Round Reduction Using Faults

Hamid Choukri[1] and Michael Tunstall[1,2]

[1] Gemplus Card International, Applied Research & Security Centre,
Avenue des Jujubiers, La Ciotat, F-13705, France.
hamid.choukri@gemplus.com, michael.tunstall@gemplus.com
[2] Royal Holloway, University of London, Information Security Group,
Egham, Surrey TW20 0EX, UK.
m.j.tunstall@rhul.ac.uk

Abstract. This paper presents a practical implementation of a fault attack implemented on a Silvercard (a freely available smart card based on a PIC16F877 produced by Microchip). The aim of the fault attack is to effectively reduce the number of rounds of a secret key algorithm. The simplest case of reducing the number of rounds to one was chosen to facilitate subsequent cryptanalysis.
The fault injection method used is a glitch on the power supplied to the smart card. The manner in which this changes the functioning of the smart card is described, followed by how this effect can then be used to produce the desired result. A description of how this was applied to an AES implementation is given. Lastly, Various generic countermeasures are discussed to show how this type of attack can be prevented.

1 Introduction

Secret key cryptographic algorithms such as DES and AES are based on a function that is computed iteratively as a series of rounds to provide a high level of security. This function is referred to as the round function. In [2] the idea was put forward to implement a fault attack that would effectively reduce the number of rounds of a secret key cryptographic algorithm to enable the key to be derived.

In this paper an attack on AES is demonstrated which effectively reduces the number of rounds of an AES to one. The cryptanalysis of the resulting algorithm is simple and only requires two plaintext/ciphertexts pairs.

The method used to characterise the fault injection technique and then how such an attack can be achieved in practice. A transient glitch is used that produces a provisional fault within the chip. The actual effect of the fault within the chip cannot be described accurately without reverse engineering the chip, but some hypotheses are proposed as to what impact the fault may have.

It should be noted that the Silvercard contains no sensors that test the smart cards environment. This is a standard feature of all modern smart cards and are designed to thwart fault attacks such as that presented in this paper. The research was more oriented towards the possible effects of fault injection and fault analysis rather than the possible effects on actual smart cards.

The AES algorithm implementation used is a naive one, in the sense that no countermeasures are present against any sort of attack. The implementation will be vulnerable to power analysis and other forms of fault attacks. The aim being to show that precise faults can be induced within a chip that can lead to theoretically simple attacks.

2 Fault Injection

There are various different ways in which a fault can be induced in a microcontroller. These include methods such as a particle accelerator [3], a laser [8] or light [10]. These types of fault injection need the chip to be decapsulated so that the surface of the chip can be accessed. A simpler method of fault injection is to insert a glitch on the power supply pin, so that the processor misinterprets an instruction or a variable. A description of the different types of glitches that can be applicable to a microcontroller can be found in [1].

Figure 1 represents a SPICE (a general-purpose circuit simulation program [12]) simulation of an inverter's response to a transient variation on the power supply voltage. The goal is to give an example of a glitch's effect on the chip. An inverter was chosen as this is the basic building block from which logical gates can be created.

When the inverter output is set to 0 the glitch has no noticeable effect. However, when the output is switched to 1 the glitch changes the output value quite considerably. This shows that a glitch will produce an effect within the chip but unfortunately we cannot predict exactly what this will be. Similar effects can be produced using a laser [6].

In order to find a glitch that would have this type of impact on a smart card, an implementation of AES was used as a way of detecting a glitch. Numerous different glitch configurations were applied to AES, with no effort made to produce any particular effect. Every time an erroneous result was produced, the configuration that produced that result was recorded.

The external clock speed was varied between 1 MHz and 5 MHz in steps of 1 MHz. The size of the glitch varied between 1 clock cycle and 10 clock cycles in steps of 1 clock cycle. The applied voltage started at 3 volts and was incremented in steps of 0.5 volts to 5 volts. All the possible combinations of these three parameters were tested at 200 different positions in the computation of an AES. The voltage applied during the glitch was determined by dichotomy by finding the voltage limit at which the smart card did not respond correctly. The boundaries set for the dichotomy were between 0.25 volts and the voltage applied during the normal functioning of the smart card.

