A Simple Power Analysis Attack on the TwoFish Key Schedule

Jose Javier Gonzalez Ortiz*
Universidad Pontificia Comillas

Kevin J. Compton†
University of Michigan

August 24, 2016

Abstract

This paper introduces an SPA power attack on the 8-bit implementation of the TwoFish block cipher. The attack is able to unequivocally recover the secret key even under substantial amounts of error. An initial algorithm is described using exhaustive search on error free data. An error resistant algorithm is later described. It employs several thresholding preprocessing stages followed by a combined approach of least mean squares and an optimized Hamming mask search. Further analysis of 32 and 64-bit TwoFish implementations reveals that they are similarly vulnerable to the described SPA attack.

Keywords: *TwoFish, SPA, Power Attack, Block Cipher, Error Tolerance*

1 Introduction

In 1997, the National Institute of Standards and Technology (NIST) started the Advanced Encryption Standard process in order to designate a successor of the aged Digital Encryption Standard (DES). Among the five finalists is the TwoFish block cipher, and although in the end Rijndael was designated as the winner of the AES contest, TwoFish was one of the strongest contenders, excelling in categories such as general security, hardware performance and design features. After the introduction of linear and differential attacks they became a concern in the design of new ciphers. Thus, the encryption and key schedule algorithms of the submissions were designed to prevent these types of attacks. Nevertheless, among the finalists both AES and Serpent have been found to be susceptible to *side-channel attacks*. These attacks focus on the information that can be gained from the physical implementation of cryptosystems in especially accessible systems such as smart cards.

In 1999 Biham and Shamir [BS99] carried out a preliminary analysis of power attack susceptibility of various AES cadidates. They assessed that TwoFish Key Schedule had a complex structure and seemed not to reveal direct information on the key bits from Hamming measurements. The results presented in this work refute this assertion since an efficient and robust SPA attack that retrieves the secret key was found.

*jjgo@umich.edu
†kjc@umich.edu

We present a side channel attack on the TwoFish key schedule which finds the secret key in just one execution of the algorithm. Further analysis of the algorithm and the different existing implementations render all of them vulnerable to this attack, posing a concern for the way 8-bit manipulations are described and performed in the TwoFish Key Schedule.

We will start by providing a background in the TwoFish Block Cipher operation and the formulation of its associated Key Schedule in Section 2. Section 3 describes an attack provided error free data by employing a reduced exhaustive search for each byte of the key. The attack is then incrementally improved in Section 4 to cope with a more than significant amount of measurement error. Section 5 displays the accuracies achieved under varying degrees of error and key size as well as the elapsed times. These results are later analyzed on Section 6 and further work is outlined in Section 7.

1.1 Previous Work

Side Channel Attacks were first described in 1996 by Kocher [Koc96] with the introduction of timing attacks, which had the ability of compromising a cryptosystem by analyzing the time taken to execute specific parts of cryptographic algorithms. However, these attacks can be easily overcome by employing simple software countermeasures. Since access to the hardware was one of the assumptions, attacks that used the power consumption of the cryptosystem started developing. The first research power attack is credited to *Kocher et al* [KJJ99], in which they were able to obtain information of the data manipulated by the processor by carefully measuring the power consumption of a CMOS chip. In his work, two different types of power attacks are described:

- **Differential Power Attacks** (DPA) – In an analogous fashion to classical differential cryptanalysis, DPAs try to find patterns and relationships between plaintexts and their associated power traces. Similarly, a large amount of samples are required for the results to convey statistical significance.
- **Simple Power Attacks** (SPA) – An SPA focuses in particular vulnerabilities of the algorithm design. These vulnerabilities could leak enough sensitive information to compromise the confidentiality of the encryption or even the secret key itself.

Since microprocessors perform discrete operations on blocks of data in a sequential fashion, physical imperfections of the system make possible to correlate the Hamming weights of the manipulated values and the power utilization. Research has proved that this correlation is significant enough to find Hamming weights of numerous intermediate variables from the power utilization, as shown in the works of [MDS02] and [MS00].

The concern of these types of attacks has lead to a number of research efforts [Mes01], [MOP07] to thwart them by masquerading the power consumption values in order to break the correlation that Power Attack use as a basis to acquire information. Nevertheless, they are still far from being industry standards and most modern smartcards and ASICs do not yet include mechanisms like those ones by default. Thus, for the purpose of this paper said techniques have not been considered.

Recent research efforts have shown that both Rijndael [VBC05] and Serpent [CTV09] key schedules are susceptible to Simple Power Attacks. Both attacks used a power trace of the algorithm to unequivocally

recover the secret key used in the encryption. In both cases the weakness arose from the key schedule computation; patterns in the hamming weights of the key schedule algorithm revealed enough information to compromise the secret key. TwoFish's key schedule follows different principles and constructions to generate the necessary subkeys but it is nonetheless susceptible to this type of attack as this paper demonstrates.

Even though TwoFish smart-card implementation performance and versatility have been throughly analyzed [Kea99], [RHW11], no power attacks have been found for the encryption algorithm or the key schedule. Nevertheless, analysis on the key schedule such as [MM99] have outlined some deficiencies and weaknesses in the cipher.

2 TwoFish Block Cipher

TwoFish is a symmetric key block cipher with a block size of 128 bits and key sizes up to 256 bits. It was one of the five finalists of the Advanced Encryption Standard contest. It was submitted by Schneier et al. [SKW+98].

Twofish features pre-computed key-dependent S-boxes, and a relatively complex key schedule. One half of an n-bit key is used as the actual encryption key and the other half of the n-bit key is used to modify the encryption algorithm. Twofish borrows some elements from other designs such as a Feistel structure from DES or the pseudo-Hadamard transform [STM10] from the SAFER family of ciphers [Mas94].

This section will briefly introduce the encryption scheme and key schedule in the TwoFish block cipher following the notation and terminology from [SKW+98].

2.1 The TwoFish Encryption Algorithm

The TwoFish encryption is a 16 round Feistel Network with both input and output whitening. Each round operates only in the higher 64 bits of the block and swaps both halves. A total of 40 subkeys are generated from the secret key, with each key being 32 bits. Keys $K_0 \ldots K_3$ are used for the input whitening, $K_4 \ldots K_7$ are used for the output whitening and $K_8 \ldots K_{39}$ are used as the round subkeys. Each round employs a function F which is a key-dependent permutation on 64-bit values. The function F splits the 64-bit string into two 32-bit substrings and applies the g function to each half. The function g applies a fixed number of S-box substitutions and XORs with parts of the secret key (the number of steps this is performed depends on the size of the key).

This is followed by a MDS (Maximum Distance Separable) Matrix transform, a Pseudo-Hadamard Transform and round key XOR with the two subkeys associated for that said round. Therefore each round r uses two subkeys as round key, namely K_{2r+8} and K_{2r+9}. Finally, the output is XORed with one half of the block following the Feistel Network scheme. Bitwise rotations and shifts are performed at strategic points in the encryption to maximize diffusion. They have been omitted to simplify the explanation. The algorithm can be visualized in Figure 2.1

