University of Bergen
Department of informatics

Differential Power Analysis of the SKINNY Family of Block Ciphers

Author: Martin Tverråen
Supervisors: Martijn Stam, Øyvind Ytrehus

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

November, 2020

Abstract

The SKINNY family of lightweight block ciphers is well-researched in terms of standard cryptanalysis, but little has been done in the field of power analysis attacks. By sequentially dividing and conquering, univariate Differential Power Analysis attacks are performed against SKINNY. As the resulting diffusion from MixColumns introduces redundancy in terms of leakage, we introduce an alternative placement scheme for the tweak material in the related-tweakey setting to minimize leakage of the key material.

Acknowledgements

First and foremost, I would like to express my depeest gratitude towards my supervisors Martijn and Øyvind for all the mentoring, supervision, and extreme patience. Martijn; It was great fun and a true honor to dip my toes into your field of research. I would also like to thank everyone at Simula@UiB for all the support, inspiration, and Friday lunches. You have built a fantastic community. -Both for writing theses and having Fika.

Thanks to my fantastic wife, Thea, for the support and endless patience. The last months have been tiring for both of us, and I'm extremely grateful for everything you have done. I also feel obligated to thank my daughter Oline for making sure I get up every morning. Finally, I want to thank my grandfather and hero, Olaf, for being an endless inspiration and an absolute badass.

Martin Tverråen
28 October, 2020

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The most widely used cryptographic algorithms today are RSA and The Advanced Encryption Standard, AES. RSA is an asymmetric cipher based on the "factoring problem", which requires that the device on which RSA is implemented, store a set of two keys: One for encryption and one for decryption. These keys are large prime numbers, and the required storage combined with the complexity of the operation, makes RSA a bad candidate for low-resource embedded systems. AES, on the other hand, is by nature better suited for this than RSA. As it is a symmetric cipher and only operates with a single key for both encryption and decryption, the cipher's operations can be implemented more easily as a circuit. The encryption and decryption circuits are often similarly constructed, with the decryption being the inverse of encryption.

While servers, desktop computers, and smartphones are powerful enough to handle conventional cryptographic algorithms, more resource-constrained devices are not. Microcontrollers are manufactured to suit a wide array of use-cases. In small microcontrollers of, e.g., 4-bit, the limited instructions available on the device may require a large number of computing cycles if they are to execute algorithms like AES and RSA. This, in turn, could affect the device's power consumption, speed of computation, or heat-production. Typical interconnected devices like those used in sensor networks, industrial control systems, RFID, or IoT are often designed for specific applications and have to communicate wirelessly. As with all wireless communication, the transmitted messages should be encrypted to ensure confidentiality and integrity, but the application-specific bounds for resource consumption make RSA and AES bad candidates for securing these.

The messages transmitted by these highly-constrained devices are often of short length and do not necessarily require the traditional cryptosystems' full capacity. Due to this, the notion of cryptographic algorithms optimized for low-resource implementations and short message length has begun gaining momentum. The American National Institute of Standards (NIST) released the first draft of the NIST Interagency Report (NISTIR-8114[18]) in 2016 where they described an overview of NIST's work on lightweight cryptography and plans for the standardization of lightweight cryptographic algorithms. In the following years, a competition was announced in which researchers submit candidates to be considered for the next standard of lightweight cryptographic algorithms.

We generally think of one out of two approaches for attacking cryptosystems. An attacker can attack the cryptographic algorithms' design directly via cryptanalysis. The other approach is to exploit some characteristics of the device the cryptographic algorithm is implemented on, rather than the algorithm itself. While these so-called "side-channel" attacks generally exploit the device rather than the cryptographic algorithm itself, they often require physical access to this. Relevant side-channels could be, e.g., electromagnetic radiation, heat output, or power consumption. As IoT-devices become increasingly popular, gaining physical access becomes easier. In some side-channel attacks, the attacker does not necessarily require access to the specific device they want to attack in the first place, as it is enough with an identical unit which the attacker then can profile and reverse-engineer.

Physical access to the device allows for a category of side-channel analysis called Power Analysis attacks. In Power Analysis attacks, the attacker views the power consumption as a function depending on the data's size to be encrypted and the operation itself, and characterize the cryptosystem's behavior by measuring this power consumption. Power Analysis requires physical access to the device in order to measure the power consumption, but it is very effective at recovering information about the secret key. The Differential Power Analysis attack introduced by Paul Kocher in 1999 [19] reduces the search space for the key recovery of AES to 16×255, which is dramatically smaller than an exhaustive search for the full key.

AES is the most widely used cryptographic algorithm for low-resource devices and can thus be considered the standard to beat. While optimizations are done to make it more suitable for low-resource devices, lightweight cryptosystems like the "SKINNY family of lightweight block ciphers", are designed to be equally secure require less computational overhead. Therefore, it is interesting to see how SKINNY, while claiming to be more efficient

in terms of implementation complexity and resource use, is inherently more vulnerable to power analysis attacks than its predecessor.

Our Contribution. In this thesis, we investigate one candidate in NIST's "lightweight crypto standardization process": The "SKINNY family of lightweight block ciphers", to examine how resistant it is against a specific type of Power Analysis attacks called Differential Power Analysis (DPA). We investigate how the algorithm leaks and what an adversary can learn about the secret key, and whether key recovery is possible. By performing variants of a univariate DPA-attack on SKINNY, we show that the single-key mode of SKINNY is vulnerable and that the key-dependence of the different cells of the internal state leaks more for some parts of the key than others. After exploring these characteristics, we then present an alternative placement scheme for key and tweak material to minimize key material leakage.

Organization. This thesis is organized as follows: Chapters 2 to 4 is "background knowledge" about SKINNY, AES and DPA-attacks, while chapter 5 and 6 contains our contributions. In chapter 2, a general introduction to Differential Power Analysis (DPA) is given, along with a description of a common category of DPA-attacks based on maximum-likelihood hypothesis testing. In chapter 3, DPA-attacks against AES is studied in more detail. A brief description of SKINNY, its round function, and ongoing work cryptoanalysis are presented in chapter 4. In chapter 5, a DPA-attack is performed on SKINNY-64-64, and the results are discussed along with a comparison with AES. Chapter 6 contains a discussion about how the insights in chapter 5 affect SKINNY's other possible configurations and how to minimize the leakage of key material when using SKINNY with a tweak. In chapter 7, the thesis is concluded along with a description of proposed further work.

Chapter 2

Differential Power Analysis

2.1 Introduction

Differential Power Analysis (DPA) attacks as introduced by Paul Kocher, Joshua Jaffe, and Benjamin Jun in 1998[19] is a type of side-channel attacks which aims to reveal the secret keys of cryptographic devices by analyzing a large number of power traces that have been recorded during encryption or decryption of different blocks of data. Mangard, Oswald, and Popp[24, p. 120] defines Differential Power Analysis as:

"DPA attacks exploit the data dependency of the power consumption of cryptographic devices. They use many power traces to analyze the power consumption at a fixed moment of time as a function of the processed data. "

In [24, p. 120], they also described a general five-step strategy for DPA attacks:

1. *Choosing an intermediate result of the executed algorithm*
   The first step of a DPA attack is to choose a intermediate result of the algorithm. In this step the adversary wants to identify a function $f(d, k)$ where d is a known and non-constant data value (typically either the plain- or ciphertext) and k is a small part of the key. This function is often the SubBytes/s-box-step. In the case of Kocher's papers about power analysis of AES[19][20], the intermediate function is the outputs of the s-box of the first round:

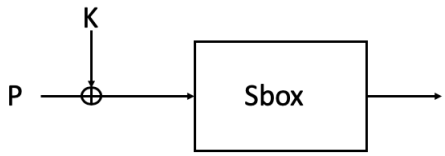


Figure 2.1: Target intermediate output of AES

2. *Measuring the power consumption*
   The next step is to measure the device's power consumption while encrypting or decrypting different data blocks. For each encryption, the adversary needs to know the vector of corresponding data values $d = (d_1, \ldots, d_D)$. d_i denotes the data of the i-th run and the power trace that corresponds to a data block d_i is denoted $t_i = (t_1, \ldots, t_T)$, where $\mathbf{T}$ denotes the length of the trace. This organizes the power traces as lists of power consumption values per timestep. We order $\mathbf{T}$ as a table, where each row $\mathbf{T}_i$ correponds to the power consumption for a sigle trace per timestep i. Each column of $\mathbf{T}$, t must correspond to the same operation. If the power traces are not aligned, calculating accurate characteristics of the power consumption of the device is difficult. A way to ensure that traces are aligned (and to avoid having to align them manually) is to set the trigger signal of the measurement tool to record the same sequence of operations for every run.

3. *Calculating hypothetical intermediate values*
   The next step is to calculate hypothetical intermediate values for every possible choice of k, called a *key hypothesis* vector $\mathbf{k} = (k_1, \ldots, k_K)$, where K denotes the total number of possible values for k. In the case of AES $K = 256$, as that is every possible value a byte can have, while in 64-bit SKINNY, $K = 16$.

   For each block d_i of data, the adversary calculates a matrix $\mathbf{V}$ of hypothetical intermediate power consumption values. Each row of $\mathbf{V}$, v_i contains a vector of all possible intermediate values for $f(d_i, k_j)$:

$$v_{i,j} = f(d_i, k_j) \quad i = 1, \ldots, D \quad j = 1, \ldots, K$$

   The DPA attack's objective is to identify which column of V that has been used in the encryption run of the device. As K contains all possible values of k, we know that the hypothetical power consumption values for the correct key hypothesis are contained

within $\mathbf{V}$. As soon as we learn which column of $\mathbf{V}$ that corresponds to the actual power consumption, we also discover the correct key hypothesis, k_{cr}

4. *Mapping intermediate Values to power consumption values*
   After computing $\mathbf{V}$, the next step of the attack is to map the list of hypothetical intermediate values $\mathbf{V}$ to a list of hypothetical power consumption values $\mathbf{H}$. The power consumption of each hypothetical intermediate value $v_{i,j}$ is simulated to obtain a hypothetical power consumption value $h_{i,j}$. How accurate the simulated power consumption matches the target device's actual power consumption depends on the adversary's knowledge of the device. The better the simulation matches the characteristics of the device, the more effective the attack is. Own Lo, William J. Buchanan and Douglas Carson addresses two such problems in DPA-attacks [23]. They observed that it was common practice in open source cryptographic libraries to combine the four steps of the AES round function into a single operation, thus making it difficult to accurately observe and determine the output of SubBytes in round 1. Another general problem with "real-life" DPA-attacks is identifying points of interest in the device's power traces, or *when*, to capture the trace. Different devices and implementations may behave differently as well, introducing even more complexity. Instrumentation of the capture device and proper trace alignment is therefore very important for the success of a DPA-attack against a hardware device.

5. *Comparing the hypothetical power consumption values with the power traces*
   The final step of the DPA-attack is to compare each column h_i of $\mathbf{H}$ to each column t_j of $\mathbf{T}$. In short this means that the attacker compares the hypothetical power consumption values with the recorded power traces at every position i, j of the matrices. A distinguisher (or refered to as a selection function) are used to compare the columns h_i and t_i, and outputs the results in a matrix $\mathbf{R}$ of size $K \times T$. The correct key hypothesis, k_{ck} are then revealed. In correlation power analysis, the distinguishing scores in $\mathbf{R}$ are the correlation coefficients of h_i and t_i, while for a maximum-likelihood distinguisher, the score $r_{i,j}$ in $\mathbf{R}$ could be the log-likelihoods for $\Pr(h_i = t_i)$.

2.2 Maximum Likelihood Testing

In [1] Agrawal et al. introduced an adverserial model that is based on the concept of generalized maximum-likelihood testing[17] combined with a CMOS leakage model. In this

strategy it is assumed that the adversary has L statistically independent sets of signals, $\mathbf{O}_i \quad i = 1, \ldots, L$. Assuming that there are K equally likely hypotheses, and the probability of observing the signal $\mathbf{O}$ given a key hypothesis H: $\Pr(\mathbf{O}|H)$. The maximum-likelihood hypothesis test is considered optimal and chooses the hypothesis H_k if:

$$k = \underset{1 \leq k \leq K}{argmax} \prod_{i=1}^{L} \Pr(\mathbf{O}_i|H_k) \tag{2.1}$$

The drawback of this approach is that an adversary will need to know the exact distribution of $\mathbf{O}$. However, the adversary can make an assumption that the probabilities for a given leakage follows a multivariate Gaussian distribution with mean μ_H and a covariance matrix Σ_H, as stated in [1]. With this assumption, the probability of observing a leakage vector $\mathbf{o}$ of a N-dimensional Gaussian distribution $p(\cdot|H)$ is:

$$p(\mathbf{o}|H) = \frac{1}{\sqrt{(2\pi)^n|\Sigma_H|}} \exp(-\frac{1}{2}(\mathbf{o} - \mu_H)^T)\Sigma_H^{-1}(\mathbf{o} - \mu_H)), \quad \mathbf{o} \in \mathcal{R}^n, \quad [1][8]$$

The Gaussian assumption [28][24] makes it possible to simplify the hypothesis testing considerably as it now only requires a partial characterization of the leakage distributions. In [1] Agrawal et al. also state that it is "in practice possible to obtain near-optimal results by making the right assumption about the distribution of the signals". In the paper "Template attacks" by Chari, Rao, and Rohatgi [8], such near-optimal results are obtained. This paper also introduced the notion of "Template attacks". A template attack is a profiling attack that is very similar to the maximum-likelihood testing introduced by Agrawal. Chari et al. perform a template attack on an implementation of RC4 on a chip-card, where the assumption of the noise distribution being gaussian is used.