The corrupt responses were used to determine the different glitch configurations that succeeded in inducing a fault. Various different configurations that worked with an external clock set to 5 MHz. The smallest glitch size that created a fault was chosen to give as much precision as possible when an attempt is made to use this in a fault attack. In this implementation this was one clock cycle. The voltage applied to the card was chosen arbitrarily amongst those configurations

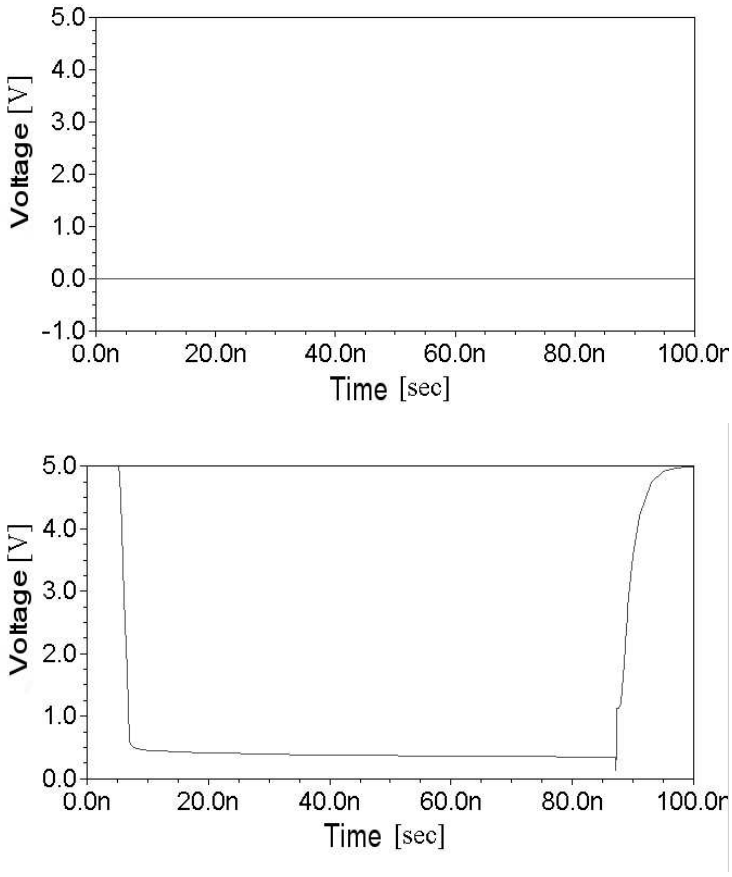

Fig. 1. The upper image shows the response of the inverter to a glitch when its output should be 0. The lower image shows the response when the output should be 1. In each case, a glitch has been applied to the supply voltage of the inverter. The effect can clearly be seen in the lower image.

that worked. The voltage level of the glitch itself was not deemed to have any importance, as it was assumed to be variable given the chip's behaviour would change as it heats up. This process took approximately 24 hours.

3 The Fault Target

The model used to try and imagine the effect of a fault during the execution of AES was that of a fault producing a change in the code executed, as described in [1]. For this reason the code executed was examined to determine where a fault could be injected.

In general, the implementation of a secret key cryptographic algorithm in the PIC assembly language will have the following format.

```
   movlw    0Ah
   movwf    RoundCounter
RoundLabel

   call     RoundFunction

   decfsz   RoundCounter
   goto     RoundLabel
```

The RAM variable (`RoundCounter`) is set to the number of rounds required, which for the example that is described in this paper (AES) the value loaded into the counter is 0A in hexadecimal. The round function is executed, which has been represented by a call to the function `RoundFunction`. The `RoundCounter` variable is then decremented, and the round is repeated until `RoundCounter` is equal to zero, at which point the loop exits. It is this loop that we are trying to change so that it exits earlier than expected.

The target of the fault is the `decfsz` step, which consists of a decrement, a test, followed by a conditional jump. The conditional jump is present as jump of one instruction when the test is positive; otherwise the next instruction is executed. The aim of the attack is to reduce the algorithm to one round. It is not possible to remove the first round entirely as the first conditional test is after the first round. The instruction can be broken down into three different tasks, the first being the decrementation:

```
Decrement task:
        RoundCounter <= RoundCounter - 1
```

There are three tasks that are necessary when a variable in RAM is decremented by 1. All three of these tasks represent a potential point for a fault to arise. They are:

1. The transfer of the RAM variable contents to the internal processor accumulator.

2. Decrementation of the internal processor accumulator.
3. Transfer of the contents of the internal processor accumulator to RAM.