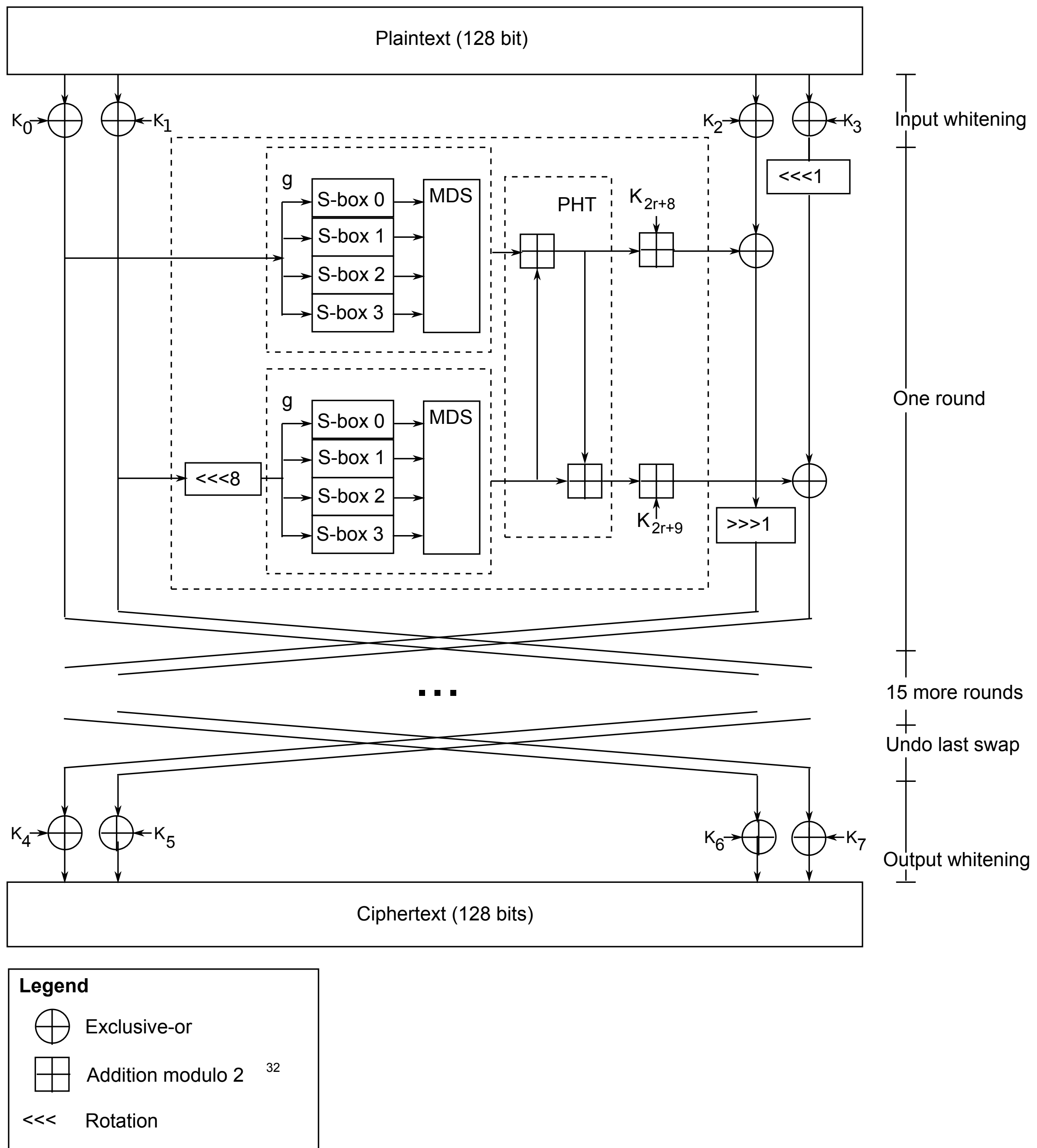


Figure 2.1: *TwoFish algorithm*

2.2 The TwoFish Key Schedule

The key schedule has to provide 40 32-bit words of expanded key $K_0, \ldots, K_{39}$. TwoFish is defined for sizes $N = \{128, 192, 256\}$. Keys shorter can be padded with zeros to the next larger key length. The key is split into $2R$ 32-bit words $K = (M_0, M_1, \ldots M_{2R-1})$ where $R = N/64$. To generate each subkey all the bits in the secret key are employed. The secret key is also employed to derive the vector $S = (S_{R-1}, S_{R-2}, \ldots, S_0)$ which is obtained by applying a Reed Solomon Transformation to the key. These values are used in the g function inside the F permutation as part of the encryption algorithm.

Subkeys are generated in even-odd pairs K_i, K_{i+1}, with even i. To generate this pair of keys, two 32-words are initialized, the first word has all its bytes equal to i and the second word has all its bytes equal to $i+1$. Then, both words go through R rounds, each round being composed of a specific substitution box arrangement followed by an XOR with a corresponding part of the secret key. This combination makes the S-boxes key dependent. For example, for $|K| = 128$ we have $R = 2$, so the h function will have two rounds as shown in Figure 2.2.

Next, the words go through another S-box substitution, and through a MDS transform. All of these 32 bit S-boxes are composed of a predefined choice of two 8-bit permutations $q0$ and $q1$. This choice is fixed and only depends on the stage of the function we are in. The transformation up to this point is defined to be the h function. Next, an 8 bit right rotation is applied in the odd word. Finally a Pseudo Hadamard Transform is applied to both values resulting in the pair of subkeys K_{2n}, K_{2n+1}.

We can see that the procedure is quite similar to that of the round function and in fact the g function can be expressed in terms of the h function. However, the round function does not use directly the bits of the secret key, but instead a a Reed Solomon Matrix is applied beforehand as we previously mentioned. For a further explanation refer to the equations and diagrams in [SKW$^+$98].

2.3 TwoFish Subkey generation

The main problem when using the notation in [SKW$^+$98] is that it describes the algorithm mainly in operations of 32-bit words. Since we would like to perform an attack that will use the trace of Hamming weights, we are interested in having a mathematical formulation that operates on 8-bit values. Therefore in this section we will derive the equations for the TwoFish key schedule in byte form, laying out the mathematical notation that we will use in the Simple Power Attack in the next section.

Although the secret key values are both used for the round function and the key schedule, the attack was found just by looking at Hamming weights of the key schedule algorithm, so only notation for the h function is going to be introduced.

In section 2.2 we described the h function involved in the key schedule procedure, which is depicted in Figure 2.2 for a 128-bit key. Expanding the diagram to take into account all the descriptions of the key schedule, we get the layout shown in Figure 2.3.

Here i is an even integer and the bytes $m_0, \ldots, m_{15}$ are the 16 bytes of the key. Blocks $q0$ and $q1$ are byte substitution boxes implemented using look-up tables, and their position along the rows and rounds is part of the specification of the algorithm. Variables v and w will be later used to express the

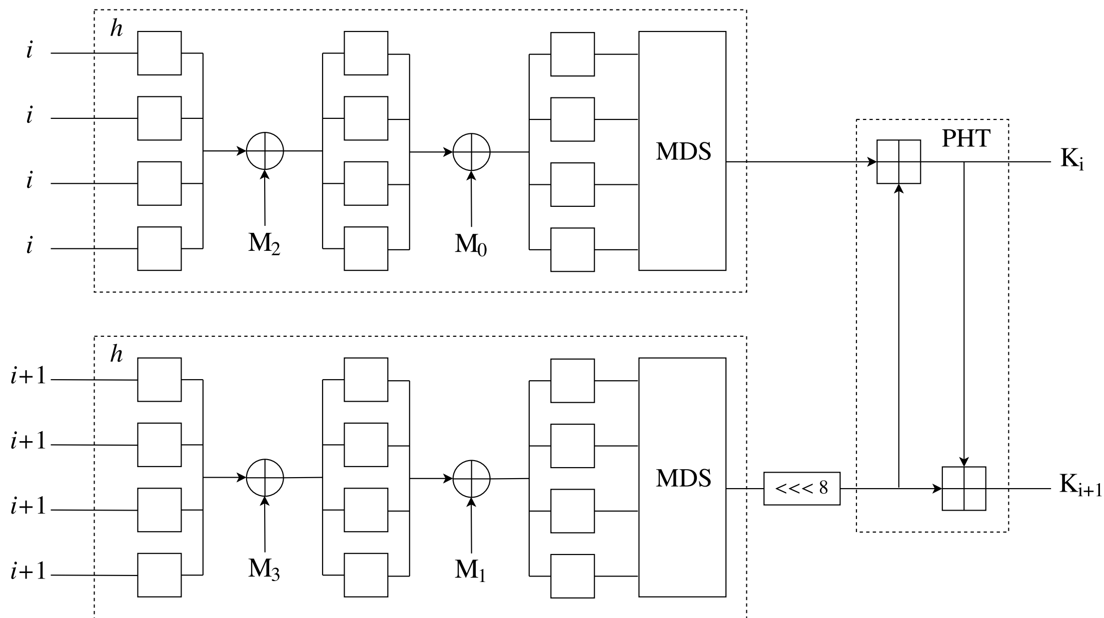


Figure 2.2: *TwoFish Key Schedule (128-bit key)*

relationship between the different rounds of the algorithm.Finally, the grey dots represent the places where a Hamming weight is retrieved from the trace in the algorithm described in Section 3.

As we can see from the diagram we will need at least three indexes in order to identify a particular Hamming weight inside the h function structure.

i – Identifies the subkey we are generating as stated previously. In general i will range from 0 to 39, since 40 subkeys need to be generated.

j – Specifies the byte within the 32-bit word taken in the h function. j will range from 0 to 3. In the diagram this is represented as rows.

k – Will identify in which round inside the h function we are at. Since the number of rounds will depend on the size of the secret key, k will range from 0 to R, where $R = |K|/64$.

In general, the key consists of $8R$ bytes, which are all used in every execution of the h function.

$$K = \{m_0, m_1, \ldots, m_{8R-1}\} \tag{2.1}$$

In order to describe the relationships between the intermediate values within the h function we can use two byte vectors $\boldsymbol{V}$ and $\boldsymbol{W}$ to represent the values before and after applying a S-box respectively.