In a template attack, the adversary uses a device identical to the one they attack and identify a small section of a sample S that often consists of very few traces that only depend on a few unknown key bits. By experimentation, the adversary builds templates consisting of the mean signal and noise distributions for every possible value of the unknown key bits of S. These templates are then used to classify this section of S and thus to narrow the choice of possible key bits. The templates are then expanded to include a larger section of S until templates are constructed for the whole trace.

In any DPA-attack, the measurement consists of two parts: a signal generated by the operation and noise that can be either intrinsically generated or ambient. The adversary has to guess both the value of k from the measured signal and the probability distribution of the noise. While the signal component of the measurement can be identical for repeated executions, it is difficult to construct a noise distribution that exactly matches the device. Therefore, an adversary's best guess is to assume that the noise follows a Gaussian distribution, and draw random samples from that.

A maximum-likelihood distinguishing function follows the principle of the template attack to some extent but does not build templates from a single trace of leakage. The adversary first examines one trace, i.e. a single row of $\mathbf{T}$, t_i, and finds the probability for every single key hypothesis k_j, where $j = 1, ..., K$ being correct given t_i. *Bayes' Theorem* is then used to calculate the probability $\Pr(k_j \mid t_i)$, based on the prior probability $\Pr(k_l)$ and the probability $\Pr(t_i \mid k_l)$ for $l = 1, \ldots, K$. [24, p. 155]:

$$\Pr(k_j \mid t_i) = \frac{\Pr(t_i|k_j) \times \Pr(k_j)}{\Sigma_{l=1}^{K}(\Pr(t_i|k_l) \times \Pr(k_l))}$$

The maximum-likelihood approach is considered optimal in the sense that the probability of error decreases as $\mathbf{T}$ grows [24, p. 156]. There is often not enough information in a single trace to reveal the correct key. This makes it difficult to distinguish k_{ck} from the other hypotheses. By extending this approach to multiple traces we can find the probability of k_j being correct given $\mathbf{T}$, $\Pr(k_j|\mathbf{T})$. Traces are assumed independently collected, and it is, therefore, possible to apply Bayes' theorem iteratively as shown in [24, p. 156]:

$$\Pr(k_j|\mathbf{T}) = \frac{(\prod_{i=1}^{D} \Pr(\mathbf{t}_i|k_j)) \times \Pr(k_j)}{\sum_{l=1}^{K}(\prod_{i=1}^{D} \Pr(\mathbf{t}_i|k_l)) \times \Pr(k_l)}$$

The probability of each k_j is updated with every new trace analyzed. As the same set of scores is updated iteratively, trends are easily identified, making this a very effective distinguisher. The two obvious drawbacks are that this still is a profiling attack and that as the probabilities get very small, numerical inconsistencies as rounding errors will occur. This can be mitigated somewhat by using log-likelihoods instead of actual likelihoods. The probabilities get very small, and replacing multiplications by additions reduces the size of the number in bytes.

2.3 Standard Univariate DPA Attacks

The hypothesis test introduced by Agrawal (see eq. 2.1) is very similar to the Bayesian distinguisher given by Mangard, Oswald and Standaert[25, Eq.2]:

$$\tilde{s} = \underset{s^*}{argmax} \prod_{i=1}^{q} \hat{\Pr}[l_i \mid m_i^{s*}] \tag{2.2}$$

Attacks using Bayes theorem target an approximated probability density function for the leakages and selected a subkey candidate with maximum likelihood. The leader $\tilde{s}$ is chosen by taking the argmax for each subkey s^* when calculating the maximum likelihoods for observing a given leakage l_i given a modeled leakage for the subkey m_i^{s*}.

In this paper, they also describe a "standard univariate DPA-attack", which follow three general steps:

1. For different plaintexts and subkey candidates, the attacker predicts some intermediate values. - For example, s-box outputs.
2. For each predicted value, model the leakage. This could, e.g., be the Hamming Weight of the predicted values.
3. For each subkey candidate, the attacker compares the modeled leakage with actual measurements where the same plaintexts and a secret subkey are used. Attacks where each modeled leakage is independently compared with a single point in the traces, are reffered to as univariate attacks.

2.4 The Hamming Weight Leakage Model

In power analysis, it is often necessary to map the processed device's data values to power consumption values. These power consumption values can be viewed as a power simulation of the device. In the context of a power analysis attack, the relative differences between the

simulated power consumption values are important, and not the absolute values themselves. Due to this fact and that the attacker often has very limited knowledge of the target device, generic power models like the Hamming-distance and Hamming-weight model are widely used (see [24, p. 39][25][7][29][12] among others.

The Hamming-distance model is commonly used to describe the power consumption of the target device's data buses and registers. The Hamming-weight model is simpler and is applicable even if the attacker has no knowledge of the target device's power consumption [24]. The Hamming Distance model requires either the preceding or succeeding value of the bus, as the power consumption that is used to change the value on a bus from v_0 to v_1 is proportional to $HD(v_0, v_1) = HW(v_0 \oplus v_1)$. The Hamming weight power model does not require this knowledge. The attacker assumes that the power consumption when processing a data value is proportional to the number of bits set to 1, thus ignoring the previous and next data values.

As the power consumption of a CMOS circuit depends on the transitions taking place rather than the processed data value, the HW-model is poorly suited to describe it. As $0 \rightarrow 1$ and $1 \rightarrow 0$ yields slightly different power consumption values ($0 \rightarrow 1$ can have a bigger power consumption.[24]), the Hamming weight is at least related to the actual power consumption by a certain degree. Mangard, Oswald, and Popp also states in [24, p.40] that

"Attackers only resort to the Hamming-weight model if the Hamming distance model cannot be applied."

In the case of DPA against a pure software implementation, there is no CMOS circuit to describe. Thus making the Hamming-weight model more relevant as the only thing an attacker *can* have any knowledge of is the processed data.

The noisy Hamming weight leakage model. We have a sensitive variable and assume that the target operation, e.g., the s-box leaks the hamming weight of that sensitive variable plus noise. The leakage samples are often observed as a fixed constant + the hamming weight of the sensitive variable + noise. Suppose a leakage sample l_i can be written as the sum of a deterministic part and a random part. The random part is independent of the deterministic part and identically distributed for all messages and subkeys. In that case, it is said to have additive noise as defined by Mangard, Oswald and Standaert [25]. This ties to the Gaussian assumption as l_i is further defined as being assumed Gaussian if it has additive noise and the random part follows a normal distribution $\mathcal{N}(\mu = 0, \sigma^2_{\mathcal{R}})$.

Chapter 3

Power Analysis attacks against AES

The advanced encryption standard (AES) is a symmetric SPN-cipher created by Vincent Rijmen and Joan Daemen[9] and is the most commonly used cryptosystem in embedded systems today. Due to this, AES is also one of the most analyzed cryptosystems in side-channel analysis. AES is also the target cryptographic algorithm in the first paper on power analysis attacks, written by Paul Kocher[19]. The "standard" divide and conquer attack introduced in this paper targets each cell of the internal state individually. The same attack can be performed against SKINNY as well. However, as the key propagation allows for more sophisticated attacks, it is interesting to compare how resistant AES and SKINNY are in regards to DPA-attacks.

Where the Rijndael cipher itself is a flexible block cipher that can operate with different key and block sizes that can be any multiple of 32 bits[9], AES is specified with fixed block/key sizes of 128, 196, and 256 bits.

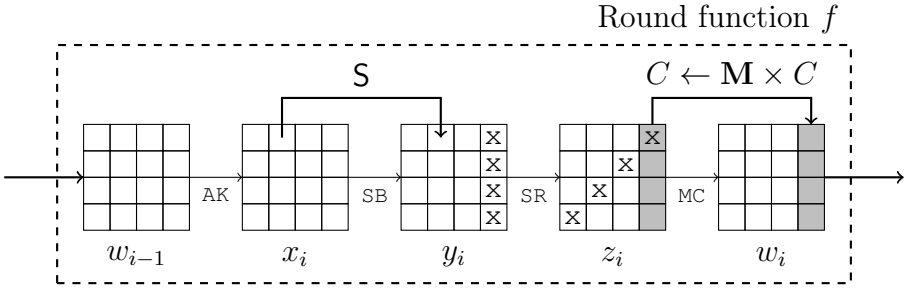


Figure 3.1: The AES round function. Image credit: Jérémy Jean[14]

The most interesing operations, in the context of DPA-attacks are AddRoundKey and SubBytes. The linear operations, ShiftRows and MixColumns are executed after SubBytes and have no impact on a DPA-attack against AES-128. When expanding to other versions of AES, e.g AES-256, these linear operations have to be executed on the intermediate internal state, to "clock" round 1 manually.

AddRoundKey The first function of encryption is "AddRoundKey" (see fig. 3.1). At the start of the first round of encryption, a "pre-whitening" operation is performed, and the cipher key is xor-ed with the plaintext. The key-whitening increases the complexity when performing an exhaustive search for the cipher key.

Round keys are then derived from the cipher key, and the intermediate values of the internal state are xor-ed with the corresponding round key of each round. More formally $IS_i = w_i \oplus k_i, \quad 0 \leq i \leq 15$ where w_i is the i-th byte of the internal state, and k_i the i-th byte of the corresponding round key.

SubBytes In SubBytes, each byte α_i gets substituted with another byte β_i according to an s-box. Every word of the internal state is one-to-one mapped to a different output byte.

3.1 Differential Power analysis of AES

In DPA attacks against AES, the most common intermediate results to target are either the first or last round, since the attacker knows the input (either the plaintext in round 1 or ciphertext in round $n-1$). This makes it possible to distinguish key and data[1]. In AES, attacking one of these rounds are sufficient to recover the full cipher key due to the "pre-whitening" in the first round.

[1]refers to the data encrypted with the key, either the plaintext or ciphertext

DPA against AES-128. In DPA attacks against AES, the most common intermediate results to target are the s-box output of either the first or last round, since the attacker knows the input (either the plaintext in round 1 or ciphertext in round $n-1$) See e.g [20]. This makes it possible to distinguish key and data. In AES, attacking one of these rounds are sufficient to recover the full cipher key due to the "pre-whitening" in the first round.

The general strategy of DPA attacks against AES is to divide and conquer. Say that the first round of encryption with AES-128 is targeted. As explained [20]: For a given power trace i, let IS_i denote the 16-byte intermediate internal state of the cipher after SubBytes. Let the n-th byte of IS state ($n \in 0, ..., 15$) be denoted by $l_{i,n}$, the first roundkey (Whitening key) denoted K and similarly $X_{i,n}$ denotes the n-th byte of plaintext. The byte $IS_{i,n}$ only depend on the single bytes $X_{i,n}$ and K_n, as shown in:

$$l_{i,n} = S[X_{i,n} \oplus K_n] \tag{3.1}$$

The objective of the DPA-attack is to test if a given candidate for K_n is correct. By dividing and conquering an attacker can solve for each K_n separatly when recovering all 16 K_n bytes of the whitening key. Recovering K requires at most 16×256 queries, as each byte K_n only has 256 possible values.

By using eq. 3.1 with a known K_n, $IS_{i,n}$ can be derived for each trace $X_{i,n}$. The adversary then uses a distinguishing function based on $IS_{i,n}$ to produce the key hypothesis.

Attacking AES-256 When attacking the initial round of AES-128, each byte of the internal state depend on an individual byte of the full cipher key. The key whitening in AES-256 spans two "initial rounds " where 128 bits of the cipher key are XOR-ed with the plaintext. Recovering the entire 256 bit cipher key of AES-256 from the encryption function therefore require that the attacker perform a DPA-attack against these two rounds, sequentially by first recovering the first 128 bits of the key, and then compute the next round's intermediate values to recover the second half[2]. The same principle translates to attacking the decryption, if only the cipher text is known.

[2]Known from "side-channel folklore". There exist very little literature about DPA-attacks against AES-256

Chapter 4

Description Of SKINNY

4.1 The SKINNY Family Of Block Ciphers

The SKINNY Family of Block Ciphers [6] is a family of tweakable block ciphers designed to compete with NSA's lightweight cryptosystem SIMON [5].

The SKINNY family's lightweight block ciphers have two block sizes: 64-bit and 128-bit, denoted by n. The internal state is viewed as a 4×4 matrix for both $n = 64$ and $n = 128$. In the case of $n = 64$, each cell is populated by a 4-bit nibble instead of an 8-bit byte. The cell of the internal state in row i and column j is denoted by $IS_{i,j}$ and is zero-indexed. If the internal state is to be viewed as a vector by concatenating rows, the internal state's cell at position i is denoted with a single subscript IS_i.

As SKINNY is a flexible tweakable block cipher (further explained in this chapter), it can be used with key lengths up to three times the block size. The following sizes of SKINNY are defined in the original specification [6], but other key sizes can be implemented by extending the Tweakey framework [13].

