $RS = (rs_0, rs_1, \ldots, rs_{63})_{16}$ containing the randomised S-box.

for $i \leftarrow 0$ **to** 63 **do**
    $rs_i \leftarrow s_{(i \oplus \mathbf{R})} \oplus r$
end

return RS

If just the order in which the S-boxes are treated is randomised an attack could be envisaged based on searching for S-box elements that never change the ciphertext when modified. This is because the information about which index value has been changed will be present. If the same S-box element is repeatedly changed, but after numerous executions with the same message the ciphertext never changes, it can reasonably be assumed that this index value does not represent a key hypothesis for any part of the first subkey. The expected number of executions required to be sure of this information is 22 (given by the coupon collectors test as defined in [8]). The randomisation of the order of treatment would therefore render the attack much slower. 1408 fault injections would be required to treat every S-box value for a given message, this would give an expected number of hypotheses of 55.5 per S-box and a total key search of $2^{54.3}$ possibilities. A total of 10 different messages would be required to bring the expected key search to $2^{39.5}$ which is more possible. The amount of fault attacks required may make this amount of fault injections unrealistic if the effect of the fault is not deterministic.

The attack presented in this section require a high degree of precision, as information is derived from a fault having occurred and then not having an effect on the ciphertext. This was possible due to the manner in which the fault was injected, but is unlikely to be possible with other fault injection methods. A fault is generally expected to be successful with a certain probability when it is applied to a chip [3], in this case the probability of success was equal to 1.

7 Modifying Unknown S-Box Values

If an S-box element can be modified, but the attacker does not know which element has been modified (i.e. Algorithm 4 is used), the attack described in Section 6 will not work. Nevertheless, an attack can be used by implementing the algorithms given in [2, 6].

If one S-box value is modified and used in round 15, and only in round 15, then the ideas described in [2] will apply. The modification of 1 S-box look-up in the fifteenth round will, on average, change the entry value for 3.2 different S-boxes in the sixteenth round, providing differential across these S-boxes for key hypothesis testing.

The advantage of this attack over the attack described in Section 6 is that the effect of the desired fault can be seen in the ciphertext. The fault can be detected by calculating the differential of the S-box output in the fifteenth round, which can be done by observing the ciphertext. If only one nibble in this value is not equal to zero, then there is a high probability that the corresponding S-box value was only used in the fifteenth round. The probability that this event occurs is $\left(\frac{63}{64}\right)^{15} \frac{1}{64} = 0.0123$.

This probability is high enough that an attacker can conduct the attack numerous times until the desired event is observed. Some key information can then be derived and the process repeated.

There is a possibility that the S-box is used in the fourteenth round and that this will yield a value that will be detected as a S-box value used in the fifteenth round. This occurs when the modification in the fourteenth round produces a 1 bit fault (all the output bits go to different S-boxes). There are 4 possible values among the 15 possible faults that will produce this effect. Half of these values will modify more than one S-box in the fifteenth round, i.e. they will span two S-boxes due to the expansion permutation. This leaves only two possible values from the fifteen possible faults. The probability of a false positive is therefore $\frac{2}{15}\left(\frac{63}{64}\right)^{15} \frac{1}{64} = 0.00165$.

The probability of a false positive is relatively high when compared to the probability of the event that will enable the attack. Approximately 1 in 7 detections will be false positives. However, as described in [6], the false hypotheses introduced by these false positives will not have a major effect on the success of the attack.

As detailed in Section 6, S-box values are usually stored in a compressed state so an attacker may be forced to modify several S-box entries at once. If two S-box entries are modified the probability of one of the values being used in the fifteenth round is $2\left(\frac{63}{64}\right)^{15} \frac{1}{64} \left(\frac{63}{64}\right)^{16} = 0.0192$.

This is more efficient than modifying 1 S-box value as the probability of the S-box value being used in the fifteenth round is higher. This probability will change following the method of S-box compression, but only the case under study is analysed.

The probability of one S-box value being used in the fourteenth round and causing a false positive can be calculated as before. If two modified S-box values are used in the fourteenth round this can also provoke a false positive if two one-bit errors are caused and these bits are used in the same S-box without being reproduced by the expansion permutation in the fifteenth round. This probability was derived by simulating all the possible combinations as 89/147456.

The overall probability of a false positive is therefore $2\frac{2}{15}\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16} + \frac{89}{147456}\left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^2 = 0.00256$.

The probability of a false positive given that a detection has occurred is about the same ($\approx 2/15$) given that the event has been detected for both implementations. The implementation using a compressed S-box will provide results quicker as the desired event occurs with a higher probability.

This attack was implemented on the same chip as the attack described in Section 6 because the fault used was ideal for modifying the S-box values as they were created. The first attempt at this attack was against a DES implementation that just used data masking and constructed S-boxes using Algorithm 4. The tools conducting the attack waited until at least 1 differential had been found across each S-box before conducting an exhaustive search of the hypotheses derived from the fault injection. The tools found the key after 8 minutes.

A second attempt was conducted with the addition of random delays in hardware and software, so that a fault would be produced with a lower probability. The same tools took 20 minutes to derive the key.

This attack was easier to implement than the attack described in Section 6, as only one fault injection position was needed to attack a random S-box entry. In the previous attack it was necessary to shift the position of the fault injection for each new fault injection attempt.