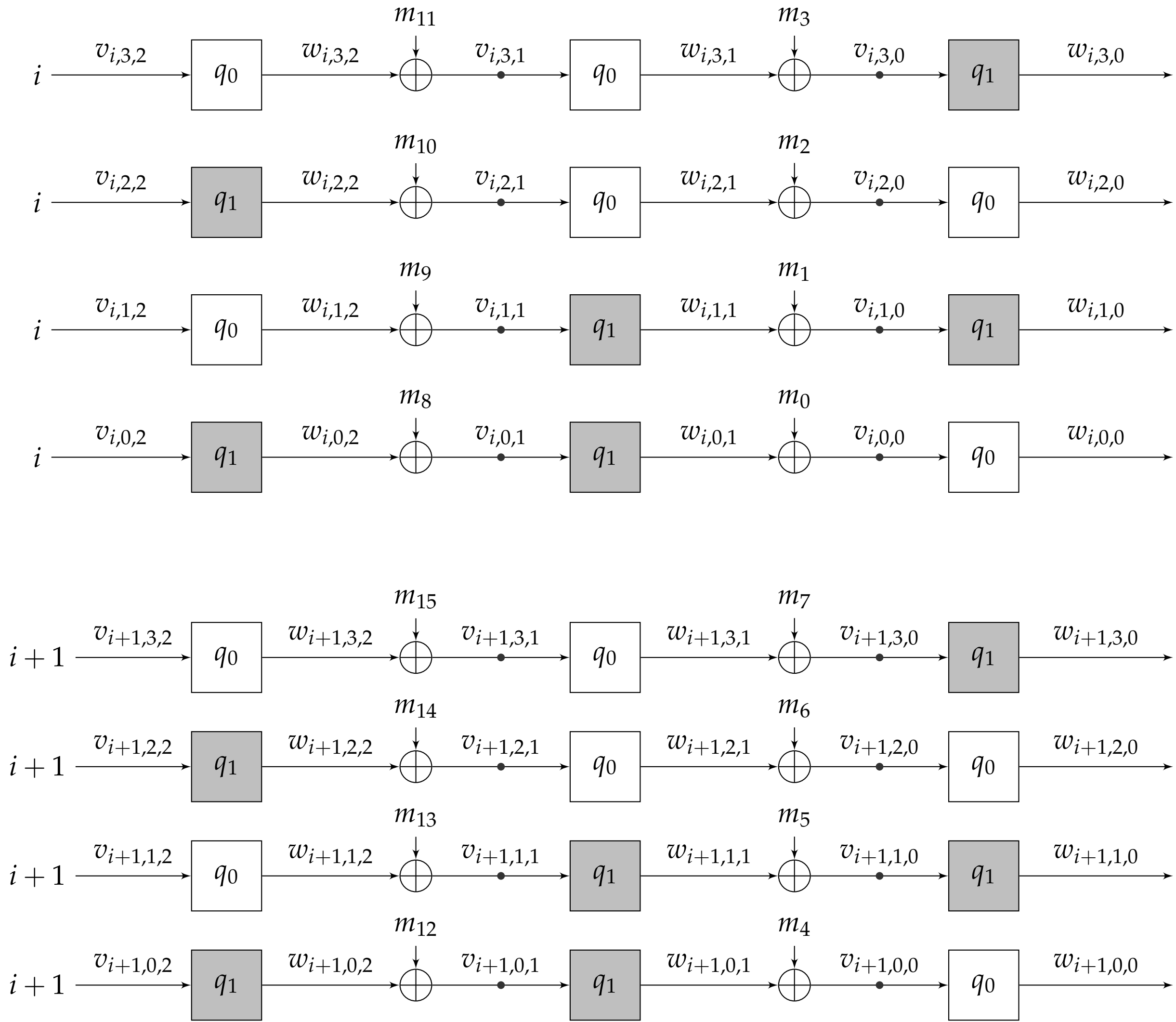


Figure 2.3: *Key schedule computations for a 128-bit key*

$$\boldsymbol{V} = \left\{ v_{i\,j\,k} \in \{0,1\}^8 : \quad i = 0,1,\ldots,39 \quad j = 0,1,2,3 \quad k = 0,1,\ldots,R \right\} \tag{2.2}$$

$$\boldsymbol{W} = \left\{ w_{i\,j\,k} \in \{0,1\}^8 : \quad i = 0,1,\ldots,39 \quad j = 0,1,2,3 \quad k = 0,1,\ldots,R \right\} \tag{2.3}$$

This notation is displayed in Figure 2.3. One important observation is that the index k decreases as we apply more rounds. This comes from the fact that increasing the key size adds more rounds to the left of the h function, leaving the rest of the rounds unaltered. Since the layout of $q0$ and $q1$ boxes depends both on j and k, the simplest way to define this relationship is to increment k from right to left. Furthermore, to generalize the notion of the S-boxes to these indexes, we can create a matrix P whose elements P_{jk} are 0 when the substitution used in row j and round k is $q0$ and 1 if it is $q1$. This allows us to succinctly

store the disposition of $q0$ and $q1$ in the algorithm and define a permutation $Q_{jk}[x]$ that will evaluate to either $q0$ or $q1$.

$$Q_{jk}[x] = \begin{cases} q_0[x] & \text{if} \quad P_{jk} = 0 \\ q_1[x] & \text{if} \quad P_{jk} = 1 \end{cases} \qquad P = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix} \tag{2.4}$$

Given all these definitions we can expresses the function h using the following relations.

$$v_{i,j,R} = i \tag{2.5}$$

$$w_{i,j,k} = Q_{jk}[v_{i,j,k}] \tag{2.6}$$

$$v_{i,j,(k-1)} = w_{i,j,k} \oplus m_l \tag{2.7}$$

As we see in equation (2.7), a index l for the specific byte of the key is required. In the diagram we could see directly where every byte of the key was used. In general, half of the executions of h will require even words of the key $M_e = \{M_0, M_2, \dots\}$ and the other half will require odd words of the key $M_0 = \{M_1, M_3 \dots\}$.

We can express l in terms of the rest of the indexes, so from now on when l is used, it can be thought as a function of i, j, k.

$$l(i, j, k) = 8(k-1) + j + 4(i \bmod 2) \tag{2.8}$$

3 TwoFish Key Schedule Power Analysis Attack

From the formulation described in the previous section we can see that each byte of the key m_l is used 20 different times in order to generate the 40 round subkeys. Since $m_l \in \{0,1\}^8$, knowing the Hamming weights of w_{ijk} should allow us to determine the secret key with a extremely high probability.

We can define the Hamming weight of a N-bit variable as follows, where b_i is the ith bit of x.

$$\begin{aligned} &\mathrm{H} : \{0,1\}^N \mapsto \mathbb{N} \\ &\mathrm{H}[x] = \sum_{i=0}^{N-1} b_i \end{aligned} \tag{3.1}$$

Since we want to compute Hamming weights it is useful to express $v_{i\,j\,k}$ and $w_{i\,j\,k}$ in binary form as

follows:

$$v_{i\,j\,k} = \sum_{n=0}^{7} c_{i\,j\,k\,n} \cdot 2^n \qquad c_{i\,j\,k\,n} \in \{0\,1\} \tag{3.2}$$

$$w_{i\,j\,k} = \sum_{n=0}^{7} d_{i\,j\,k\,n} \cdot 2^n \qquad d_{i\,j\,k\,n} \in \{0,1\} \tag{3.3}$$

Where $c_{i\,j\,k\,n}\, d_{i\,j\,k\,n}$ represent the nth bit of $v_{i\,j\,k}$ and $w_{i\,j\,k}$ respectively

In order to express the xor relation described in Equation (2.7), it will be convenient to have a bit representation of m_l, where $x_{l\,n}$ will be unknowns since they represent the individual bits of the secret key.

$$m_l = \sum_{n=0}^{7} x_{l\,n} \cdot 2^n \tag{3.4}$$

Now we can formulate the relationship of Equation 2.7 in a bitwise form

$$c_{i,j,(k-1),n} = d_{i,j,k,n} \oplus x_{l,n} \tag{3.5}$$

Taking Hamming weights in both sides of said equation renders the following expression.

$$\mathrm{H}[v_{i,j,(k-1)}] = \mathrm{H}[w_{i,j,k} \oplus m_l] = \sum_{n=0}^{7} d_{i,j,k,n} \oplus x_{l,n} \tag{3.6}$$

Looking into (3.6) we can see that we will have 40 equations and 16 variables, but even and odd equations will refer to different m_l and $x_{l,n}$. This is caused by the alternation in even and odd expressions. This allows us to split this system of equations into two disjoint systems of 20 equations and 8 variables since they are completely independent of one another.