After the `RoundCounter` variable is updated its content is tested to see it is equal to zero.

```
Testing task:
        If (RoundCounter == 0)
            Status <= 1
        Else
            Status <= 0
```

An injected fault could potentially have an effect during two different phases of this task execution:

1. The Register content test.
2. The change to the status value.

Following the status of the test the program counter (`PC`) will take one of two different values.

```
Jump task:
        If (Status == 1)
            PC <= PC1
        Else
            PC <= PC2
```

If the Status `variable` is equal to 1, the hardware sets the program counter to `PC1`. Otherwise it sets the program counter to `PC2`. In the case of the `decfsz` command `PC1` will be equal to the program counter plus two and `PC2` will be the program counter plus one. This gives two cases that could be potential targets for a fault attack.

1. The Status value test.
2. The modification to the value of `PC`.

In the case of the PIC16F877 chip, this instruction executes in one clock cycle when the test is negative, and two when the following instruction is not executed. The next instruction is a `goto` that will change the program counter so that the round function is executed again, which takes two clock cycles to execute. This means that the target for the attack is three cycles long, where the first two tasks are present in the `decfsz` step and the last is in the `goto`.

4 Reducing the number of rounds

Once the target of the fault has been determined, this target needs to be found within the command that executes the algorithm under attack. The execution

of the code described above can be easily detected by monitoring the current consumption of a smart card as the round function will create a pattern that will repeat itself. Figure 2 gives the example of the current consumption during a command that implements AES. A pattern that repeats itself nine times is visible, with a tenth pattern that is slightly shorter due to the absence of the MixColumn function.

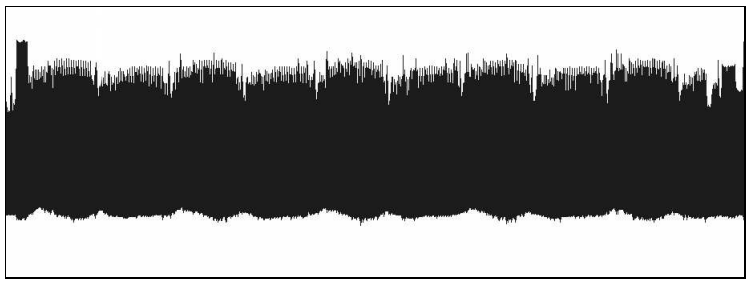

Fig. 2. A current consumption waveform that shows the rounds of AES visible as a repeating pattern.

This can give an approximate position of where the target opcode is executed. The more effort that is put into this stage, the quicker the attack can proceed. If the functions within the round function can be identified, the amount of positions that need to be tried to get the desired results can be reduced. There is a lower limit to the amount of positions that will need to be tested as the test that will exit the loop could be placed either before or after the MixColumn function.

The simplest case was taken by applying a glitch at what appears to be the middle of the first round, and incrementing the glitch position in steps of one clock cycle to the middle of the second round. This took approximately 48 hours of testing to scan 9000 different positions. The same parameters as chosen during the characterization stage were chosen and the voltage applied during the glitch varying in the same way. This produced five different positions where AES had been reduced to one round in the fashion that was expected.

If an open card is available where it is possible to change the key used, the desired output can be searched for within the data acquired. This will show when the algorithm has been reduced to one round.

If an open card is not available, the I/O channel needs to be acquired each time the glitch is applied. This can be used to signal when the attack has been successful as the time in between the command and the response will shorten as 9 rounds of AES have been removed. An example of this is shown in figure 3 where the shortening of the command can be seen in the I/O and is confirmed by the current consumption. This shortening of the command time is only significant when the status returned by the card implies that everything has executed

correctly and sixteen bytes have been returned. Otherwise it could be confused with a warm reset provoked by an overly aggressive glitch.

Once a position has been found this can be attacked several times with different plaintexts to acquire the data needed to derive AES key being used in the card under attack.

The five different positions where a glitch had previously worked were attacked with three different massages. The data that had been acquired during the characterisation stage did not appear to give the same result with a given glitch configuration. For this reason the voltage level to which the glitch dropped was varied in the same fashion as used during the characterisation phase. This resulted in 150 different attempts to reduce AES to one round. All the data returned with the corresponding message were kept for subsequent interpretation.