Block size n	$z = 1$	$z = 2$	$z = 3$
64	64	128	192
128	128	256	384

Table 4.1: The 8 configurations of SKINNY introduced in [6]

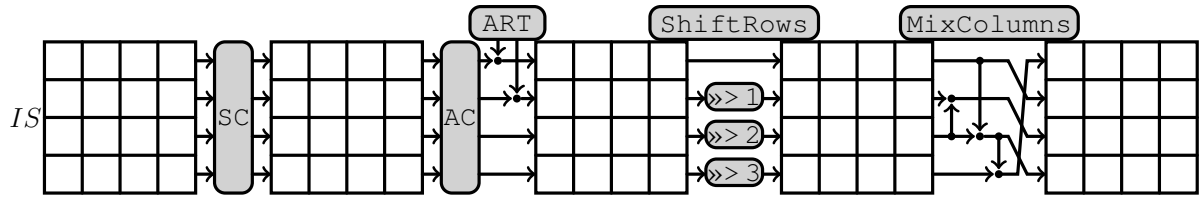


Figure 4.1: Round function of SKINNY. Image credit: Jérémy Jean with Tikz[16],

4.2 The SKINNY Round Function

SKINNY is a substitution-permutation network like AES, which applies several rounds of substitutions and permutations to produce the ciphertext. SKINNY's round function consists of the operations SubBytes, AddConstants, AddRoundTweakey, ShiftRows, and MixColums. In SubBytes, an s-box is applied to mix up the words in the internal state. After this step, round constants are generated and xor-ed with the internal state, before the top two rows of the roundTweakey is XORed with the top two rows of the internal state. ShiftRows then rotates the second, third, and fourth rows to the right by 1, 2, and 3 positions, respectively, before MixColumns is applied to shift the values column-wise. See Fig.4.1. After only six rounds, all versions of SKINNY achieve full diffusion.

Block size	n	2n	3n
64	32 rounds	36 rounds	40 rounds
128	40 rounds	48 rounds	56 rounds

Table 4.2: Number of rounds for SKINNY-n-t, with n-bit internal state and t-bit tweakey state. As presented by the authors in[6]

4.2.1 Initialization

The cipher takes as input a plaintext $p = p_0||p_1||...||p_{14}||p_{15}$, where p_i are s-bit cells where $s = n/16$. For 64-bit block SKINNY we have $s = 4$, and for 128-bit block SKINNY $s = 8$. Internal state is set by $IS_i = p_i$ for $0 \leq i \leq 15$.

4.2.2 SubCells

Depending on the block size, either a 4- or 8-bit s-box is applied to each Internal state cell.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S_4[x]$	c	6	9	0	1	a	2	b	3	8	5	d	4	e	7	f
$S_4^{-1}[x]$	3	4	6	8	c	a	1	e	9	2	5	7	0	b	d	f

Table 4.3: 4-bit Sbox S_4 for $s = 4$

4.2.3 AddConstants

The round constants are generated by a 6-bit affine LFSR, whose state is denoted $(rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)$, where rc_0 is the least significant bit. The LFSR's update function is defined as: $(rc_5||rc_4||rc_3||rc_2||rc_1||rc_0) \rightarrow (rc_4||rc_3||rc_2||rc_1||rc_0||rc_5 \oplus rc_4 \oplus 1)$.

The 6 bits are initialized to zero and updated before use in a given round. They are then arranged into a 4×4 array, depending on the size of the internal state.

$$\begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

with $c_2 = 0x2$ and

$$\begin{aligned} (c_0, c_1) &= (rc_3||rc_2||rc_1||rc_0, 0||0||rc_5||rc_4) \text{ when } s = 4 \\ (c_0, c_1) &= (0||0||0||0||rc_3||rc_2||rc_1||rc_0, 0||0||0||0||0||0||rc_5||rc_4) \text{ when } s = 8 \end{aligned} \tag{4.1}$$

As shown in the matrix, only the first column of the state is affected by the LFSR. The round constants are xor-ed with the state, respecting array positioning. The values of the $(rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)$ constants are given in the following table, encoded to byte values and rc_0 being the least significant bit, as defined by the authors in [6, p. 9]

Rounds	Constants
$1 - 16$	$01, 03, 07, 0F, 3E, 3D, 3B, 37, 2F, 1E, 3C, 39, 33, 27, 0E$
$17 - 32$	$1D, 3A, 35, 2B, 16, 2C, 18, 30, 21, 02, 05, 0B, 17, 2E, 1C, 38$
$33 - 48$	$31, 23, 06, 0D, 1B, 36, 2D, 1A, 34, 29, 12, 24, 08, 11, 22, 04$
$49 - 62$	$09, 13, 26, 0C, 19, 32, 25, 0A, 15, 2A, 14, 28, 10, 20$

TK	s	LFSR
TK2	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_2\|\|x_1\|\|x_0\|\|x_3 \oplus x_2)$
TK2	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0\|\|x_7 \oplus x_5)$
TK3	4	$(x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_2 \oplus x_3\|\|x_3\|\|x_2\|\|x_1)$
TK3	8	$(x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1\|\|x_0) \rightarrow (x_0 \oplus x_6\|\|x_7\|\|x_6\|\|x_5\|\|x_4\|\|x_3\|\|x_2\|\|x_1)$

Table 4.4: The LFSRs used to update TK2 and TK3. The TK-parameter gives the number of tweakey words and the s-parameter gives the size of each cell in bits

4.2.4 AddRoundTweakey

The first two rows of all tweakey arrays are extracted and bitwise xor-ed to the internal state of the cipher, respecting array positioning. For $i = 0, 1$ and $j = 0, 1, 2, 3$ we have:

$$\begin{aligned} IS_{i,j} &= IS_{i,j} \oplus TK1_{i,j} \text{ when } z = 1 \\ IS_{i,j} &= IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \text{ when } z = 2 \\ IS_{i,j} &= IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j} \text{ when } z = 3 \end{aligned} \tag{4.2}$$

4.2.5 The TWEAKEY schedule

The tweakey input in SKINNY allows the user to choose how much of the tweakey is used as the tweak and how much is used as the key. In the specification of SKINNY [6], three main tweakey sizes are proposed: $t = n, t = 2n, t = 3n$, where n denotes the block size. Furthermore, the size of the tweakey/block size ration is denoted $z = t/n$. It is worth mentioning that in the specification, the entire tweakey is used as key material, as the authors while providing recommendations, have not made the tweak schedule a part of the specification.

The tweakey state is generally viewed as n 4×4 matrices of cells of s bits, where s is either a nibble or a byte depending on the block size. The Tweakey ratio z determines how many tweakey words are used in a configuration. If $z = 1$, we use a single tweakey word, TK1, and from an adversarial perspective, we denote this scenario the single-key (**SK**) model. For $z = 2$, we use both TK1 and TK2, and for $z = 3$, we use TK1, TK2, and TK3. This model

is referred to as the related-tweakey scenario. The cell of the tweakey state at Row i and Col j of the z-th tweakey word is denoted $TKz_{i,j}$. This notation is extended to a single subscript vector view: $TK1_i, TK2_i, TK3_i$.

The tweakey arrays are then updated (as illustrated in Fig 4.2). A permutation P_T is applied to on the cell positions of all tweakey arrays: for all $0 \leq i \leq 15$, we set $TK1_i \leftarrow TK1_{P_T}[i]$ with $P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7]$. When $z = 2$, the same permutation is applied to TK2 as well, and when $z = 3$, the permutation is applied for both TK2 and TK3 in addition to TK1. The permutation P_T corresponds to the following reordering of the cells of the internal state, where indices are taken row-wise:

$$0, \ldots, 15 \xrightarrow{P_T} (9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7)$$

Then every cell of TK2 and TK3, given that $z > 1$ are updated individually with the LFSR given in table 4.4 (as shown in Table 3 of the original SKINNY paper[6])

4.2.6 The TWEAKEY framework

The tweakey framework [13] is a framework aiming to unify the design of tweakable block ciphers and allows the building of primitives with arbitrary key and tweak sizes. The tweakable block cipher is a cryptographic primitive, which has ana additional "tweak " input in addition to the plaintext and key. A tweak generally serves the same function as an initialization vector and introduces more randomness into the ciphertext generation. Block ciphers are deterministic in the sense that a message encrypted with a given key always will produce the same ciphertext. The introduction of a tweak, therefore introduces variability in a more elegant way than those mentioned in [21].

Instead of a tweak and key, the TWEAKEY framework takes an input that can be both tweak or key material. In the case of SKINNY, some parts of the tweakey are used as key material, while others can be used as a tweak. The TWEAKEY construction makes it possible to build an n-bit tweakable block cipher, with at-bit tweak and k-bit key. It consists of two states: A n-bit internal state s and the $(t + k)$-bit tweakey state tk. If the amount of tweak or key material is increased, the authors introduced a TK-p class of tweakable block ciphers that handles $p \times n$ tweakey material. An ordinary non-tweakable cipher with a block

and key size of n fits in TK1. A cipher with a block key and tweak size of n (or $K = 2n$) fits in TK2, and likewise for TK3. The TWEAKEY framework allows for any amount of key and/or tweak.

Instantiating SKINNY with Tweak material. When there is tweak material, TK1 is dedicated for this (As this tweakey word is never updated), and xor a bit set to "1" every round with $IS_{0,2}$.

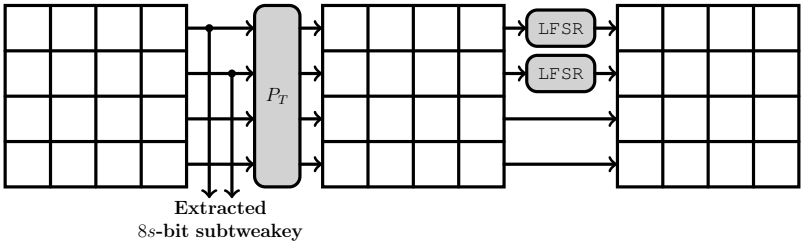


Figure 4.2: TWEAKEY schedule in SKINNY. TK1,TK2,TK3 follows a similar transformation, except that no LFSR is applied to TK1, as shown in [6, Fig 4]. Image credit: Jérémy Jean, Tikz[15]

4.2.7 ShiftRows

The second, third and fourth row of the internal state are rotated respectively 1, 2 and 3 positions to the right. More formally, a permutation P is applied to the cells of the internal state cell array: for all $0 \leq i \leq 15$, we set $IS_i \leftarrow IS_{P[i]}$ with

$$P = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7]$$

4.2.8 MixColums

Each column of the cipher state is multiplied by the following binary matrix $\mathbf{M}$:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

And as represented with row-operations:

$$IS = \begin{bmatrix} IS_0 \oplus IS_2 \oplus IS_3 \\ IS_0 \\ IS_2 \oplus IS_3 \\ IS_0 \oplus IS_2 \end{bmatrix}$$

4.3 Traditional cryptanalysis of SKINNY

The designers of SKINNY have organized several cryptanalysis competitions in order to inspire researchers to study it. Generally, the research efforts revolve around attacking a reduced-round version of SKINNY, and results so far indicate that it is fairly resistant to traditional cryptanalysis. The reduced-round attacks have little transitional value to the full version of SKINNY. Attacks based on SCA, on the other hand, can attack the full SKINNY. Even though these attacks are irrelevant to side-channel analysis, it is included for completeness as it shows how effective SCA can be in comparison.

In terms of analysis of a single key,[27] presents several impossible differential attacks against 18-round SKINNY when $z = 1$, 20 rounds for $z = 2$ and 22 rounds for $z = 3$, by extending an 11-round impossible differential distinguisher. In [10], problems with the 11-round impossible differential is examined, and it is revealed that this attack takes 2^{12} times the time, and data first reported ($2^{47.5}$ chosen plaintexts for 18-round SKINNY-64-64 and $2^{62.7}$ chosen plaintexts for 20-round SKINNY-64-128. [30] presents another impossible differential attack against 17-round SKINNY-n-n and 19-round SKINNY-n-2n[27] introduced zero-correlation linear attacks against 14-round SKINNY-64-64 and 18-round SKINNY-64-128. See Table. 4.5 for the complexity of these attacks.

Key (bits)	Attack	Rounsds	Time	Data	Memory	Source
64	zero-correlation	14	2^{62}	$2^{62.58}$	2^{64}	[27]
64	Impossible differential	17	$2^{61.8}$	$2^{59.5}$	$2^{49.6}$	[30]
64	Impossible differential	18	$2^{57.1}$	$2^{47.52}$	$2^{58.52}$	[27]
128	zero-correlation	18	2^{126}	$2^{62.68}$	2^{64}	[27]
128	Impossible differential	18	2^{116}	2^{60}	2^{112}	[27]([10])
128	Impossible differential	19	$2^{119.9}$	2^{62}	2^{110}	[30]
128	Impossible differential	20	$2^{121.08}$	$2^{47.69}$	$2^{74.69}$	[27]

Table 4.5: Complexity of traditional cryptanalytic attacks against SKINNY-n-n and SKINNY-n-2n for SKINNY-64 and SKINNY-128, as first presented by Dunkelman et al. [10, Table 1]

.

Regarding analysis of related tweakey models, [27] also presents zero-correlation attacks against 20-round SKINNY(64,64) and 23-round SKINNY(64,192) was presented in [3]. While [3, Table 1] presents the complexity of other lightweight ciphers there is a considerable overlap with 4.5 in the example of SKINNY. Among other impossible differential attacks, [22] presents an attack against 19-round SKINNY for z=1, 23-round for z=2 and 27 round for z=3, and [2] presents an additional impossible differential attack against 21-round SKINNY-64-128 as well as two attacks on 22- and 23-round SKINNY-64/128).

Chapter 5

Differential Power Analysis of SKINNY-64-64

5.1 Motivation

The most lightweight configuration of SKINNY has block and key sizes of 64 bits. While this configuration does not fulfill the required minimum key length of 112 bits as set by NIST [18][4] the small size of the processed data and lightness of computation make this configuration suitable for operating in highly constrained devices with less strict security requirements. For SKINNY with a block size of n=64, the following configurations fulfill these requirements: SKINNY-64-128 with at most 8 bits of tweak material and SKINNY-64-192 with up to 64 bits of tweak material. The wide use of SKINNY-64-64 "in the wild" is therefore not expected, but the results in this chapter DPA-attack holds directly for all versions of SKINNY where $z = 1$.