Thus, for each j, k we will two systems of equations as shown in Equations (3.7) and (3.8).

$$\begin{cases} \mathrm{H}\left[v_{0,j,(k-1)}\right] = d_{0,j,k,0} \oplus x_{l,0} + d_{0,j,k,1} \oplus x_{l,1} + \ldots + d_{0,j,k,7} \oplus x_{l,7} \\ \mathrm{H}\left[v_{2,j,(k-1)}\right] = d_{2,j,k,0} \oplus x_{l,0} + d_{2,j,k,1} \oplus x_{l,1} + \ldots + d_{2,j,k,7} \oplus x_{l,7} \\ \mathrm{H}\left[v_{4,j,(k-1)}\right] = d_{4,j,k,0} \oplus x_{l,0} + d_{4,j,k,1} \oplus x_{l,1} + \ldots + d_{4,j,k,7} \oplus x_{l,7} \\ \qquad \ldots \\ \mathrm{H}\left[v_{38,j,(k-1)}\right] = d_{38,j,k,0} \oplus x_{l,0} + d_{38,j,k,1} \oplus x_{l,1} + \ldots + d_{38,j,R,7} \oplus x_{l,7} \end{cases} \tag{3.7}$$

$$\begin{cases} \mathrm{H}\Big[v_{1,j,(k-1)}\Big] = d_{1,j,k,0} \oplus x_{l',0} + d_{1,j,k,1} \oplus x_{l',1} + \ldots + d_{1,j,k,7} \oplus x_{l',7} \\ \mathrm{H}\Big[v_{3,j,(k-1)}\Big] = d_{3,j,k,0} \oplus x_{l',0} + d_{3,j,k,1} \oplus x_{l',1} + \ldots + d_{3,j,k,7} \oplus x_{l',7} \\ \mathrm{H}\Big[v_{5,j,(k-1)}\Big] = d_{5,j,k,0} \oplus x_{l',0} + d_{5,j,k,1} \oplus x_{l',1} + \ldots + d_{5,j,k,7} \oplus x_{l',7} \\ \qquad \ldots \\ \mathrm{H}\Big[v_{39,j,(k-1)}\Big] = d_{39,j,k,0} \oplus x_{l',0} + d_{39,j,k,1} \oplus x_{l',1} + \ldots + d_{39,j,k,7} \oplus x_{l'7} \end{cases} \tag{3.8}$$

In general, we can solve both these systems of equations as long as we know all $d_{i,j,k,n}$ for a fixed k, by simply using a brute force search in the value m_l. Note that since we are performing a Simple Power Attack, careful readings will allows to the values of $\mathrm{H}[v_{ij(k-1)}]$ for all i, j, k. The search only takes 2^8 different tries with each taking at most 20 equation evaluations. We simply iterate m_l from 0 to 255 and evaluate by xoring it with $w_{i,j,k}$ and checking against $\mathrm{H}[v_{i,j,(k-1)}]$ for each $i \in \{0, 2, \ldots 38\}$. To crack m'_l the same procedure is performed, except that this time i moves in the range $i \in \{1, 3, \ldots 39\}$. As soon as one equation is not met, we try the next possible value of m_l. Solving for a fixed j, k produces two keys m_l and $m'_l = m_{l+4}$ due to the even and odd scheme shown before. The algorithm employed for the brute force search is outlined in Algorithm 1.

While $\mathrm{H}[v_{ij(k-1)}]$ values are easy to derive from the execution of the algorithm, bit values $d_{i,j,k,n}$ will generally be unknown due to the dependence between rounds. Nevertheless, we can compute the values of $d_{i,j,k,n}$ for $k = R$. The values of $v_{i,j,R}$ are predetermined (Equation (2.5)) and $Q_{jk}[x]$ is a known fixed permutation so we can derive the values of $w_{i,j,R}$, thus knowing all the binary variables $d_{i,j,R,n}$. We can then solve both systems of equations for $k = R$ obtaining the 8 most significant bytes of the key.

By cracking the most significant 64 bits of the key we have obtained enough information to calculate the values of v in the next round ($v_{i,j,(R-1)}$) via equation (2.7). Using the appropriate S-boxes (Equation (2.6)) we can also resolve the values of and w in the next round ($w_{i,j,(R-1)}$). Since we know $\mathrm{H}[v_{ij(R-2)}]$ from the measurements and we just derived $w_{i,j,(R-1)}$ we will have all $d_{i,j,R-1,n}$. So we can solve the systems of equations for $k = R - 1$ getting the following 8 bytes of the key.

By applying this scheme repeatedly we can successfully crack the whole key independently of its size with a quite narrow search space. Given $8R$ different bytes and a brute force search of at most 20×2^8, and with $R \leq 4$ the search space is upper bounded by $(20 \cdot 2^8) \cdot (8 \cdot 4) < 2^{18} = 262144$.

It is important to notice that the system of equations is generally overdetermined and if the Hamming values are unequivocally measured, then a compatible solution will always exist. However, there is no guarantee of the system not being underdetermined. If only 7 or fewer of the 20 equations are linearly independent, multiple solutions may be possible and the described search will return the lowest one. Nevertheless, since S-boxes are designed for diffusion and we have 20 equations, the probability of this event happening is extremely unlikely. Furthermore, the modifications that are introduced later to cope with the presence of error can easily resolve this situation, so no further elaboration is needed.

The algorithm was implemented in Python and given a Hamming traces of TwoFish smartcard implementation and the algorithm successfully recovered the key every time. Numerical results are shown in Section 5.

Algorithm 1 Guess Key Byte

Require: $\mathrm{H}[v_{i,j,(k-1)}] \qquad \forall i = 0, 1, \ldots 39$

Require: $w_{i,j,k} \qquad \forall i = 0, 1, \ldots 39$

```
1:  procedure BreakKeyByte(H[v_{i,j,(k-1)}], w_{i,j,k})
2:      for m_l = 0 to 255 do
3:          Valid ← True
4:          for i = 0 to 38 step 2 do
5:              if H[v_{i,j,(k-1)}] ≠ H[w_{i,j,k} ⊕ m_l] then
6:                  Valid ← False
7:                  break
8:          if Valid = True then
9:              break
10:     for m_{l+4} = 0 to 255 do
11:         Valid ← True
12:         for i = 1 to 39 step 2 do
13:             if H[v_{i,j,(k-1)}] ≠ H[w_{i,j,k} ⊕ m_{l+4}] then
14:                 Valid ← False
15:                 break
16:         if Valid = True then
17:             break
18: return (m_l, m_{l+4})
```

4 Attack in the presence of error

Unfortunately, smart card attacks usually involve an amount of random noise overlapped with the signal. The work of Mayer-Sommer suggests that we can use correlation measures to determine Hamming weights [MS00]. However, the key schedule has a significant amount of redundancy, so we can directly shield the algorithm from noise without statistical measures, simplifying the attack.

Adding noise in our measurements will mean that the values that we assumed we knew unequivocally have now added a degree of uncertainty. We can express the measured Hamming weight with added measurement error of $v_{i\,j\,k}$ as follows.

$$\mathrm{H}\left[v_{i\,j\,k}\right] + \epsilon \qquad \epsilon \leftarrow \mathcal{N}(0, \sigma^2) \tag{4.1}$$

Where ϵ is a random Gaussian variable with zero mean and variance σ^2. Gaussian noise has been considered because even if ϵ is not a random variable, by the central limit theorem, repeated measurements over time should render a Gaussian distribution. Zero mean should be a consequence of the tuning procedure used when measuring the values, since a calibration that rendered zero mean expected error should minimize the measured error.

4.1 Least Mean Squares

Since the quantity measured is now real, and our algorithm worked with purely integer values, a first good step would be rounding to the nearest integer. Although not ideal, this technique will correct a significant number of our measurements. Therefore, we can define a modified Hamming function as follows, where the term $\mathrm{H}[x] + \epsilon$ reflects the measure value as a whole with both the original value and the added error ϵ. The operator $\{\cdot\} : \mathbb{R} \to \mathbb{N}$ represents the nearest integer.

$$\mathrm{H}^*_\epsilon(x) = \{\mathrm{H}[x] + \epsilon\} \tag{4.2}$$

Rounding the variables to the closest integer makes the systems of equations incompatible with a extremely high probability, since they are 20×8 in size. A very common way to deal with this problem is just applying the Least Mean Squares closed form solution to get an approximation of the values. However, the systems of equations presented in Eq. (3.7), (3.8) are not linear, since they are using xors.