In the case of an implementation using compressed S-boxes it would be logical to use the event of both faulty S-box values being used in the fifteenth round. As previously, this can be observed by looking for two nibbles with a non-zero differential in the ciphertext. This information can be combined with the event of one nibble having a differential in the ciphertext. The probability of this occurring is $2\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16} + \left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^2 = 0.0193$.

As previously, there is a chance of a false positive. In this case the events of one or two modified S-box values being used in the fourteenth round could potentially simulate one or two changed values in the fifteenth round. If one S-box value is changed in the fourteenth round the probability that two values are modified in the fifteenth round is 1/5, if two values are changed in the fourteenth round the probability that two values are changed in the fifteenth round is 914609/29491200. Again, these were derived by simulating all the possible combinations. The probability of a false positive is therefore $2\frac{2}{15}\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16} + \frac{89}{147456}\left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^2 + 2\frac{1}{5}\left(\frac{63}{64}\right)^{15}\frac{1}{64}\left(\frac{63}{64}\right)^{16} + \frac{914609}{29491200}\left(\left(\frac{63}{64}\right)^{15}\frac{1}{64}\right)^2 = 0.00640$.

The probability of getting useful information remains approximately the same as when only one modified S-box is considered, but the probability of a false positive is 2.5 times greater. The data acquired will therefore be much more noisy and will increase the amount of time required to conduct the attack. There is therefore little interest in conducting the attack in this manner.

8 Countermeasures

There are several countermeasures that can be used to protect an algorithm against this type of attack. As has been described above, randomisation of the algorithm is not an efficient countermeasure against this fault attack.

Random Delay: If a high degree of precision is required the attack could be slowed to the point where an attacker will not believe the attack is possible. This applies to both hardware and software random delays. A study of the effect of random delays on DPA is given in [5], similar effects will be seen when this is used against fault attacks.

Checksums: If S-boxes need to be constructed in RAM they need to be protected by a checksum. The simplest method of achieving this would be to XOR all the values together after the table has been created i.e. after the table has been written to memory. This has the added advantage of removing the randomisation, as the amount of entries in the S-box will be an even number. Nevertheless, this is not adequate to defend against the attacks described above. If the checksum is on 1 byte an attacker could modify several values and have a probability of 1/256 of having a valid checksum. A second checksum calculated in a different manner could remove this problem, as the second checksum can be chosen such that there is no fault that will allow both checksums to remain valid.

Redundancy: It is already known that it is advisable to repeat the first 2 or 3 rounds of a secret key algorithm to protect against attacks like [7]. The initial functions can be repeated and the memory contents verified, in the same way that rounds of an algorithm are repeated to ensure no exploitable faults can be injected. However, this is prohibitively time consuming especially for the construction of S-boxes.

Memory Randomisation: All "work" areas of RAM used can be filled with independent random values before the start of the algorithm. The feasibility of the attack would then rest on the quality of the random values used. If, for example, an LFSR was used to generate these values it may be possible to predict the value of one byte if the previous byte is known. This could mean that the attack described in Section 4 is still possible with very little change i.e. the end search would be 2^{16} rather than 2^8 because an attacker would have to exhaust the possible initial values of the random used.

9 Acknowledgements

The authors would like to thank Pascal Moitrel and Christophe Mourtel who designed and built the hardware mentioned in this paper, which enabled us to implement the attacks described above. The work described in this paper has been financially supported by the European Commission through the IST Program under Contract IST-2002-507932 ECRYPT.

10 Conclusion

Several different attacks where faults were used to generate faults at the beginning of a secure implementations of AES and DES were presented. The implementations of some of these attacks have been briefly described. The algorithms were chosen because the source code for several different implementations was already available.

These attacks are generic attacks and can be considered to apply to any secret key implementation. These attacks show that DPA countermeasures are not an intrinsic barrier against fault attacks and that depending on round redundancy is not sufficient to achieve a secure implementation on smart cards.

References

1. M.-L. Akkar and C. Giraud. An implementation of DES and AES secure against some attacks. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptogaphic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer-Verlag, 2001.
2. E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In B. S. Kaliski Jr., editor, *Advances in Cryptology — CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer-Verlag, 1997.
3. J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). In R. N. Wright, editor, *Financial Cryptography — FC 2003*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer-Verlag, 2003.
4. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards approaches to counteract power-analysis attacks. In M. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer-Verlag, 1999.
5. C. Clavier, J.-S. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer-Verlag, 2000.
6. C. Giraud and H. Thiebeauld. A survey on fault attacks. In Y. Deswarte and A. A. El Kalam, editors, *Smart Card Research and Advanced Applications VI — 18th IFIP World Computer Congress*, pages 159–176. Kluwer Academic, 2004.
7. L. Hemme. A differential fault attack against early rounds of (triple-)DES. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 254–267. Springer-Verlag, 2004.
8. D. Knuth. *The Art of Computer Programming*, volume 2, Seminumerical Algorithms. Addison–Wesley, third edition, 2001.
9. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
10. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

11. D. Naccache, P. Q. Nguyễn, M. Tunstall, and C. Whelan. Experimenting with faults, lattices and the DSA. In S. Vaudenay, editor, *Public Key Cryptography — PKC 2005*, volume 3386 of *Lecture Notes in Computer Science*, pages 16–28. Springer-Verlag, 2005.