In the *AddRoundTweakey*-operation of the round function, the top rows of each tweakey word are xor-ed with the top two rows of the internal state. The number of tweakey words depends on z, the size-ratio between the block- and key-material. In the case of SKINNY's most lightweight configuration $z = 1$ and only a single tweakey-word, $TK1$ is used. Let SK_i denote the tweakey array $TK1_i^r$, where $i = \{0, 1\}$.

Selecting Intermediate Results. In the first round of SKINNY, there is no key-dependence in SubBytes. Due to the structure of the round function, it is necessary to attack round i of SKINNY to reveal the RoundTweakey of round $i-1$. By extension, this requires the adversary to attack both rounds 2 and 3 to recover the full tweakey. AddRoundTweakey is applied between SubBytes and the linear operations ShiftRows and MixColumns, hence requiring that the cipher complete a full round before the attack. This means that the following transformation has to be applied to the output, *P'* of the first round: $MixCol^{-1}(ShiftRow^{-1}(P'))$ to retrieve $AddConstants(SubBytes(P))) \oplus SK_i$, where P is the known plaintext.

While the intermediate values we target stems from SubBytes, it is, in fact, *MixColumns* that makes SKINNY an interesting target. After this operation, we have that three rows of the internal state depend on SK_0, and one row depends on SK_1, as shown in eq. 5.1. The top row of Fig. 5.1 illustrates how the top row of the RoundTweakey propagates through the linear operations. Row 0,1,3 are fully dependent on SK_0, while row 2 depend only SK_1, as shown in the bottom row of Fig. 5.1. This means that we, in theory, should have three times more leakage for SK_0 than SK_1.

It should be possible to exploit this redundancy to improve the standard divide and conquer attack, compared to a DPA-attack against AES where each cell of the internal state in the first round depends on an individual byte of the key, requiring an adversary to attack each cell of the internal state individually to recover the full key.

$$\xrightarrow{ShiftRow} \begin{bmatrix} 0 & 1 & 2 & 3 \\ 7 & 4 & 5 & 6 \\ 10 & 11 & 8 & 9 \\ 13 & 14 & 15 & 12 \end{bmatrix} \xrightarrow{Mixcol} \begin{bmatrix} 0 \oplus 10 \oplus 13 & 1 \oplus 11 \oplus 14 & 1 \oplus 8 \oplus 15 & 3 \oplus 9 \oplus 12 \\ 0 & 1 & 2 & 3 \\ 7 \oplus 10 & 4 \oplus 11 & 5 \oplus 8 & 6 \oplus 9 \\ 0 \oplus 10 & 1 \oplus 11 & 1 \oplus & 3 \oplus 9 \end{bmatrix} \tag{5.1}$$

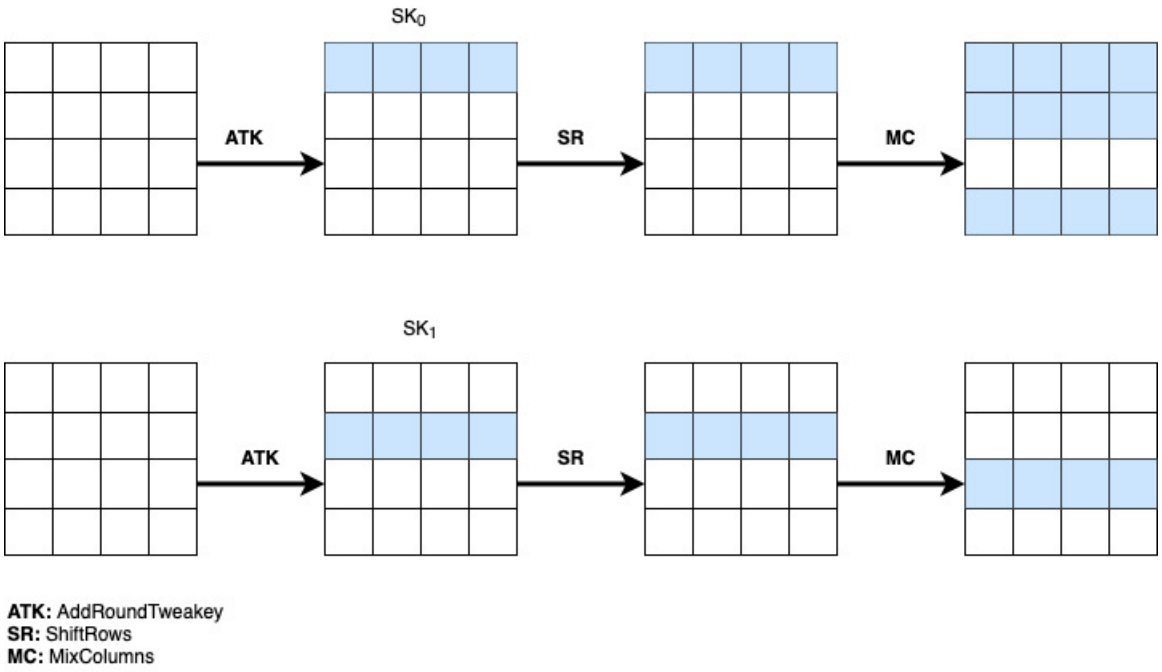


Figure 5.1: RoundTweakey dependence

After xor-ing SK_i with IS_i, the key schedule is updated by applying the permutation to the TK1-array: $P_T = [9, 15, 8, 13, 10, 14, 12, 11, 0, 1, 2, 3, 4, 5, 6, 7]$. This permutation shifts the top two rows of TK1 with the bottom two while shuffling the new top half cells. Cells are not mixed between the two halves of TK1, thus requiring an adversary to attack multiple rounds of SKINNY. In the context of a DPA attack, the first two rounds behave similarly. By dividing and conquering, the increased leakage of the top row of the roundTweakey can be exploited, such that in total, half (first and third quarter) of TK1 can be retrieved with increased confidence.

Experimental setup. In this thesis, the experiments are run against a python implementation of SKINNY created by Calvin McCoy[26]. As this is strictly software-based implementation, analysis is performed on generated power traces based on noisy Hamming-weight leakage, as there is no device to characterize. See Appendix A for an overview of the implementation of the experiments.

5.2 Methodology

The intermediate results are computed by doing partial encryptions of N plaintexts while recording the internal state (see Fig. 1). We denote IS' the internal state of the cipher post $S(P \oplus SK)$. We denote P' as the modified plaintext after $MixCol(ShiftRows(AddConstants(SubBytes(P$ has been applied, and $TK1'$ as the modified tweakey-matrix after the same set of round operations has been applied.

Algorithm 1 Computing intermediate values

```
procedure COMPUTE INTERMEDIARY(P,K)
    TK', P' ← encrypt(K, P)
    IS' ← S(p' ⊕ K')
end procedure
```

Hamming weight traces for each cell of the internal state IS' are then generated by sampling a univariate normal distribution $\mathcal{N}(\mu, \sigma^2)$ where $\mu = IS_i$. (see algorithm 2) and the amount of simulated gaussian noise is given σ.

Algorithm 2 HW trace generation

```
procedure GEN HW TRACE(IS', σ)
    for all IS' do
        for i ← 0, 16 do                          ▷ For each cell of IS'
            μ ← HW(IS'_i)
            t_i ← N(μ, σ^2)
        end for
        T ← T + t
    end for
    return T t                                    ▷ List of HW-traces for all IS'
end procedure
```

5.2.1 The Standard Attack

We denote the univariate standard attack described in [25] as the "standard attack", and in our attacks against SKINNY(64,64) a Bayesian distinguisher is used. To avoid numerical inconsistencies, we choose to take the log-likelihoods instead, as this transformation does not affect the size distribution of our scores.

Given a set of plaintext and leakage pairs (P, L), we want to find the probability of key hypothesis i being equal to the correct key k: $\Pr(k_{cr} = i \mid ((P_1, l_1) \wedge (P_2, l_2) \wedge ... \wedge (P_n, l_n)))$. It follows from Bayes theorem that this is equivalent to $\Pr((P_1, l_1) \wedge ... \wedge (P_n, l_n) \mid k = i)$. As we assume each pair of plaintexts and leakages to be collected independently, the joint probability can be calculated as $\Pr((P_1, l_1) \times ... \times (P_n, l_n) \mid k = i)$, transforming our original equation to:

$$\prod_{j}^{N} \Pr((P_j, l_j) \mid (k = i))$$

The probability of observing the pair of plaintext and leakage given that the key hypothesis is correct, $\Pr((P_j, l_j) \mid (k = i))$ can be written as the probability of the hamming weight of the intermediate value p_j xor-ed with the correct key hypothesis i being equal to the observed leakage l_j: $\Pr(HW(p_j \oplus i) = l_j)$. The equation can then be written as:

$$\prod_{j} \Pr(HW(p_j \oplus i) = l_j)$$

Since a log-likelihood estimator is used and $HW(p_j \oplus i) = v_{ji}$, the equation can be further simplified to:

$$\sum_{j} \log \Pr(HW(p_j \oplus i) = l_j) = \sum_{j} \log \Pr(v_{ji} = l_j)$$

The correct key hypothesis k_{ck} can be then be identified by taking the **Argmax** of the list of log-likelihood scores for $i = 0 \ldots 15$.

For the set of HW-traces $\mathbf{T}$, and the matrix of corresponding hypothetical power consumption values $\mathbf{V}$, we compute a score for each keyguess by first computing a probability density function for the univariate gaussian distribution $X \sim \mathcal{N}(\mu, \sigma^2)$, where $\mu = v_i$ (the hypothetical power consumption for a key hypothesis i). Then we take the sum of all scores for each keyguess over every pair of t, v: $\sum_i^n \log Pr(X = t_i)$, and take the **Argmax** of the list of log-likelihoods to identify k_{ck}.

5.2.2 Exploiting The Redundancy In The RoundTweakey

By solving for each nibble of the internal state $IS_i \quad i = \{0, \ldots, 15\}$, we now have candidates for all 16 key nibbles. Due to the redundancy caused by mixCol (See Fig .5.1), we now

Algorithm 3 Selection function for a log likelihood distinguisher

procedure STANDARD ATTACK(V, T, σ)
  for $k \leftarrow 0, 16$ **do**
    for all $t \in T, v \in V$ **do**
      $\mu \leftarrow v_k$
      $X \sim \mathcal{N}(\mu, \sigma^2)$
      $score_k \leftarrow score_k + log(\Pr(X = t))$
    end for
  end for
  return Argmax($scores$)
end procedure

have three guesses for each nibble of SK_0 while only having a single guess for each nibble of SK_1. This redundancy can be then be exploited by modifying the distinguisher. In this subsection, we explore three approaches to leverage this for SK_0.

Unanimous Attack. As all three nibbles should reveal the same key candidate, we can apply a distinguishing function to the three lists of scores and output the key candidate only if the standard attacks against the three same-key-nibble-dependent cells predict the same key hypothesis. (See algorithm 4)

Algorithm 4 Unanimous Attack

procedure UNANIMOUS-SELECTION-FUNCTION(s_0, s_1, s_2) ▷ Input: the three lists of same-nibble distinguisher scores
  if **Argmax**(s_0) == **Argmax**(s_1) == **Argmax**(s_2) **then**
    returns_0
  else
    discard
  end if
end procedure

Majority Vote Attack. Where the unanimous attack discards the key hypothesis if the key candidates produced by the standard attacks diverges, we now say that at least two identical predictions are enough, and call a majority vote to determine the most probable key candidate. (Algorithm 5)

Algorithm 5 Majority vote Attack

```
procedure MAJORITY VOTE SELECTION-FUNCTION(s_0, s_1, s_2)
    leader ← Majority Vote(Argmax(s_0), votes(s_1), Argmax(s_2))
    if size of Argmax(votes) == 1 then          ▷ If the votes are tied, discard
        discard
    else
        return Argmax(votes)
    end if
end procedure
```

The Simultaneous Attack. Where the Majority voting and anonymous attacks refined the key candidates produced by running 16 separate standard attacks, it is also possible to attack the three cells that depend on the same key-nibble simultaneously. Instead of finding the scores for $\Pr(l = i)$ individually, we instead sample a multivariate normal distribution $X \sim \mathcal{N}(\mu, \Sigma)$, where $\Sigma = \sigma^2 \times I_3$ and $\mu = E(v_1), E(v_2), E(v_3)$. We want to find the probability of observing a given leakage-vector $[L_0, L_1, L_3]$ by sampling a normal distribution were μ is a vector of corresponding hypothetical power consumption values (See Alg.6).

$$\begin{pmatrix} L_0 \\ L_1 \\ L_2 \end{pmatrix} \sim N \left[\Big(V_0 V_1 V_2 \Big), \quad \begin{pmatrix} \sigma^2 & 0 & 0 \\ 0 & \sigma^2 & 0 \\ 0 & 0 & \sigma^2 \end{pmatrix} \right]$$

Instead of producing three lists of scores (one for each cell), the simultaneous attack outputs a single list of scores and then takes the **Argmax** of that. It is worth noting that theoretically, this list of scores should be identical to taking the sum of the three lists produced by attacking the same corresponding cells with standard attacks. No information is lost between sampling a univariate distribution for each cell $X \sim \mathcal{N}(\mu_i, \sigma^2) \quad i = 0, 1, 2$ and sampling a multivariate normal distribution $X \sim \mathcal{N}(\mu, \Sigma) \quad \mu = [\mu_0, \mu_1, \mu_2]$. See table 5.1.