We can find a way of expressing the previous system as a system of linear equations. Let's start by considering that the terms $a_{ij} \oplus x_i$ can be decomposed as follows

$$a_{ij} \oplus x_i = \begin{cases} x_i & \text{if} \quad a_{ij} = 0 \\ \overline{x}_i = 1 - x_i & \text{if} \quad a_{ij} = 1 \end{cases} \tag{4.3}$$

By substituting the displayed expression in a arbitrary equation that follows the xor pattern shown in the systems of equations we get:

$$\begin{aligned} b &= \sum_{i=0}^{7} a_i \oplus x_i \\ b &= \sum_{\substack{i=0 \\ a_i=0}}^{7} x_i + \sum_{\substack{i=0 \\ a_i=1}}^{7} 1 - x_i \\ b &= \mathrm{H}[a_i] + \sum_{\substack{i=0 \\ a_i=0}}^{7} x_i + \sum_{\substack{i=0 \\ a_i=1}}^{7} -x_i \\ b - \mathrm{H}[a_i] &= \sum_{i=0}^{7} (-1)^{a_i} x_i \end{aligned} \tag{4.4}$$

We can make use of this simplification into the previous equations and get a linear system of equations, where $a_{ijkn} = -2d_{ijkn} + 1$. We use $-2x + 1$ simply as a mapping $\{0, 1\} \to \{1, -1\}$. We can show the translated formulation for a fixed j, k in Equations (4.7) and (4.8). Since we can express the system with linear equations, we can apply the least mean squares closed form solution.

$$\begin{aligned} A_{jk}\, \overline{x}_l^* &= \overline{h}_{j(k-1)} \\ \overline{x}_l^* &= (A_{jk}^\top A_{jk})^{-1} A_{jk}^\top\, \overline{h}_{j(k-1)} \end{aligned} \tag{4.5}$$

However, the vector $\overline{x}_l^*$ will be real valued so it needs to be mapped to $\{0, 1\}$, so a simple mapping $\mathbb{R} \to \{0, 1\}$ is performed.

$$x_{l,n} = \max\big(0, \min\big(1, \{x_{l,n}^*\}\big)\big) \tag{4.6}$$

Solving the equations and applying the transformation shown in Equation (4.6) gives a relatively good performance for small variances σ^2. The main problem arises from forward propagating errors. If we make a mistake predicting a byte of the key m_l all the bytes in that row $m_{l-8}, m_{l-16}, \ldots$ will also be affected. Thus if we have a probability p_1 of making a mistake solving a specific system, the probability of making at least one mistake in that round will be $p_2 = 1 - (1 - p_1)^8$ since we guess 8 bytes per round. Furthermore, the probability of not making a mistake in any round out of $R - 1$ rounds will go as $p_3 = (1 - p_2)^{R-1} = (1 - p_1)^{8(R-1)}$. Compounding these two expressions reveals the sensitivity of the algorithm to error. As the key size increases the accuracy decreases significantly.

$$\begin{cases} \mathrm{H}_\epsilon^*\Big(v_{0,j,(k-1)}\Big) - \mathrm{H}\big[w_{0,j,k}\big] = a_{0,j,k,0} \cdot x_{l,0} + a_{0,j,k,1} \cdot x_{l,1} + \ldots + a_{0,j,k,7} \cdot x_{l,7} \\ \mathrm{H}_\epsilon^*\Big(v_{2,j,(k-1)}\Big) - \mathrm{H}\big[w_{2,j,k}\big] = a_{2,j,k,0} \cdot x_{l,0} + a_{2,j,k,1} \cdot x_{l,1} + \ldots + a_{2,j,k,7} \cdot x_{l,7} \\ \mathrm{H}_\epsilon^*\Big(v_{4,j,(k-1)}\Big) - \mathrm{H}\big[w_{4,j,k}\big] = a_{4,j,k,0} \cdot x_{l,0} + a_{4,j,k,1} \cdot x_{l,1} + \ldots + a_{4,j,k,7} \cdot x_{l,7} \\ \qquad \cdots \\ \mathrm{H}_\epsilon^*\Big(v_{38,j,(k-1)}\Big) - \mathrm{H}\big[w_{38,j,k}\big] = a_{38,j,k,0} \cdot x_{l,0} + a_{38,j,k,1} \cdot x_{l,1} + \ldots + a_{38,j,R,7} \cdot x_{l,7} \end{cases} \tag{4.7}$$

$$\begin{cases} \mathrm{H}_\epsilon^*\Big(v_{1,j,(k-1)}\Big) - \mathrm{H}\big[w_{1,j,k}\big] = a_{1,j,k,0} \cdot x_{l',0} + a_{1,j,k,1} \cdot x_{l',1} + \ldots + a_{1,j,k,7} \cdot x_{l',7} \\ \mathrm{H}_\epsilon^*\Big(v_{3,j,(k-1)}\Big) - \mathrm{H}\big[w_{3,j,k}\big] = a_{3,j,k,0} \cdot x_{l',0} + a_{3,j,k,1} \cdot x_{l',1} + \ldots + a_{3,j,k,7} \cdot x_{l',7} \\ \mathrm{H}_\epsilon^*\Big(v_{5,j,(k-1)}\Big) - \mathrm{H}\big[w_{5,j,k}\big] = a_{5,j,k,0} \cdot x_{l',0} + a_{5,j,k,1} \cdot x_{l',1} + \ldots + a_{5,j,k,7} \cdot x_{l',7} \\ \qquad \cdots \\ \mathrm{H}_\epsilon^*\Big(v_{39,j,(k-1)}\Big) - \mathrm{H}\big[w_{39,j,k}\big] = a_{39,j,k,0} \cdot x_{l',0} + a_{39,j,k,1} \cdot x_{l',1} + \ldots + a_{39,j,k,7} \cdot x_{l',7} \end{cases} \tag{4.8}$$

4.2 Minimizing Hamming difference

In order to circumvent the problem outlined in the previous section, we can use the high redundancy built into the system to correct the mistakes made by the least mean squares approximation.

When we make a mistake solving for m_l it is because one or more of the bits $x_{l,n}$ are incorrect. The most common scenario is one incorrect bit, then two incorrect bits and so on and so forth. Thus, we would like to try to correct the possible the errors in order of increasing complexity.

We can solve this issue by ordering the set of integers by Hamming weight as follows

$$H_M = \{0, 1, 2, 4, 8, 16, 32, 64, 128, 3, 5, 9, \ldots, 254, 255\} \tag{4.9}$$

Given this order, we can xor these masks to toggle one or several bits of the estimated key m_l producing the set of candidate keys ordered by Hamming distance to the original estimate.

$$M_l = \{m_l \oplus h_M : \quad \forall h_M \in H_M\} \tag{4.10}$$

However we need a measure of how good is the fit of an arbitrary $m'_l \in M_l$ with respect to other candidates in the set. A good approach is to minimize the sum of Hamming distances of the predicted values $\mathrm{H}\left[w_{i,j,k} \oplus m'_l\right]$ after xoring with m'_l and the rounded measured Hamming weights $\mathrm{H}^*_\epsilon\left(v_{i,j,(k-1)}\right)$. However, this can result in overfitting the key to this particular measure by getting a mask with a unnecessary large Hamming weight just because it renders the largest value for this measure. To compensate this effect we also try to minimize the sum of Hamming distances at the output of the substitution boxes, comparing the predicted values $\mathrm{H}\left[Q_{jk}\left(w_{i,j,k} \oplus m'_l\right)\right]$ with the rounded measured values $\mathrm{H}^*_\epsilon\left(w_{i,j,(k-1)}\right)$. Until now we only needed the trace of Hamming weights of $\boldsymbol{V}$, but to implement this correction the trace of $\boldsymbol{W}$ will also be needed.

$$m^*_l = \underset{m'_l \in M_l}{\operatorname{argmin}} \left\{ \sum_{i=0}^{18} \left| \mathrm{H}\left[w_{i,j,k} \oplus m'_l\right] - \mathrm{H}^*_\epsilon\left(v_{i,j,(k-1)}\right) \right| + \sum_{i=0}^{18} \left| \mathrm{H}\left[Q_{jk}\left(w_{i,j,k} \oplus m'_l\right)\right] - \mathrm{H}^*_\epsilon\left(w_{i,j,(k-1)}\right) \right| \right\}$$

The algorithm was implemented and the accuracy of the algorithm was significantly improved under the presence of error, even with high variance.