5 Interpreting the Results

An AES that has been reduced to one round will consist of the following functions:

```
AddRoundKey();
ShiftRows();
SubBytes();
MixColumns();
AddRoundKey();
```

The AddRoundKey function is a xor with the relevant subkey, the ShiftRows is a Bytewise permutation, the ByteSub function is a non-linear transformation normally expressed as a substitution table and the MixColumn is a linear transformation in $GF(8)$. These functions are described in [11]. As stated before, the MixColumn function is not necessarily present depending on the implementation.

If we present two plaintexts (m_1 and m_2) to this algorithm to produce two corrupt ciphertexts (c_1 and c_2). If the first subkey is referred to as k, then the data acquired can be compared in the following fashion:

$$\text{SubBytes}\left(m_1 \oplus k\right) \oplus \text{SubBytes}\left(m_2 \oplus k\right) = \text{MixColumn}^{-1}\left(c_1 \oplus c_2\right) \quad (1)$$

The ShiftRow function is not taken into account as it is a bytewise permutation. The last AddRoundKey is ignored as the effect of this function is removed by xoring the two corrupt ciphertexts together.

The right hand side of the equation expresses the Hamming distance between AES calculations after the SubByte function. As m_1 and m_2 are known this equation can be evaluated for each byte with all the possible values of the first subkey that is xored with that byte.

With two ciphertexts this will usually lead to two different hypotheses for each byte of the first subkey. There is no calculation involved in the generation

Fig. 3. The acquisitions from top to bottom: The I/O trace of a normal AES execution and its current consumption, where the rounds can be seen. Followed by the I/O trace of an AES execution where the number of rounds have been reduced to one and the current consumption showing the reduction in the number of rounds.

of the first subkey so these hypotheses apply directly to the key. This can be seen by calculating a table of differentials as used in [4] for the DES. This leads to an exhaustive search of 2^{16} possible keys, as:

$$\left(\frac{\Sigma(\text{Non zero differentials})}{\#(\text{Non zero differentials})}\right)^{16} = 2^{16} \tag{2}$$

If three ciphertexts are available then the number of keys that are included in an exhaustive search are much reduced. This is because three different comparisons can be made using the formula described above. Each one providing approximately two different hypotheses for each byte on the key.

In practice, the data acquired is likely to be noisy as there will be some faults that will produce a corrupt ciphertext and change the I/O but will not have the desired properties. For this reason it is best to compare each ciphertext that comes from a command that returned 16 bytes that are not equal to the correct ciphertext with all the others that fulfill the same criteria. Otherwise the analysis could be made impossible fulfils the required criteria with all of the others so that the analysis is not made impossible.

This does not slow down the attack as if one of the two, or both, ciphertexts does not have the required properties the calculated hamming distance will be meaningless, as deriving a list of hypotheses will not be possible. The probability that random data will produce at least one hypothesis for every byte of the key is:

$$\left(\frac{\#(\text{Non zero differentials})}{256^2}\right)^{16} = 3.14 \times 10^{-3} \tag{3}$$

Even with a large amount of acquired data the time needed to search for the key remains reasonable.

From the 150 glitch attempts made against the Silvercard a pair of results that produced was found almost instantly using this method.

6 Other Algorithms

The attack presented in this paper can be directly applied to other secret key algorithms. The main difference is in the manner in which the data acquired is exploited. In the case of the DES, hypotheses on the key can be derived by inspection due to the structure of the Feistel network. This will give a keyspace of 2^{24} to be searched from one corrupt ciphertext. This can be further reduced by examining other corrupt ciphertexts to derive the first subkey. Reducing subsequent DES executions to two rounds can give direct information on the 8 bits not present in the first subkey. As stated in [2] this can provide a method of attacking the DES without knowledge of the plaintexts.

7 Countermeasures

Software countermeasures would include having some form of redundancy with the register referred to RoundCounter above. This would involve having two tests at the end of each round. In the example presented, a glitch of one clock cycle was used to provide as much precision as possible. It is perfectly possible to find a glitch that lasts for several cycles that will induce a fault. A large enough glitch could potentially effect both tests if they are made one after the other. There is also the possibility of using multiple glitches, one for each test, but it is unlikely to be practical. The effectiveness of this attack relies on being able to detect that the attack has been successful by observing the I/O trace. There is no simple way of detecting when one glitch has succeeded in changing one test before trying to position the second glitch on the second test.

Another method would be to repeat all, or part, of the algorithm in such a way that any reduction in the number of rounds could be detected. This can have a large impact on the performance of the algorithm if the whole algorithm is repeated.