The simultaneous attack should be superior to the other attacks that have been presented in this section, and it should be three times as effective as the standard attack, as it has

Algorithm 6 Simultaneous selection function

procedure MULTIVARIATE SELECTION FUNCTION(T,P') ▷ T: Traces, P'
  $V = [v_0, v_1, v_2]$ ▷ For a given triple [0,1,2], of key dependent indices
  for $i \leftarrow 0, 16$ **do**
    for all $t \in T$ **do**
      $\mu \leftarrow E(v_1), E(v_2), E(v_3)$
      $\Sigma \leftarrow \sigma^2 \times I_3$
      $X \sim \mathcal{N}(\mu, \Sigma)$
      $T \leftarrow [t_{d_0}, t_{d_1}, t_{d_2}]$
      $score_i \leftarrow score_i + \log \Pr(X = T)$
    end for
  end for
  return Argmax($score$)
end procedure

three times the leakage. The majority voting and unanimous attacks offer improvements over the standarda attacks, depending on whether an attacker wants to reduce the number of traces that are required for recovering the Tweakey or trade an increase in this number with increased confidence in correctness of the key hypotheses. However, these attacks simply improve upon the scores from the standard attacks by consolidating them, so in terms of performance, they require the execution of three standard attacks nibble of TK1. They should therefore be outperformed by the simultaneous attack.

5.3 Results And Analysis: Recovery Of The First Half Of TK1

In this section, the distinguishers' efficiency is compared by investigating the probability of RoundTweakey recovery and looking at the success rates to determine the best distinguisher before attempting recovery of the second half of TK1. Figure 5.2 shows the distinguisher scores in a standard attack against a single cell as **T** grows. This trend is found in standard attacks against all the nibbles of SK. The correct key hypothesis is visually identifiable after 6 traces (See Appendix B.1 for the distinguishing scores). Thus with maximum-likelihood testing, the correct key hypothesis, k_{ck}, is identified after relatively few traces. So the standard attack is already quite effective.

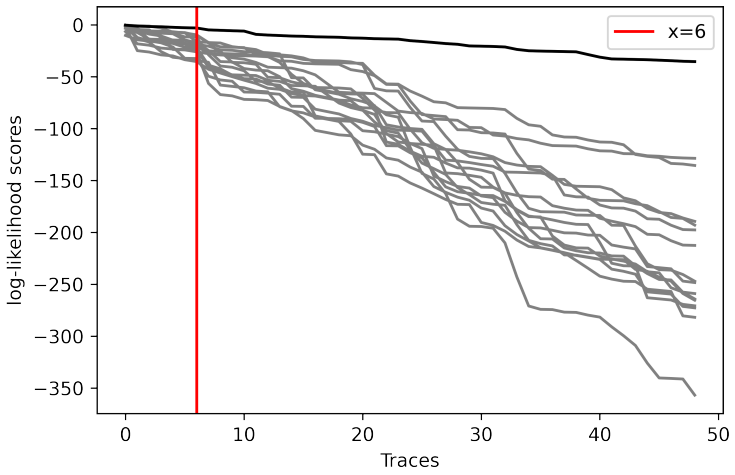
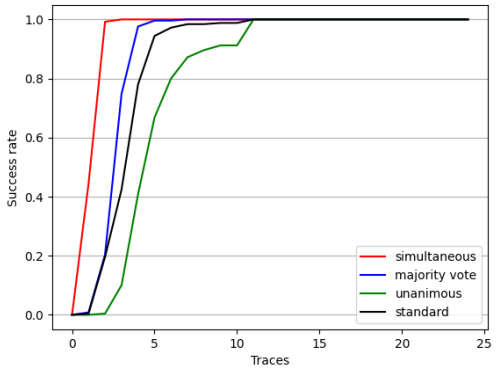


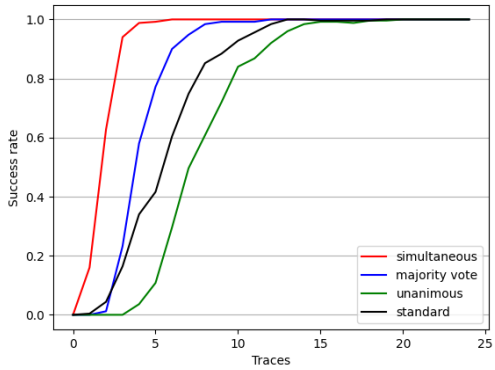
Figure 5.2: Distinguisher scores for the standard attack against $TK1_{0,0}$. The correct key hypothesis is plotted in black. $\sigma = 0.5, \mathbf{T} = 50$, X=6

To determine if the simultaneous attack is better than the other distinguishers, we then compare the success rates from 200 experiments at different noise levels $\sigma = 0.3, 0.5, 0.7, 0.9$ (See figure 5.3). As expected, the simultaneous attack is superior to the other distinguishers and requires the lowest amount of traces for the different SNRs (see also Fig. 5.4). There is also a slight advantage to be achieved by the majority voting over the standard attack. However, with the majority vote attack, we risk that the distinguisher might select the wrong candidate, and this is likely the reason that the standard attack has a higher success rate than the majority vote attack for < 4 and < 5 traces in figures 5.3c and 5.3d. Generally $\sigma = 0.5$ is used for the experiments, due to two reasons: Figures 5.35.4 show that this sigma is representative for how the different distringuishers compare to each other. In adition to this, the low noise makes the computations in the experiments faster.

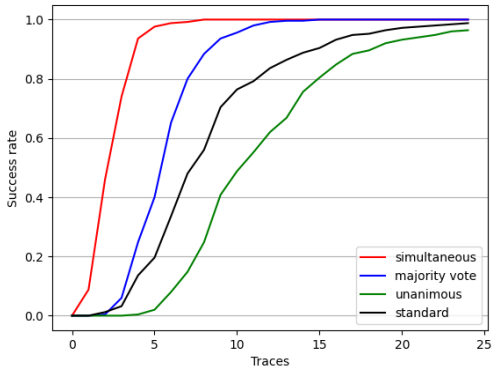
For a DPA-attack against the first nibble of $TK1_0$, we have the following dependent cells of the internal state: $[0, 4, 12]$. The first three columns in table 5.1 show the distinguisher scores of the standard attack performed sequentially against these cells. The fourth row contains the sum of the three scores for each key hypothesis K_i, and the last row contains the distinguishing scores from the simultaneous attack against the same Tweakey. The single list of scores produced by the simultaneous attack is identical to the sum of the three corresponding standard attacks against the same nibble of TK1.



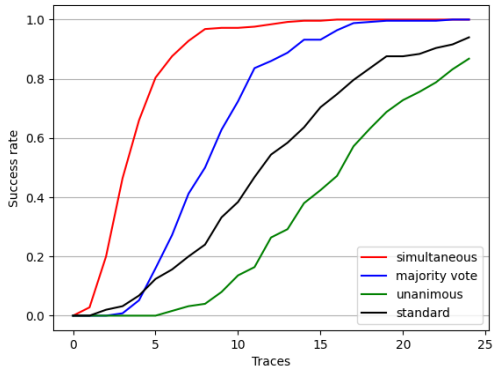
(a) $\sigma = 0.3$



(b) $\sigma = 0.5$

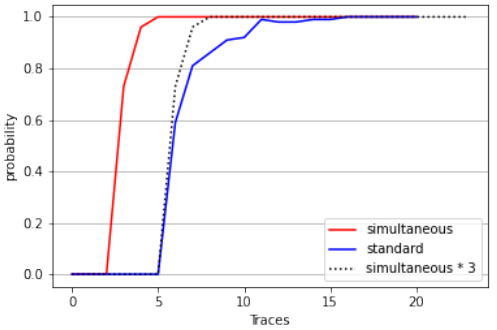


(c) $\sigma = 0.7$

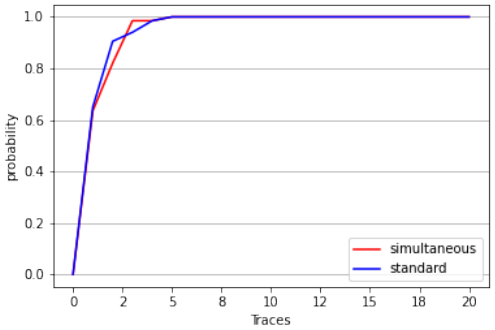


(d) $\sigma = 0.9$

Figure 5.3: Success rates of the four distinguishers for $\sigma = 0.3, 0.5, 0.7, 0.9$ taken over 200 experiments



(a) Success rates of the simultaneous and standard attack given $3x$ the traces of standard attacks against a single nibble $TK1_{0,0}$, the simultaneous attack multiplied with 3 and the standard attack, $\sigma = 0.5$



(b) Standard attack given the traces of the simultaneous

Figure 5.5: Standard and simultaneous attacks compared

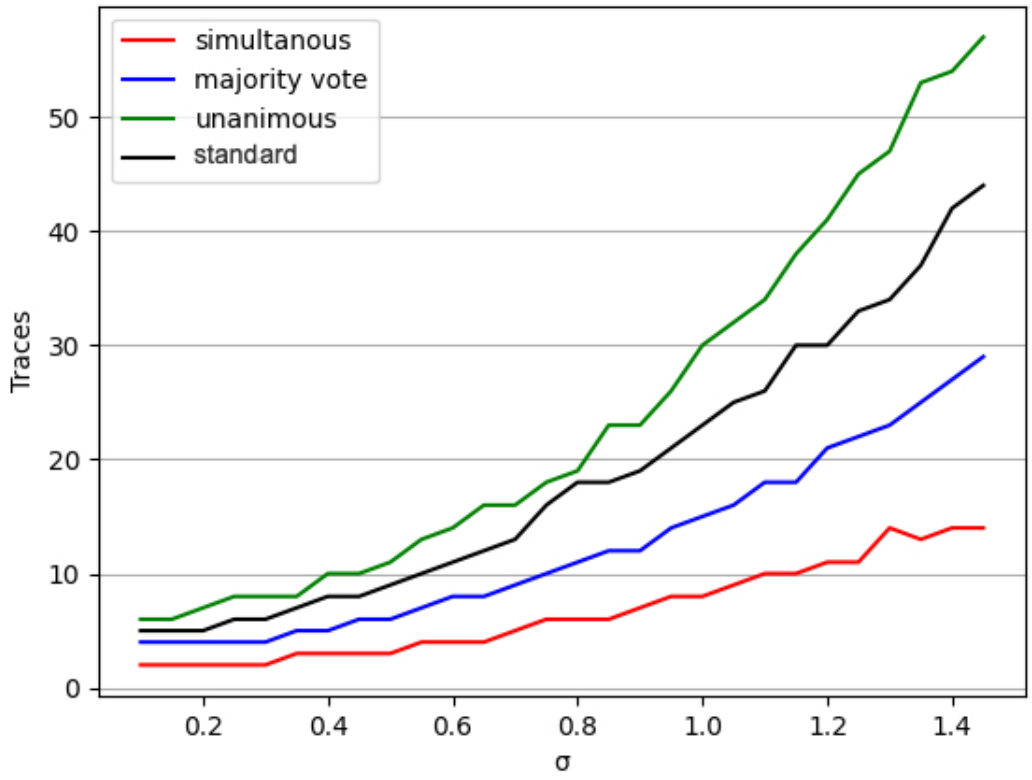


Figure 5.4: Required number of traces to achieve a success rate of 0.995 when recovering the first half of TK1, $TK1_{i,j} \quad i = \{0, 1\}, j = \{0, \ldots, 3\}$. To be updated with new label

As previously stated, we hypothesize that the success rate of the simultaneous attack for n traces corresponds to those of the standard attack given 3n traces. In Fig. 5.5a, the success rates of recovering the first half of TK1 are shown for the simultaneous and standard attacks. The simultaneous attack shifted three times (the dotted line) should be identical to the standard attack. Even though the curves are not identical, we are careful to discard the hypothesis just yet. When comparing the simultaneous attack for n traces with a standard attack given $3n$ traces as shown in Fig. 5.5b, we observe conflicting results indicating that the simultaneous attack is identical to the standard attack given $3x$ the traces in terms of success rate, and the conflicting results in Fig. 5.5a can be a result of not running enough experiments.

k_i	$TK1_0$	$TK1_4$	$TK1_{12}$	$\sum_i TK1_i$	$TK1_0$ simultaneous
0	-92.828	-111.701	-116.761	-321.292	-321.292
1	-9.582	-11.212	-15.648	-36.444	-36.444
2	-146.752	-126.559	-134.544	-407.856	-407.856
3	-75.130	-89.819	-108.715	-273.666	-273.666
4	-66.599	-29.396	-42.465	-138.461	-138.461
5	-92.935	-118.874	-166.741	-378.551	-378.551
6	-102.422	-85.251	-84.876	-272.549	-272.549
7	-81.752	-118.516	-119.373	-319.641	-319.641
8	-111.167	-110.261	-93.669	-315.098	-315.098
9	-84.280	-113.708	-85.165	-283.154	-283.154
10	-136.392	-142.407	-115.559	-394.359	-394.359
11	-49.880	-66.106	-31.771	-147.757	-147.757
12	-99.453	-121.448	-92.286	-313.188	-313.188
13	-104.494	-144.487	-136.537	-385.519	-385.519
14	-92.334	-59.479	-55.155	-206.970	-206.970
15	-96.382	-122.730	-189.185	-408.298	-408.298

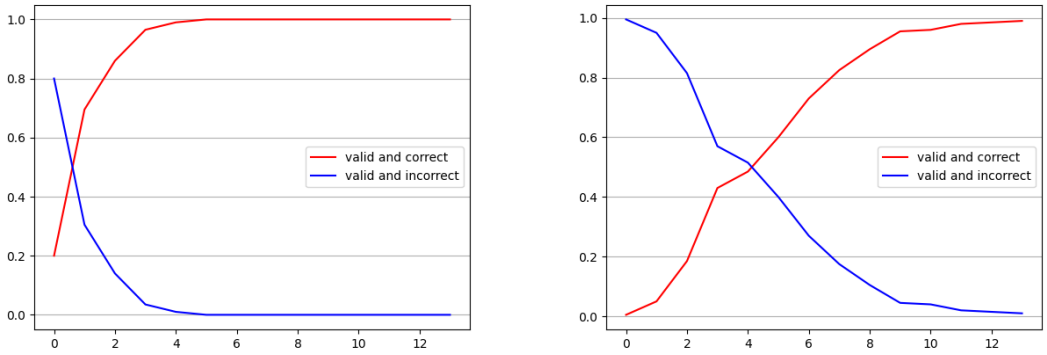
Table 5.1: Distinguisher scores for standard attacks against same round-tweakey-nibble dependent cells,
$\mathcal{K} = 1731513316006295928 \quad |T| = 20 \quad \sigma = 0.5$

5.3.1 Regarding The Confidence Of The Distinguishers

While the simultaneous attack is superior in terms of success rate, the level of confidence in a produced key candidate is another story. In the worst-case scenario where no clear leader can be identified from the distinguishing scores, both the standard and simultaneous attacks sample their candidate from a uniform distribution. An attacker learns nothing of the correct key hypothesis, as all the hypotheses are equally likely. In the worst-case scenario of the unanimous and majority vote attacks, the distinguisher discards all the guesses, and the attacker learns at least that their guess was wrong. The success- and error rates of the simultaneous and standard attacks are plotted in fig 5.6, while the success, error, and discard rates of the unanimous and majority vote attacks are plotted in Fig. 5.7.