The main issue with this approach is caused by how computationally expensive becomes optimizing this function. In order to simplify it, we can restrict these masks to have a maximum Hamming weight. For example, if we only consider the mask with weight smaller or equal to two we reduce the space of possible masks from 255 to 37. Therefore, the subset of masks with Hamming weights smaller than a threshold τ is defined as follows:

$$\widetilde{M}^\tau_l = \{m_l \oplus h_M : \quad \mathrm{H}[h_M] < \tau \quad \forall h_M \in H_M\} \tag{4.11}$$

The size of this set does not grow linearly, due to the combinatorial terms. In general increasing τ means

that we can resolve larger amounts of error, but the procedure becomes more computationally expensive as we show in the numerical results in the next section.

$$\left|\widetilde{M}_l^\tau\right| = \sum_{n=0}^{\tau} \binom{8}{n} \tag{4.12}$$

4.3 Multiple Readings

The analysis so far was done considering just a single reading of the execution of the algorithm. Nevertheless, in real life it is not far-fetched to obtain multiple readings that use the same secret key. In general, the most difficult part consists on identifying which reading is associated with each secret key. However, given an algorithm as the one outlined in Section 4.2, we can easily cluster the readings using some kind of similarity measure between the sequences. Direct statistical analysis such as Pearson correlation would not take into account the order of the sequence which is of extreme relevance in this case. Therefore, we can represent the key estimates as elements of $\mathbb{Z}_{256}^N \quad N = \{16, 24, 32\}$ and compute the euclidean distance between them. The cardinality of $\mathbb{Z}_{256}^N$ is the same as $\mathbb{Z}_2^{8N}$, so it is extremely unlikely to randomly sample two different keys which have a small euclidean distance.

$$\left\|K - K'\right\|_2 = \sqrt{\sum_{i=1}^{256} (K_i - K_i')^2} \qquad K, K' \in \mathbb{Z}_{256}^N \tag{4.13}$$

In a real world scenario, our eavesdropper device would get a set of power readings. For each one of them we would get a different key estimate. We could then use a threshold radius tuned based on simulation results to easily identify key estimates with small distance values. Thus, we would get a series of clusters, one for each different key that the eavesdropper would have observed.

Moreover, as we have previously stated, the estimates for the individual bytes of the key for each round are independent of each other. Namely, a byte m_i will only depend on bytes from previous rounds $m_{i+8}, m_{i+16}, \ldots$. We can see that when making a mistake in m_i, it is going to propagate through the algorithm and affect the sequence $m_i, m_{i-8}, \ldots$. As we saw in Figure 2.3, there are 8 different sets of rows in the key schedule, so the probability of making a mistake in the same byte in two different readings is small. Consequently, the probability of making a mistake in the same byte using both Least Mean Squares plus Hamming mask correction and outputting the same result for a particular byte is extremely slim.

Therefore, if we had access to multiple power traces associated with different readings we would be able to correct for these simple mistakes by just performing a majority vote. Our approach treated the multiple readings independently and to get the final estimate of the key just took the most common value along the predicted bytes for that particular position. As long as we have at least two predicted keys that share the same byte estimate for each byte, we can be quite confident that the predicted key will be the original one.

Finally, if for a specific byte the majority vote does not output any value, namely that all byte estimates

are equally likely, we could still compare which was the error for each Least Min Squares estimate, what was the weight of the hamming mask employed in the correction and which error did the objective function (4.2) achieve. Comparing these values, we would be able to tell which byte is the most likely one.

5 Results

In this section the numerical results for acuraccies and runtimes are shown for the different algorithms outlined in Section 3. All the implementation was performed in Python 2.7.11 (64-bit) and run using a single core at 2.6 GHz.

5.1 Simple Power Attack

The results for the naive approach (the one that does not take measurement error into consideration) are shown in Table 5.1. A sample of 1000 random runs was used to test the performance of the algorithm. As we can see, the algorithm predicts correctly the key every time and the runtimes are in the domain of tenths of seconds.

Key Size	Accuracy	Avg. Runtime
128	100%	3.75 ms
192	100%	5.7 ms
256	100%	7.39 ms

Table 5.1: *Performance of the algorithm for* 1000 *random keys per key size.*

5.2 Error correction

In order to evaluate the performance of the error correction algorithm we need to compute the performance for several values of the Standard deviation σ. Furthermore, as we stated previously, varying the maximum Hamming weight τ of the masks used will result in a tradeoff between accuracy and runtime.

Therefore, a simulation of random keys was performed for values of $\sigma = 0.1, 0.2, \ldots, 2.0$ and $\tau = 0, 1, \ldots 8$. Note that for $\tau = 0$ the algorithm does not perform any sort of optimization and is equivalent to the one described in Section 4.1. A thousand random runs of the algorithm per combination of σ and τ were performed. Table 5.2 shows average accuracies and runtimes for several σ and τ. Average runtime t does not depend on σ so it is aggregated in the last row. We can also visualize the results of the simulation in Figure 5.1.

From the values shown on the Table we can draw several conclusions. It is easy to see an almost diagonal pattern, since higher values of σ decrease the accuracy and higher values of τ increase it. However, for values of $\tau > 3$ the improvement is negligible, whereas the runtime experiments a threefold increase.

The times have multiplied from our original brute force search algorithm. For $\tau = 0$ (no correction performed) the runtime is doubled and the accuracy is poor for values $\sigma > 0.6$. Furthermore, employing $\tau = 3$ as factor correction increases the runtime by a factor of 8. Nevertheless, increasing one order of magnitude the runtime almost completely shields the algorithm from Gaussian errors with standard deviation $\sigma < 1.0$, which is a more than considerable amount of noise.

$\sigma \setminus \tau$	0	1	2	3	4	5	6	7	8
0.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.2	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.4	97.5	99.8	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.6	63.2	96.7	99.7	100.0	100.0	100.0	100.0	100.0	100.0
0.8	12.4	77.3	96.9	99.8	99.9	100.0	100.0	100.0	99.8
1.0	1.4	42.1	87.2	97.3	98.1	98.0	97.7	98.7	98.4
1.2	0.0	12.8	62.4	83.2	84.1	84.9	83.7	85.5	86.2
1.4	0.0	3.1	27.0	52.6	56.1	56.0	56.3	53.5	54.8
1.6	0.0	0.4	8.4	15.9	21.7	21.3	19.6	23.4	22.3
1.8	0.0	0.0	2.3	3.6	4.2	5.4	4.9	4.8	4.7
2.0	0.0	0.0	0.1	0.5	0.6	0.6	0.3	0.1	0.7
t(ms)	7.27	11.52	25.94	55.36	91.46	121.11	148.50	150.94	142.14

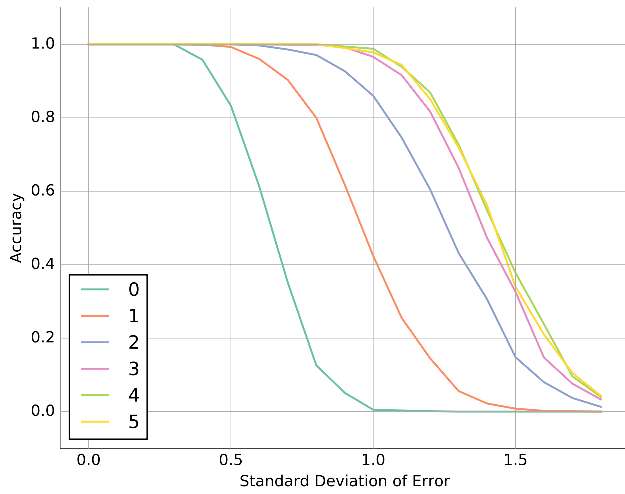
Table 5.2: *Performance of the algorithm for 128-bit key and several Standard deviations and mask sizes*

Accompanying graphics depicting the performance for different key sizes, amounts of noise and mask sizes are included in Figure 5.1. From these Figures shown below we can make a number of remarks. First, the results agree with the conclusions form the analysis of Table 5.2. Furthermore, since here can compare the performance for different key sizes we can observe that the higher the key size, the smaller the error tolerance of the algorithm. This was expected since as we commented previously, there exists a forward propagation of errors, so bigger key sizes will decrease the probability of cracking the key. In the figures we can also see that for $\tau \geq 3$ the improvement is almost negligible, so large values of τ have been omitted for clarity.