A generic countermeasure is the inclusion of a random delay before the algorithm so that it is difficult to find the correct position to attack. This will not stop an attack but will make it more difficult to achieve. If the same procedure as described in this paper was applied to a smart card with a random delay, the first results would probably be discouraging until the random delay was noticed. Once the random delay has been assessed, it would be possible to design the attack so that the random effects could be ignored, although this would greatly increase the time required to realize the attack.

However, these countermeasures are more to prevent this particular attack. All modern microcontrollers used in smart cards have sensors designed to detect this sort of attack. It is rare to find a smart card that is vulnerable to this method of fault injection. The normal response of a smart card is either to become mute or to reset itself. Whilst the software countermeasures mentioned above can defend against the attack described above, hardware countermeasures are required to have a secure implementation. The only advantage of including the software countermeasures is that they will also protect the smart card against other forms of fault injection, which may be able to produce a similar effect.

8 Conclusion

A generic attack against secret key algorithms in smart cards and its implementation has been described. It should be noted that the AES implementation studied was a naive implementation to show the potential of theoretically simple fault attacks.

The described attack will only work against implementations that are implemented as shown above. It is possible to implement algorithms so that each round is called independently *i.e.*

```
call     RoundFunction
call     RoundFunction
call     RoundFunction
...
```

This sort or implementation will not be effected by the fault attack described above, and could even be considered an effective countermeasure. However, it should be possible to remove one round from the algorithm. This will reverse the situation as it will then be possible to derive hypotheses on the last subkey rather than the first. This is because it will be possible to compare the effect of the last round on the calculation by comparing the actual ciphertext with a ciphertext that has a missing round.

The software countermeasures described can be implemented fairly easily but only defend against the attack described. It would still be possible to do other things with the fault injection method used against AES, which could still compromise the security of the smart card.

Some examples of other fault attacks that have been implemented against AES include [7], [5], and [9]. These attacks do not really compare well with the attack described above, as they exploit the mathematical properties of the algorithm itself. The advantage of these attacks is that less control is required over where the fault is produced, but a more complex mathematical treatment is necessary.

The attack described in this paper requires a high degree of control with regard to where the fault takes place but relatively little calculation is required after acquiring the desired corrupt ciphertexts.

References

1. R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. In *Proceedings of the Second USENIX Workshop of Electronic Commerce*, pages 1–11, November 1996.
2. R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. *Security Protocols*, pages 125–136, 1998. Lecture Notes in Computer Science No. 1361.
3. G. Berger and G. Ryckewaert. The heavy ion irradiation facility at CYCLONE - a dedicated SEE beam line. *IEEE Radiation Effects Data Workshop*, page 78, 1996.
4. E. Biham and A. Shamir. Differential cryptanalysis of DES-like cryptosystems. *Proceedings of CRYPTO 90*, pages 2–21, 1991. Lecture Notes in Computer Science No. 537.
5. J. Blömer and J-P. Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). *Proceedings of Financial Cryptography 2003*, pages 162–181, 2003. Lecture Notes in Computer Science No. 2742.
6. F. Darracq, T. Beauchne, D. Lewis, V. Pouget, H. Lapuyade, P. Fouillat, and A. Touboul. Single-event sensitivity of a single sram cell. In *IEEE Transactions on Nuclear Science*, volume 49, pages 1486–1490, 2002.
7. C. Giraud. DFA on AES. Cryptology ePrint Archive: Report 2003/008. http://www.iacr.org.

8. D. Lewis, V. Pouget, F. Beaudoin, P. Perdu, H. Lapuyade, P. Fouillat, and A. Touboul. Backside laser testing of ICs for SET sensitivity evaluation. *IEEE Transactions on Nuclear Science*, 48:2193–2201, 2001.
9. G. Piret and J. J. Quisquater. A differential fault attack technique against SPN structures, with application to the AES and Khazad. *Cryptographic Hardware and Embedded Systems Workshop (CHES-2003)*, pages 77–88, 2003. Lecture Notes in Computer Science No. 2779.
10. S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Cryptographic Hardware and Embedded Systems Workshop (CHES-2002)*, pages 2–12, 2002. Lecture Notes in Computer Science No. 2523.
11. Federal Information Processing Standards. Advanced Encryption Standard (AES). FIPS publication 197.
12. A. Vladimirescu. *The SPICE Book*. J. Wiley & Sons, 1993.