Thillard *et al.*[29] Defined the confidence level of an attack recovering a key k_0 by application of a selection rule $\mathcal{R}$ to output a candidate subkey $\mathbf{K}^{\mathcal{R}}$ as:

$$c(\mathbf{K}^{\mathcal{R}}) = \frac{\Pr(K^{\mathcal{R}} = k_0)}{\Sigma_{k \in \mathcal{K}} \Pr(K^{\mathcal{R}} = k)}$$



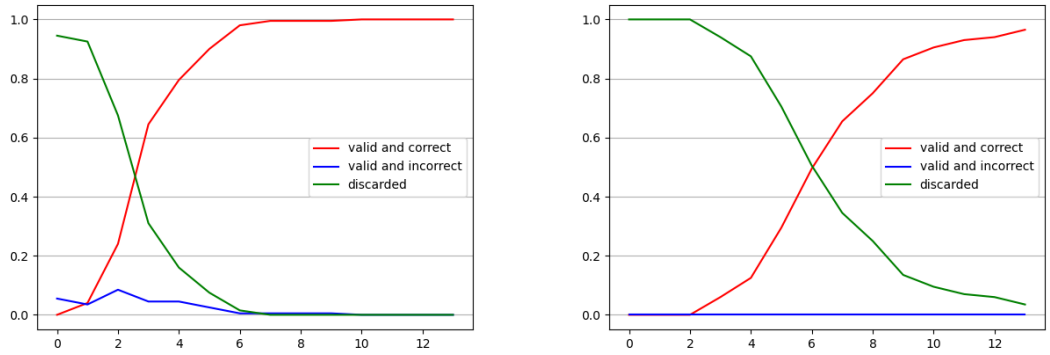
(a) simultaneous (b) individual

Figure 5.6: Correct and incorrect key candidates for simultaneous and standard attacks. $\sigma = 0.5$

In the same paper, Thillard *et al.* also remark that the confidence level associated with a rule $\mathcal{R}$ only merges with the notion of success rates if the selection rule always outputs a subkey candidate. E.g., $\mathcal{R}_0$ in [29]: "$\mathcal{R}_0$: Output, the candidate, ranked first at the end of the N-th attack." While the unanimous attack has the worst success rate of the distinguishers described in this paper, it has the highest confidence as it only outputs a key candidate when all three same-key-nibble-dependent key candidates are unanimous. The discarded guess is most likely incorrect, thus giving the adversary more confidence in a hypothesis being correct *when* it outputs a key candidate.

In 5.6, the threshold value of success/error-rates of the standard attack occurs at $n = 4$, while it occurs at $n = 1$ for the simultaneous attack, further strengthening the hypothesis of the latter has three times the leakage per trace. The same cannot be said about the majority vote and unanimous attacks. However, it is worth noting that the unanimous distinguisher output no erroneous key candidates, but at the cost of a considerably higher discard rate. Thus, it is allowing us to conclude that per the definition presented by Thillard, the unanimous attack has the highest level of confidence, while also having the lowest success rate.

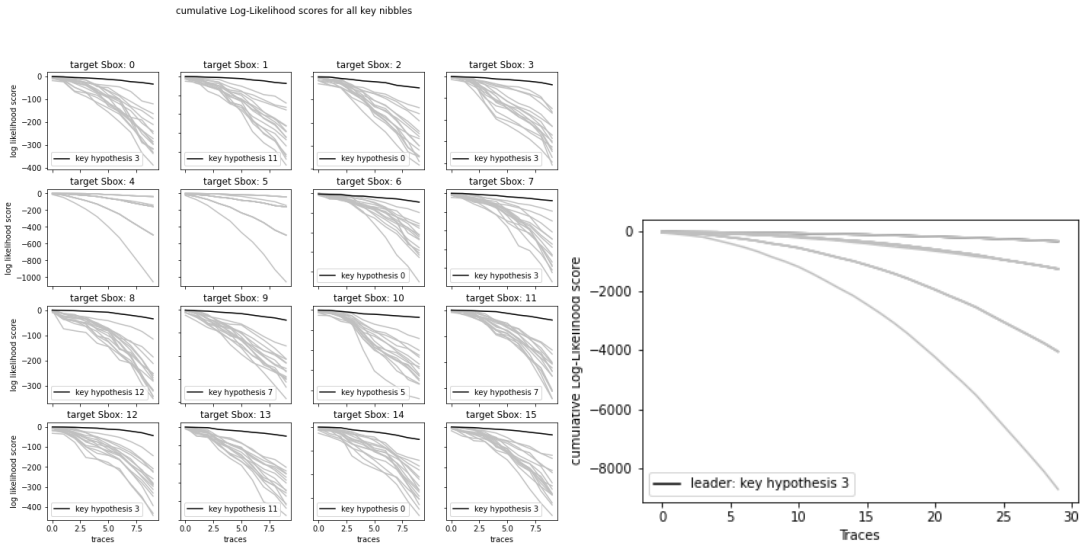
A Note About Shorter-than-n Plaintexts. When initializing the internal state, the plaintext length has to be at least the block size n, which in this case is 64 bits. As SKINNY is a lightweight block cipher, it is designed for low-resource devices sending short messages.



(a) majority vote (b) unanimous

Figure 5.7: Correct, incorrect and discarded key candidates for the discarding distinguisher attacks. $\sigma = 0.5$

If such a device running SKINNY with $n = 64$ sends a message shorter than this, the easiest way to handle this implementation-wise is to use a zero-padding from the most significant bit, until the 16 cells of the internal state are filled. This is also the case of the current implementation[26].



(a) 56-bit plaintext (b) all-zero plaintext.

Figure 5.8: Distinguishing scores for shorter-than-n-plaintexts. $\sigma = 0.5$

Fig. 5.8a shows the plots for individual attacks against the entire internal state in round 1 with a 56-bit plaintext. In the plots for s-box 4 and 5, we can observe a clustering different

from the normal distinguisher pattern in the other plots. In 5.8b, the same individual attack is executed against a random key and an all-zero vector as plaintext, and the same clustering observed in plot 4 and 5 of subfig 5.8a can be observed here. This clustering for an s-box that is only leaking on the keys, as shown in 5.8b, corresponds to the binomial coefficients of the possible hamming weights of a single nibble.

$$\binom{4}{0} = 1, \binom{4}{1} = 4, \binom{4}{2} = 6, \binom{4}{3} = 4, \binom{4}{4} = 1,$$

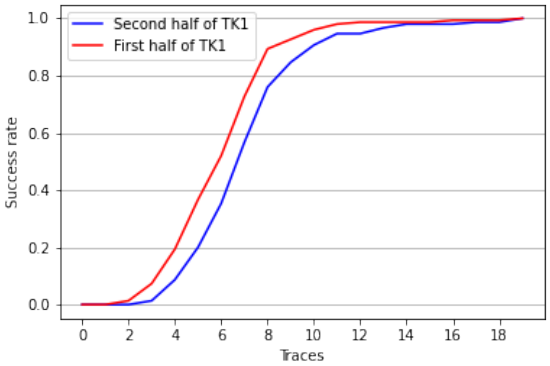
From an adversarial perspective, the only information that an attacker can learn from those plots is the hamming weight of the correct key-nibble, thus narrowing down the keyspace in an exhaustive search. DPA-attacks are generally more favorable than exhaustive searches.

5.4 Recovering The Second Half Of TK1

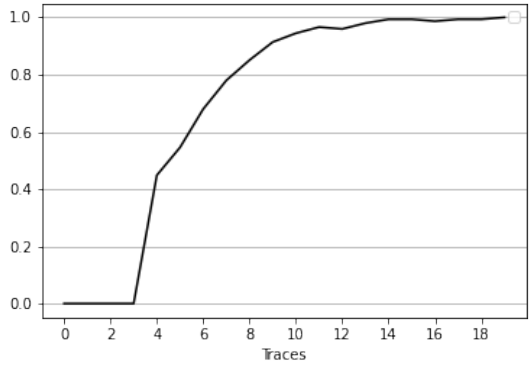
Fig. 5.9a is a plot of the success rates for a sequential divide and conquer attack. The first and third quarters of TK1 are recovered with simultaneous attacks, and the second and fourth quarter recovered with standard attacks. The success rates for recovering the second half of TK1 is slightly lower, but both plots follow the same general shape. This is expected as the same attack is performed on both rounds, except that the intermediate values used when recovering the second half are generated with the first's key hypotheses. Errors in the first round of the attack then propagate into the second round. The success rate of the sequential second round of the divide and conquer the success rate of the first, therefore, bounds attack. The probability of successfully recovering the entire TK1, given that the first half is correctly recovered, is given in Fig. 5.9b.

5.5 Comparison with AES

When attacking the first round of encryption in AES-128 we have that each byte of internal state IS_i depends only on a single byte of the cipher key K_i. This also means that it is



(a) The Success rates of recovering the $TK1_{2,3}$ after recovery of $TK1_{0,1}$



(b) Probability of successfully recovering $TK1_{2,3}$ given successfull recovery of $TK1_{0,1}$

Figure 5.9: The probability of recovering the entire TK1 successfully for $\sigma = 0.5$

possible to recover the entirety of K by divide and conquer and perform individual DPA attacks against every internal state cell. The SKINNY framework has no such whitening step, and SKINNY's Tweakey schedule makes the internal state of a round r only depend on a single half of the Tweakey words. While AddRoundKey is performed in the beginning of the encryption rounds of AES, AddRoundTweakey are located after SubBytes in SKINNY. While both cipher allows an attacker to choose the output of SubBytes as their intermediate function, SKINNY require that round $r+1$ has to be attacked in order to recover SK^r. Take for example the recovery of TK1 in SKINNY-n-n. To recover TK1 an attacker is required to attack the second and third rounds in order to recover each half of TK1. To recover the key materal leakage from the s-boxes in SubBytes of SKINNY, one has to apply the linear operations $ShiftRows^{-1}(MixColums^{-1}(P \oplus K))$, to get the correct leakage from $TK1_{i,j}$.

While attacking the initial round of AES-128 recovers the full cipher key, attacking the first round of SKINNY only recovers half of the Tweakey. Full-key recovery of SKINNY resembles the key recovery of AES-256 rather than AES-128, as both require a DPA to be executed against at least two rounds to recover both halves of the cipher key.

In both SKINNY and AES-256 it is possible to attack both encryption and decryption The poor diffusion in the Tweakey schedule of SKINNY allows an attacker to predict all future positions of a given cell of the tweakey words: $TKn^r_{i,j} \rightarrow TKn^{r+1}_{i,j}$. There is no diffusion between tweakey halves, thus enabling the tweakey halves to be recovered sequentially. This is the direct result of the update functions of the tweakey schedule being the permutation P_T and a LFSR which operates on the individual cells. In AES, on the other hand we have that to compute roundkey k_i we are required to know k_{i-1}. While the first 15 words of the key schedule are the cipher key itself, the remaining key roundkeys are not so easily predicted as in SKINNY as it, in SKiNNY is possible to combine the attacks against a round of encryption and a round of decryption.

What separates AES and SKINNY the most in the context of DPA, is the redundancy of leakage for half of SKINNY's RoundTweakeys. MixCol perform diffusion in such a way that three rows of the internal state of a given round depend on the first row of the roundTweakey, while the last row of the internal state depend on the other. When attacking AES, we divide and conquer each cell $IS_{i,j} \quad i = 0, 3, j = 0, 3$ individually, and per power trace, we only have a single source of leakage per word of the cipher key. In SKINNY, on the other hand, as in total half of the tweakey leaks thrice per trace, we can improve upon the standard divide and conquer attack, and drastically improve the probability of key recovery. The roundTweakey-dependance of MixCol combined with the poor diffusion of the tweakey, makes unmasked implementations of SKINNY-n-n theoretically more vulnerable to DPA-attacks than unmasked AES-implementations.

Chapter 6

Expanding The Attack To Other Configurations of SKINNY

While the results in chapter 5 holds for all single-key versions of SKINNY, we will now examine how the insights learned translate to the double and triple-key configurations SKINNY-n-2n and SKINNY-n-3n.