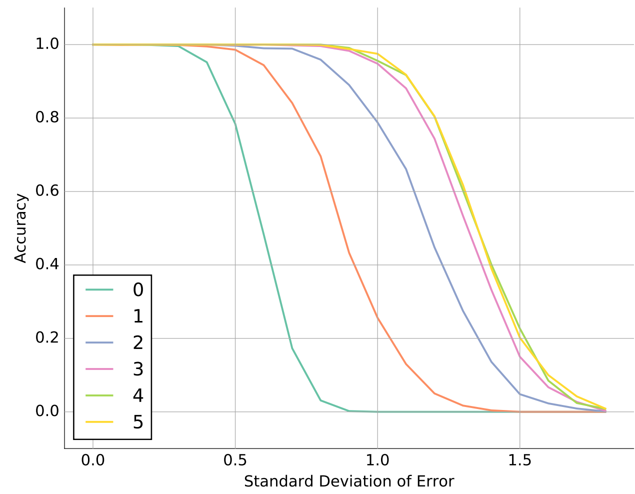
5.3 Multiple Readings

As we outlined in Section 4.3 if we observe multiple power traces from the same key we will be able to improve the error correction results for the algorithm. To analyze this behavior we performed a simulation that employed up to 5 readings per secret key. If the algorithm was able to obtain a majority vote for each byte of the key without ties it would return that key. Otherwise, it would execute a new simulation with that key up to a limit of 5 readings. We decided to use 5 readings as an upper limit since it gave significant results and it is small enough to be observed in a real world scenario.

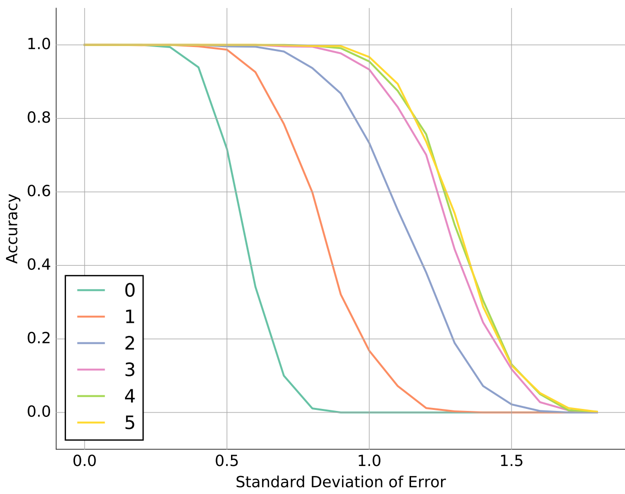
Similarly we run a thousand random runs for varying values of the parameters σ and τ we run one thousand random simulations. Results can be seen in Table 5.3 for 128-bit key. We find the same pattern as in the previous table, except that in this case error tolerances have been significantly improved. As we



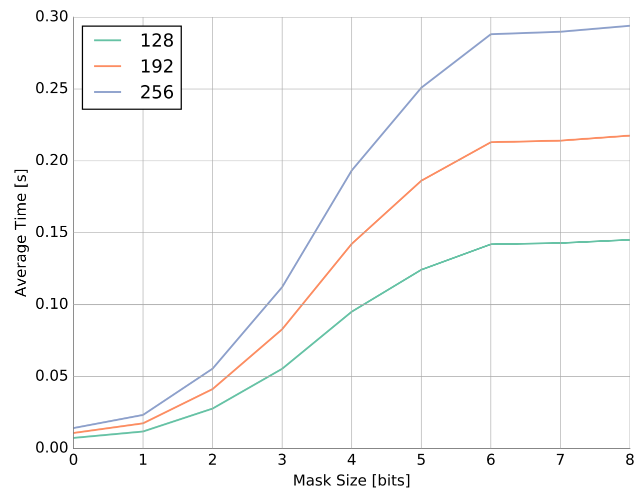
(a) *Average accuracies for 128-bit key*



(b) *Average accuracies for 192-bit key*



(c) *Average accuracies for 256-bit key*



(d) *Average Runtimes for various mask sizes*

Figure 5.1: *Results of simulation of the LMS algorithm with correction*

saw before results are almost identical for $\tau \geq 4$. Accuracies have improved for all combinations of σ, τ at the expense of times increasing by an average of a factor of three.

$\sigma \setminus \tau$	0	1	2	3	4	5	6	7	8
0.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.2	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.4	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.6	98.6	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
0.8	84.7	99.5	100.0	100.0	100.0	100.0	100.0	100.0	100.0
1.0	28.1	98.3	99.9	100.0	100.0	100.0	100.0	100.0	100.0
1.2	1.3	88.7	99.4	99.9	99.8	99.9	100.0	99.9	100.0
1.4	0.0	57.1	96.2	99.3	99.0	99.3	99.4	99.8	99.7
1.6	0.0	18.6	81.2	93.0	93.7	94.8	95.6	92.3	93.3
1.8	0.0	2.4	42.8	67.0	70.1	72.1	69.0	69.1	68.6
2.0	0.0	0.1	9.0	23.1	24.5	26.9	28.2	27.3	27.3
t(ms)	27.92	40.23	77.30	157.27	252.05	342.18	381.46	399.85	413.72

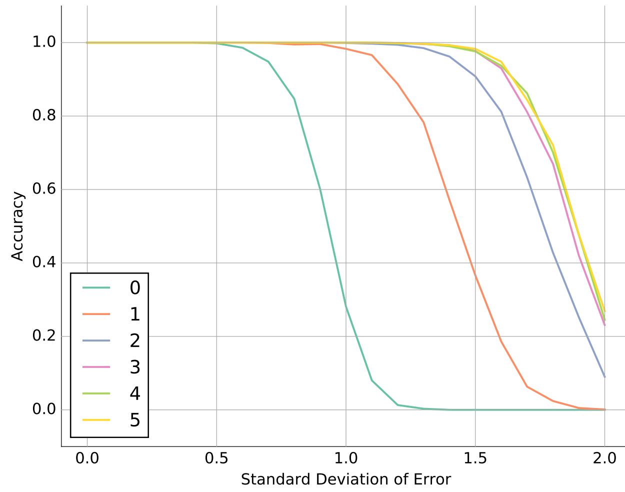
Table 5.3: *Performance of the algorithm for 128-bit key and with multiple readings per key*

Accompanying graphics are also included for this case, depicted in Figure 5.2. From the Figures we can see that for all key sizes $N = \{128, 192, 256\}$ we have improved the error tolerance from $\sigma \approx 1.0$ to $\sigma \approx 1.5$. The main difference when comparing these Figures to the ones in the previous section is the slope of the curve in the transition. Without multiple readings, the slope was fairly smooth, similar to that of a sigmoid function. Nevertheless, in the new graphics we observe that the transition curve is more abrupt, specially for 192 and 256 bit keys. This makes sense since we are introducing more layers of complexity to the error correction. The corrections make the algorithm more sensible to errors when it is unable to compensate for the amount of noise being added into the readings. Finally, for the time representation we have changed the grouping by mask size to standard deviation since now, depending on the amount of error the algorithm will do between two and five executions to determine the final key. Thus, we can see that for $\sigma \lesssim 1.0$ two executions are enough to correct all of the errors but for bigger values of the noise we start needing more executions to achieve a clear majority vote. Transition happens at $\sigma \approx 1.5$, which agrees with the accuracy results obtained in the rest of the Figures.

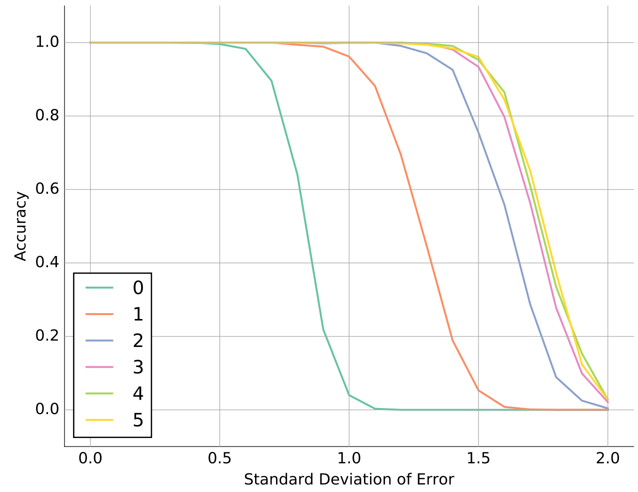
6 Discussion

A main caveat of Simple Power Attacks is that even they work in theory, they usually need a perfect Hamming trace of the execution. However, in practice we tend to experience a number of inconveniences, such as noise, uncorrelated measurements with the execution of the program and even schemes that masquerade any useful values. Therefore, a good SPA attack will not only work in theory but will also perform reasonably well under these handicaps.