Recall from section 4.2.4 that in AddRoundTweakey the first two rows of all tweakey arrays are extracted and bitwise xor-ed to the internal state of the cipher. For $i = 0, 1$ and $j = 0, 1, 2, 3$ we have:

$$\begin{aligned} IS_{i,j} &= IS_{i,j} \oplus TK1_{i,j} \text{ when } z = 1, \\ IS_{i,j} &= IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \text{ when } z = 2 \\ IS_{i,j} &= IS_{i,j} \oplus TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j} \text{ when } z = 3 \end{aligned} \tag{6.1}$$

By disregarding the linear operations associated with each round, the target of the DPA attack against SKINNY-n-n can be simplified to $S(P \oplus TK1)$. In this case, both the input plaintexts and the leakage are known to the attacker, requiring the attacker to determine TK1. We have SKINNY-n-2n and SKINNY-n-3n with the corresponding simplifications $S(P \oplus TK1 \oplus TK2)$ and $S(P \oplus TK1 \oplus TK2)$. Attacking these configurations with DPA, an attacker only directly determines the XOR of the tweakey words, which then require more processing in order to determine TK1, TK2, and TK3.

6.1 SKINNY-n-2n

6.1.1 SKINNY-n-2n with no tweak material

In SKINNY-n-n, we can recover TK1 by attacking two rounds of encryption. In SKINNY-n-2n, we have to extend the attack and divide and conquer five encryption rounds. In the related key versions of SKINNY (SKINNY-n-2n and SKINNY-n-3n), an LFSR is applied to the top two rows of TK2 each round. We denote $\mathcal{L}$, the application of the LFSR to $TK2_{i,j} \quad i = \{0, 2\}, \quad j = \{0, 4\}$ in SKINNY-n-2n (see 4.4. Equation 6.2 shows how the tweakey words are affected by the key schedule: In each round, the permutation P_T is applied to each tweakey word, swapping the two halves with each other. Then $\mathcal{L}$ is applied to the cells of the top two rows of the tweakey words. Due to P_T, we have that R2 and R4 depend on the top halves of the tweakey words, while R3 and R5 depend on the bottom halves.

$$
\begin{aligned}
&R1 : \text{No key dependence} \\
&R2 : (TK1 \oplus TK2) \\
&R3 : (P_T(TK1) \oplus \mathcal{L}(P_T(TK2)) \\
&R4 : (P_T^2(TK1) \oplus \mathcal{L}(P_T(\mathcal{L}(P_T(TK2)))) \\
&R5 : (P_T^3(TK1) \oplus \mathcal{L}(P_T(\mathcal{L}(P_T(\mathcal{L}(P_T(TK2))))))
\end{aligned}
\tag{6.2}
$$

Due to the lack of diffusion in the key schedule, we can exploit two characteristics of this: As previously stated, there is no diffusion between the two halves of the tweakey words. The eight cells of each half only get mapped one-to-one during each round, and this mapping is identical for all the tweakey words. The permutation P_T (see section 4.2) is also identical for each round and known from the specification of SKINNY. Therefore, it is trivial to map $TK1^r_{i,j}$ to $TK1^{r-1}_{i,j}$ for $i, j = \{0, \ldots, 3\}$.

Since alternating rounds of SKINNY depend on the same halves of TK1 and TK2, we should be able to use the results of R2 and R4 to determine the top two rows of TK2. Imagine that we can recover the "round tweakeys", as idealized in Eq.6.2, and wish to reconstruct the initial Tweakey consisting of TK1 and TK2. First, we recover the top

halves of TK1 and TK2 by solving $TK1 \oplus TK2 \oplus (P_T^2(TK1) \oplus \mathcal{L}(P_T(\mathcal{L}(P_T(TK2))))$ to identify the top halves individually. As P_T substitutes the cells within each half of the tweakey words, we have to re-align these cells manually. In R4, P_T is applied twice $P_T^2 = [1, 7, 0, 5, 2, 6, 4, 3, 8, 9, 10, 11, 12, 13, 14, 15]$ By applying P_T^2 another time, the cells are placed back at their original indices, and we denote the application of the realignment-permutation P_m. This cancels out TK1 from the equation (as we get $TK1 \oplus TK1$) and we are left with $(TK2 \oplus \mathcal{L}(\mathcal{L}(TK2))))$. $\mathcal{L}$ is applied to each cell of the top two rows of the tweakey words individually and is here represented as the invertible transformation A:

$$A = \begin{bmatrix} \mathcal{L}(m_0) & \mathcal{L}(m_1) & \mathcal{L}(m_2) & \mathcal{L}(m_3) \\ \mathcal{L}(m_4) & \mathcal{L}(m_5) & \mathcal{L}(m_6) & \mathcal{L}(m_7) \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

The previous equation for recovering TK1 and TK2 can be represented as $A \times t$, where A is an invertible transformation depending on $\mathcal{L}$ and t is a nibble of TK2:

$$\begin{aligned} &(TK1 \oplus TK2) \oplus (P_T^2(TK1) \oplus (P_T(\mathcal{L}(P_T(TK2)))) \\ &P_m((TK1) \oplus TK2 \oplus (TK1) \oplus (\mathcal{L}(TK2)))) \\ &(TK2 \oplus \mathcal{L}(TK2)) = \mathcal{A} \oplus TK2 \end{aligned} \tag{6.3}$$

Retrieving $TK2_{i,j}$ is done by appling the inverse of A: $TK2_{i,j} = TK2_{i,j} \oplus A \times A^{-1}$. This is done for each $TK2_{i,j}, \quad i = \{0, 1\}, j = \{0, \ldots, 3\}$. $TK1_{i,j}$ is then revealed by taking the XOR of the recovered values of TK2 with the result from R2 ($TK1 \oplus TK2$). The bottom half of TK2 is then recovered in the same manner, except that $P_m = P_T^3$ as it has to be modified to account for the extra round. The equation which we now have to solve is $R3 \oplus R5$:

$$P_T(TK1 \oplus \mathcal{L}(P_T(TK2))) \oplus P_T^3(TK1) \oplus \mathcal{L}^2(P_T^3(TK2))$$

While we have three applications of both P_T and $\mathcal{L}$ in total for round five, only two applications of $\mathcal{L}$ affect the halves we wish to recover, thus requiring us to apply $\mathcal{L}$ only twice.

6.2 Using SKINNY-n-2n with tweak material

The specification for SKINNY that is given in section 4.2 utilizes the "classical setting" where no tweak material is used (as described by the authors in [6]). However, The Tweakey-framework allows the user to use an arbitrary amount of the tweakey as tweak material. The only constraint is that the amount of key-material must be at least n bits. When some amount of tweak material is used, the authors recommend dedicating TK1 for this, in addition to XOR-ing the second bit of $IS_{0,2}$ with "1", each round. If the user wants to use different tweak-sizes when selecting tweak-material, the authors also recommend using some cells of TK1 to encode the size of the tweak material. Dedicating TK1 to tweak material is disadvantageous as it reduces the complexity of our DPA-attacks by n bits. If SKINNY-n-2n is used with n bits of tweak, the difficulty of key recovery is reduced to roughly that of SKINNY-n-n. As TK1 is considered known, it is trivial to find TK2 from $(TK1 \oplus (TK2)$.

The difficulty of recovering each cell of the internal state. Since the key schedule treats the respective positions of the tweakey words identically (except for the per-cell LFSR in TK2), we have that $TK1_{i,j}$ and $TK2_{i,j}$ has an equal amount of leakage. So for the sake of simplicity, we now examine $TK1$ in isolation. The probability of recovering a cell of $TK1^r$ depends on its previous positions in $TK1^{r-k}$, where k is the number of previous rounds. Starting from the second round, we have four possible scenarios: Both $TK1^r$ and $TK1^{r-1}$ leak once, both $TK1^r$ and $TK1^{r-1}$ leak thrice, $TK1^r$ leaks thrice while $TK1^{r-1}$ leak once and $TK1^r$ leaks once while TK^{r-1} leaks thrice. The one-to-one mapping of cells from $TK1^r_{i,j}$ to $TK1^{r+1}_{i,j}$, makes it possible to trace the amount of leakage in terms of how many times the value of each tweakey cell leaks from a row of the internal state.

$$R2 = \begin{bmatrix} 3\delta & 3\delta & 3\delta & 3\delta \\ \delta & \delta & \delta & \delta \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Where δ denotes the ammount of times each cell of the internal state leak. Eg. In R2, we have that $\delta = 3$ for the top row, as we have leakage for these cells in three rows of the

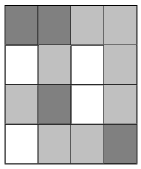


Figure 6.1: The degree of leakage of each cell $TKn_{i,j}$ after five rounds. The shades of grey correspond to the three degrees of leakage uncovered in the previous section, where darker colour means more leakage.

internal state, while the cells of the second row have $\delta = 1$, as they leak once. If we keep tracking how many times each cell $TK1_i$ leaks during the first five rounds, we get:

$$R2 = \begin{bmatrix} 3\delta & 3\delta & 3\delta & 3\delta \\ \delta & \delta & \delta & \delta \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad R3 = \begin{bmatrix} 3\delta & 3\delta & 3\delta & 3\delta \\ \delta & \delta & \delta & \delta \\ 3\delta & 3\delta & \delta & \delta \\ \delta & 3\delta & \delta & 3\delta \end{bmatrix} \quad R4 = \begin{bmatrix} 6\delta & 6\delta & 4\delta & 4\delta \\ 2\delta & 4\delta & 2\delta & 4\delta \\ 3\delta & 3\delta & \delta & \delta \\ \delta & 3\delta & \delta & 3\delta \end{bmatrix} \quad R5 = \begin{bmatrix} 6\delta & 6\delta & 4\delta & 4\delta \\ 2\delta & 4\delta & 2\delta & 4\delta \\ 4\delta & 6\delta & 2\delta & 4\delta \\ 2\delta & 4\delta & 4\delta & 6\delta \end{bmatrix}$$

After the five rounds of SKINNY required to recover TK1 and TK2 in SKINNY-n-2n, we have that $TK_{0,1,9,15} = 6\delta$, which means that these cells leak in total 6 times. $TK_{2,3,5,7,8,11,13,14} = 4\delta$, and leak four times while $TK_{4,6,10,12} = 2\delta$ and only leak twice. This means, in other words that $TK_{2,3,5,7,8,11,13,14}$ and $TK_{0,1,9,15}$ has two and three times the leakage of $TK_{4,6,10,12}$, respectively.

6.2.1 An alternative placement scheme for tweak material

The degree of leakage for each cell of the tweakey words are shown in Fig. 6.2.1. If SKINNY-n-2n is used with n bits of tweak and n bits of key, the cells that leak the most should be used for tweak material, while the cells that leak the least should be used as key material. We therefore propose an alternative placement scheme where the tweak material is placed in indices $[0, 1, 9, 15, 16, 17, 25, 31]$ and key material at indices $[4, 6, 10, 12, 20, 22, 26, 28]$. The remaining cells of the internal state have the same degree of leakage, which is somewhere between the key and tweak material, and the remaining key and tweak material can be placed freely in these.

The authors of SKINNY recommend that some cells of TK1 are used to encode how much of the tweakey is used as the tweak material. With this alternative placement scheme, it is impossible to encode such information with ease, but from an adversarial setting, where the tweak is assumed known, it provides an improvement in terms of leakage reduction.

If we, in SKINNY-n-2n, dedicate the first n cells of the tweakey as tweak material and the other n cells as key material, and count the number of times a nibble of key material and a nibble of tweak material leak, we get $\delta = 64$ for both. This means that we have an equal amount of leakage for both the key and tweak material. If we use the same assumptions from 5 and assume that the cipher leaks noisy Hamming weights, this means that the noisy Hamming weights for both the key and tweak material leaks, in total 64 times. With our scheme, we have $\delta = 80$ and $\delta = 48$ for the tweak and key material, respectively, and the amount of times key material is leaked is reduced from 64 to 48, indicating a reduction of 25%.

6.3 SKINNY-n-3n

The approach for recovering the full Tweakey of SKINNY-n-3n is very similar to SKINNY-n-2n. As in section 6.1 we have the operations on the tweakey schedule for round 1 to 5, and let $\mathcal{L}$ denote the application of the block size appropriate LFSR to $TK2_{i,j} \quad i = 0, 1, j = 0, \ldots, 3$, and M denote the application of the appropriate LFSR to $TK3_{i,j} \quad i = 0, 1, j = 0, \ldots, 3$.

$$
\begin{aligned}
&R1 : \text{No key dependence} \\
&R2 : (TK1 \oplus TK2 \oplus TK3) \\
&R3 : (P_T(TK1) \oplus \mathcal{L}(P_T(TK2) \oplus M(P_T(TK3)) \\
&R4 : P_T^2(TK1) \oplus \mathcal{L}(P_T(\mathcal{L}(P_T(TK2)))) \oplus M(P_T(M(P_T(TK3))) \\
&R5 : P_T^3(TK1) \oplus \mathcal{L}(P_T(\mathcal{L}(P_T(\mathcal{L}(P_T(TK2)))))) \oplus M(P_T(M(P_T(M(P_T(TK3))))))
\end{aligned}
\tag{6.4}
$$

The first step of the attack is to construct the permutation vector P_m to align the cells of the tweakey words. When recovering the first half of the tweakey words, $P_m = P_T^2$ is applied to R4. When recovering the second half, $P_m = P_T^3$ is applied to R5. Like with SKINNY-n-2n,

We denote $\mathcal{L}(TK2_i, j)$ as the invertible transformation A, but also denote $M(TK3_i, j)$ as the invertible transformation B.

$$\begin{aligned} &TK1_{i,j} \oplus TK2_{i,j} \oplus TK3_{i,j} \oplus TK1 \oplus \mathcal{L}(TK2_{i,j}) \oplus M(TK3_{i,j}) \\ &TK2_{i,j} \oplus TK3_{i,j} \oplus A \oplus B \end{aligned} \tag{6.5}$$

As B is constructed in the same manner as A, it is also invertible. To recover either $TK2_{i,j}$ or $TK3_{i,j}$, we then apply A^{-1} or B^{-1} to eliminate the corresponding tweakey word from the equation:

$$TK2_{i,j} \oplus TK3_{i,j} = A \times A^{-1} \times TK2_{i,j} \times B \times B^{-1} \times TK3_{i,j}$$

Like in SKINNY-n-2n, knowing $TK2_{i,j} \oplus TK3_{i,j}$ allows us to determine TK1 by substitution. Unfortunately, recovering the individual values $TK2_{i,j}$ and $TK3_{i,j}$ is not possible. Knowing $TK2_{i,j} \oplus TK3_{i,j}$ does however reduce the search space of an exhaustive search. For SKINNY-128-384, we have a complexity of $2^8 \times 2^8 = 2^{24}$ compared to a total complexity of 2^{384} if the entire tweakey are to be recovered with an exhaustive search. While in SKINNY-64-192 we have a total complexity of 2^{12} compared to 2^{192}.