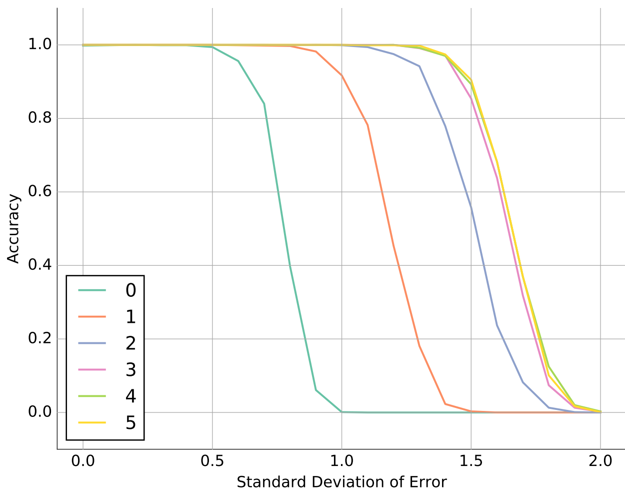
Although we did not comment it in the previous sections, obtaining a value correlated to the Hamming values has been proved to be possible to perform as shown in [MS00]. The TwoFish implementation for



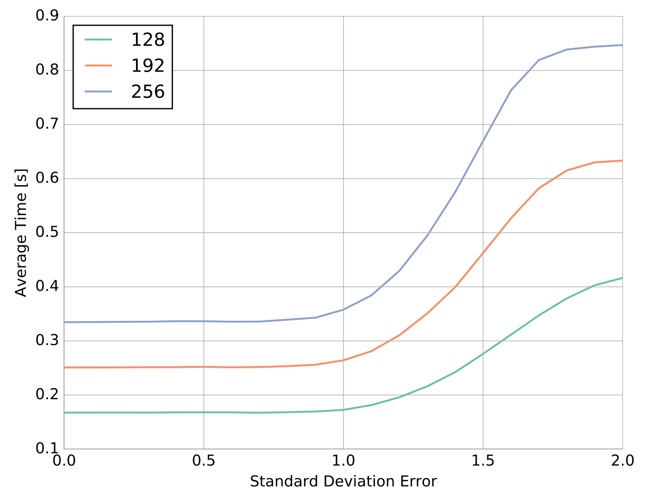
(a) *Average accuracies for 128-bit key*



(b) *Average accuracies for 192-bit key*



(c) *Average accuracies for 256-bit key*



(d) *Average Runtimes for various noise levels*

Figure 5.2: *Results of simulation with multiple readings per key*

several languages is publicly available [TSC]. Therefore, if we can run the algorithm step by step we will be able to establish the specific locations where the algorithm computes values whose Hamming weights are required for the described attack. Careful timing and some statistical analysis should resolve in the sought values when performing the attack. [MOP07]

Among the implementations are a couple of 8-bit assembly languages (Motorola 6800 and Zilog Z80) as well as more complex implementations in 32-bit and 64-bit environments. The attack works directly on both 8-bit implementations since is inherently designed for 8-bit Hamming weights. Nevertheless, examining closely the x86 Assembly and C implementations we can see that the S-box is performed as a lookup table operation. Thus, words are shifted into bytes to perform this lookup so the values needed for the attack can also be found in these implementations. TwoFish's description of the h function with such an arbitrary arrangement of S-boxes makes extremely difficult and inefficient to program an implementation that does not get the individual 8-bit values to perform the S-box substitution. Thus, as long as we can get a reliable power analysis trace, any TwoFish implementation will most likely be vulnerable to the attack presented in this paper.

Most systems vulnerable to SPAs will not give a perfect power analysis trace, so we should expect a considerable amount of noise added to our measurements. As we saw in Sections 4 and 5, modifying the algorithm to perform under error was possible. Moreover, as the results displayed, the algorithm is quite robust to Gaussian noise.

Comparing the accuracy graphics with the time plot, we can conclude that the best tradeoff accuracy-time happens at $\tau = 3$. This is probably linked to the fact that even with a high standard deviation is hard to have more than 3 bits consistently changed for the twenty bytes that we sample in the trace. Moreover, higher values of τ are not able to account for higher values of σ. Namely, for $\sigma \lesssim 1.0$, we can just increase the threshold weight τ and we will manage to get 100% accuracy. However, for values $\sigma \gtrsim 1.0$, increasing τ does not increase accuracy and in fact, for $\sigma = 2.0$ the algorithm almost always makes a mistake in at least one byte of the key.

7 Conclusion

We have shown that the TwoFish Encryption Standard is susceptible to a Simple Power Attack that is based solely in the Hamming weights of the Key Schedule Computation. The algorithm successfully obtains the value of the secret with just one run of the algorithm and in the presence of a relatively large amount of noise. Moreover, we have seen that the attack threatens not only 8-bit implementations but any TwoFish implementation since the architecture of the cipher forces the key schedule to explicitly compute the byte values that are needed for the described attack. Finally, the worst runtime of the algorithm is under one second, so the attack is not only feasible but also efficient.

7.1 Future Work

As we introduced earlier, previous research has shown that both Rijndael [VBC05] and Serpent [CTV09] key schedules were susceptible to Simple Power Attacks. Twofish brings to three the list of AES finalist

with known SPA attacks. Thus, it would be interesting to analyze the possibility of having similar SPA in the MARS and RC6 key schedules. If they were to be susceptible to similar SPAs it would set an interesting precedent for the next generation of cryptographic standards to consider. Embedded systems are becoming more and more widespread and with the introduction of frameworks such as *the Internet of Things*, confidentiality and integrity of the information acquired and transmitted by these devices will be an important concern.

Moreover, regarding the followed approach a number of considerations for further extensions can be made. The first remark is that we did not use the entirety of the key related information. As we outlined in Section 2, each round of the algorithm uses the words S_i, which are derived applying a Reed Solomon Transformation to the secret key. From the Hamming trace of the matrix calculations, a more refined search could be done to correct for errors. Furthermore, in the optimization function we only considered the input and output of a single stage of substitution. A more global approach could consider the input and output of the subsequent stages and search for the minimum of those and backtrack as necessary. Nevertheless, since the runtime would increase in combinatoric way, a smart algorithm would be needed to just consider the most likely cases and prune said search.

Finally, it would be interesting to gather real world data and perform the attack on a real system in order to consider for the statistical inference problems related to find the appropriate values in trace as well as to confirm the conclusions derived in this work.

References

[BS99] Eli Biham and Adi Shamir. Power analysis of the key scheduling of the AES candidates. In *Proceedings of the second AES Candidate Conference*, pages 115–121, 1999.

[CTV09] Kevin J. Compton, Brian Timm, and Joel VanLaven. A simple power analysis attack on the serpent key schedule. *IACR Cryptology ePrint Archive*, 2009:473, 2009.

[Kea99] Geoffrey Keating. Performance analysis of AES candidates on the 6805 cpu core. In *Proceedings of The Second AES Candidate Conference*, pages 109–114. Addison-Wesley, 1999.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[Mas94] James L. Massey. Safer k-64: A byte-oriented block-ciphering algorithm. In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 1–17. Springer-Verlag, 1994.

[MDS02] Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Comput.*, 51(5):541–552, May 2002.

[Mes01] Thomas S. Messerges. Securing the AES finalists against power analysis attacks. In *Proceedings of the 7th International Workshop on Fast Software Encryption*, FSE '00, pages 150–164, London, UK, UK, 2001. Springer-Verlag.

[MM99] Fauzan Mirza and Sean Murphy. An observation on the key schedule of twofish. In *In the second AES candidate conference, printed by the national institute of standards and technology*, pages 22–23, 1999.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[MS00] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '00, pages 78–92, London, UK, UK, 2000. Springer-Verlag.

[RHW11] S. A. M. Rizvi, Syed Zeeshan Hussain, and Neeta Wadhwa. Performance analysis of AES and twofish encryption schemes. In *Proceedings of the 2011 International Conference on Communication Systems and Network Technologies*, CSNT '11, pages 76–79, Washington, DC, USA, 2011. IEEE Computer Society.

[SKW+98] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. In *in First Advanced Encryption Standard (AES) Conference*, 1998.

[SKW+99] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. On the twofish key schedule. In *Proceedings of the Selected Areas in Cryptography*, SAC '98, pages 27–42, London, UK, UK, 1999. Springer-Verlag.

[STM10] L.M. Surhone, M.T. Timpledon, and S.F. Marseken. *Pseudo-Hadamard Transform: Pseudo-Hadamard Transform, Confusion and Diffusion, Hadamard Transform, Matrix (Mathematics), Invertible Matrix, SAFER, Communication Theory of Secrecy Systems*. Betascript Publishing, 2010.

[TSC] Twofish source code. `https://www.schneier.com/cryptography/twofish/download.html`. Accessed: 2016-03-01.

[VBC05] Joel VanLaven, Mark Brehob, and Kevin J. Compton. A computationally feasible SPA attack on AES via optimized search. In *Security and Privacy in the Age of Ubiquitous Computing: 20th International Information Security Conference (SEC 2005)*, pages 577–588, 2005.