It is worth noting that if SKINNY-n-2n and SKINNY-n-3n are used with n bits of tweak material, and the tweak placement scheme proposed in [6] are used, we can assume that TK1 is known, thus reducing the complexity of an attack against TK2 to the complexity of TK1. For SKINNY-n-3n, the complexity is reduced to that of SKINNY-n-2n, as TK1 is known. This reduction makes full tweakey recovery possible for SKINNY-n-3n as well.

6.4 Related Work

While section 4.3 gave a brief overview of the state of traditional cryptanalysis done on SKINNY, there is only a single paper that discusses power analysis attacks against this cipher in detail. In the article "Power Attack and Protected Implementation on Lightweight Block Cipher SKINNY"[11] written by Jing Ge *et al.* the single-key version of SKINNY is attacked with both correlation and differential power analysis to investigate how resilient

it is against these power analysis attacks and to calculate the cost of protecting a naive implementation with a masking scheme. While not including the specific strategy of their attacks or SKINNY implementation, they find that 20 traces are enough to recover the key via Correlation Power Analysis (CPA). In comparison, DPA requires more than 80 traces.

6.5 Comparison with AES

When attacking the first round of encryption in AES-128 we have that each byte of internal state IS_i depends only on a single byte of the cipher key K_i. This also means that it is possible to recover the entirety of K by divide and conquer and perform individual DPA attacks against every internal state cell. The SKINNY framework has no such whitening step, and SKINNY's Tweakey schedule makes the internal state of a round r only depend on a single half of the Tweakey words. While AddRoundKey is performed in the beginning of the encryption rounds of AES, AddRoundTweakey are located after SubBytes in SKINNY. While both ciphers allow an attacker to choose the output of SubBytes as their intermediate function, SKINNY require that round $r+1$ has to be attacked in order to recover SK^r. Take for example the recovery of TK1 in SKINNY-n-n. To recover TK1 an attacker is required to attack the second and third rounds in order to recover each half of TK1. To recover the key materal leakage from the s-boxes in SubBytes of SKINNY, one has to apply the linear operations $ShiftRows^{-1}(MixColums^{-1}(P \oplus K))$, to get the correct leakage from $TK1_{i,j}$. While attacking the initial round of AES-128 recovers the full cipher key, attacking the first round of SKINNY only recovers half of the Tweakey. Full-key recovery of SKINNY resembles the key recovery of AES-256 rather than AES-128, as both require a DPA to be executed against at least two rounds to recover both halves of the cipher key.

In both SKINNY and AES-256 it is possible to attack both the encryption and decryption operations. The poor diffusion in the Tweakey schedule of SKINNY allows an attacker to predict all future positions of a given cell of the tweakey words: $TKn^r_{i,j} \rightarrow TKn^{r+1}_{i,j}$. There is no diffusion between tweakey halves, thus enabling the tweakey halves to be recovered sequentially. This is the direct result of the update functions of the tweakey schedule being the permutation P_T and a LFSR which operates on the individual cells. In AES, on the other hand we have that to compute roundkey k_i we are required to know k_{i-1}. While the first 15 words of the key schedule are the cipher key itself, the remaining key roundkeys are

not so easily predicted as in SKINNY, where it is possible to combine the attacks against a round of encryption and a round of decryption.

What separates AES and SKINNY the most in the context of DPA, is the redundancy of leakage for half of SKINNY's RoundTweakeys. MixCol perform diffusion in such a way that three rows of the internal state of a given round depend on the first row of the roundTweakey, while the last row of the internal state depend on the other. When attacking AES, we divide and conquer each cell $IS_{i,j} \quad i = 0, 3, j = 0, 3$ individually, and per power trace, we only have a single source of leakage per word of the cipher key. In SKINNY, on the other hand, as in total half of the tweakey leaks thrice per trace, we can improve upon the standard divide and conquer attack, and drastically improve the probability of key recovery. The roundTweakey-dependance of MixCol combined with the poor diffusion of the tweakey, makes unmasked implementations of SKINNY-n-n and SKINNY-n-2n theoretically more vulnerable to DPA-attacks than unmasked AES-implementations.

Chapter 7

Summary

7.1 Conclusion

In this thesis, we have shown that the lack of diffusion between the halves of the tweakey words makes SKINNY susceptible to divide and conquer attacks. MixColumns, while providing efficient diffusion against traditional cryptanalysis, allows an attacker to improve the standard univariate attack when performing divide and conquer attacks. Even though the diffusion in SKINNY requires an attacker to attack multiple rounds, it introduces more leakage per trace than AES. The whitening step in AES makes each byte of the internal state depend on an individual byte of the cipher key. When attacking the s-boxes of SKINNY, multiple cells (bytes or nibbles) depend on a single byte of the Tweakey, thus introducing a redundancy that can be exploited. We have also demonstrated a multivariate standard attack that leverages this, and that requires a third of the traces of the univariate standard attack. This greatly improves the success rates for the affected parts of the Tweakey. A theoretical expansion of this attack was also presented against the related-key models SKINNY-n-2n and SKINNY-n-3n. We showed how to recover the entire tweakey for SKINNY-n-2n and reduce the complexity of full tweakey recovery of SKINNY-n-3n. To reduce the amount of exploitable key leakage, we have also presented an alternative placement scheme for the tweak and key material in the tweakey, thereby reducing the key recovery rate by 25% compared to the tweak placement scheme originally presented by SKINNYs authors.

7.2 Further Work

In this thesis, only a software implementation of SKINNY was analyzed. Attacking a hardware implementation of SKINNY could be interesting to see how well the simplifications in this thesis holds. Examining other power models might also be interesting, as the HW-model is strictly data-dependent. Some improvements can be made by, e.g., using a Hamming distance model instead. In this thesis, only encryption is analyzed. A possible further work could be to attack the decryption function, as $mixcol^{-1}$ also results in the roundTweakey-dependence phenomenon exploited in this thesis.

It could also be interesting to compare the sequential divide and conquer approach to extend-and-prune and key-ranking. In our experiments, we operate with "moderate" noise levels. It could also be interesting to examine a combination of the simultaneous, unanimous, and majority voting attacks to increase the confidence in some hypotheses in extremely noisy power traces. While the analysis of masked implementations of SKINNY is out of the scope of this thesis, it could also be interesting to see how efficient these variants of the univariate standard attack fare against different masking schemes.

Nomenclature

$\mathcal{N}(\mu, \sigma^2)$ The Gaussian Normal distribution

H The matrix of hypothetical power consumption values

K The list of all the

k The key hypothesis vector containing all possible k

R The matrix of size $K \times T$ of distinguisher scores

T The list of all the power traces

V The matrix of hypothetical intermediate values

HD Hamming-distance

HW Hamming weight

IS' Internal state of the cipher after $P \oplus RK$

$IS_{i,j}$ The active rows $IS_{i,j} \quad i = \{0, 1\}, j = \{0, \ldots, 3\}$ of the RoundTweakey

$IS_{i,j}$ The cell of the internal state at position i,j

K The list of all possible key values

P' The modified plaintext after round operations $MixCol(ShiftRows(AddConstants(SubBytes(P))))$ has been applied

T The length of **T**

TKn'_i The modified i-th Tweakey after operations $MixCol(ShiftRows(AddConstants(SubBytes(RK))))$ has been applied

$TKn^r_{i,j}$ The cell of TKn at position i,j in round r

TK-p A tweakey word of class "p": Eg TK1, TK2, TK3

Appendix A

Experimental setup and source code

As the experiments are performed on a software implementation with simulated hamming weights, some simplifications are made. The round function of SKINNY outputs the internal state and roundkey of the first two rounds directly for convenience (See Listing A.1). The code snippets that generate both the internal state matrices, intermediate results and power traces are found in Listings A.2 and A.3. Listing A.3 Also contains the code for the Bayesian distinguishers used in the attacks. For the profiling and comparison of the distinguishers, see Listing A.4 for the source code for the success rate. Listing A.5 is used to generate Fig. 5.4 and the code for figures 5.6, 5.7 are found in Listing A.6.

A.1 The Modified Python Implementation of SKINNY

Listing A.1: Modified SKINNY implementation

```
from __future__ import print_function
from array import array
from operator import xor

class SkinnyCipher:

    # Sbox Constants
    sbox4 = array('B', [12, 6, 9, 0, 1, 10, 2, 11, 3, 8, 5, 13, 4, 14, 7, 15])
    sbox4_inv = array('B', [3, 4, 6, 8, 12, 10, 1, 14, 9, 2, 5, 7, 0, 11, 13, 15])

    sbox8 = array('B', [0x65, 0x4c, 0x6a, 0x42, 0x4b, 0x63, 0x43, 0x6b, 0x55, 0x75, 0x5a, 0x7a, 0x53, 0x73, 0x5b, 0x7b,
                        0x35, 0x8c, 0x3a, 0x81, 0x89, 0x33, 0x80, 0x3b, 0x95, 0x25, 0x98, 0x2a, 0x90, 0x23, 0x99, 0x2b,
                        0xe5, 0xcc, 0xe8, 0xc1, 0xc9, 0xe0, 0xc0, 0xe9, 0xd5, 0xf5, 0xd8, 0xf8, 0xd0, 0xf0, 0xd9, 0xf9,
                        0xa5, 0x1c, 0xa8, 0x12, 0x1b, 0xa0, 0x13, 0xa9, 0x05, 0xb5, 0x0a, 0xb8, 0x03, 0xb0, 0x0b, 0xb9,
                        0x32, 0x88, 0x3c, 0x85, 0x8d, 0x34, 0x84, 0x3d, 0x91, 0x22, 0x9c, 0x2c, 0x94, 0x24, 0x9d, 0x2d,
```

```
                    0x62, 0x4a, 0x6c, 0x45, 0x4d, 0x64, 0x44, 0x6d, 0x52, 0x72, 0x5c, 0x7c, 0x54, 0x74, 0x5d, 0x7d,
                    0xa1, 0x1a, 0xac, 0x15, 0x1d, 0xa4, 0x14, 0xad, 0x02, 0xb1, 0x0c, 0xbc, 0x04, 0xb4, 0x0d, 0xbd,
                    0xe1, 0xc8, 0xec, 0xc5, 0xcd, 0xe4, 0xc4, 0xed, 0xd1, 0xf1, 0xdc, 0xfc, 0xd4, 0xf4, 0xdd, 0xfd,
                    0x36, 0x8e, 0x38, 0x82, 0x8b, 0x30, 0x83, 0x39, 0x96, 0x26, 0x9a, 0x28, 0x93, 0x20, 0x9b, 0x29,
                    0x66, 0x4e, 0x68, 0x41, 0x49, 0x60, 0x40, 0x69, 0x56, 0x76, 0x58, 0x78, 0x50, 0x70, 0x59, 0x79,
                    0xa6, 0x1e, 0xaa, 0x11, 0x19, 0xa3, 0x10, 0xab, 0x06, 0xb6, 0x08, 0xba, 0x00, 0xb3, 0x09, 0xbb,
                    0xe6, 0xce, 0xea, 0xc2, 0xcb, 0xe3, 0xc3, 0xeb, 0xd6, 0xf6, 0xda, 0xfa, 0xd3, 0xf3, 0xdb, 0xfb,
                    0x31, 0x8a, 0x3e, 0x86, 0x8f, 0x37, 0x87, 0x3f, 0x92, 0x21, 0x9e, 0x2e, 0x97, 0x27, 0x9f, 0x2f,
                    0x61, 0x48, 0x6e, 0x46, 0x4f, 0x67, 0x47, 0x6f, 0x51, 0x71, 0x5e, 0x7e, 0x57, 0x77, 0x5f, 0x7f,
                    0xa2, 0x18, 0xae, 0x16, 0x1f, 0xa7, 0x17, 0xaf, 0x01, 0xb2, 0x0e, 0xbe, 0x07, 0xb7, 0x0f, 0xbf,
                    0xe2, 0xca, 0xee, 0xc6, 0xcf, 0xe7, 0xc7, 0xef, 0xd2, 0xf2, 0xde, 0xfe, 0xd7, 0xf7, 0xdf, 0xff])

sbox8_inv = array('B', [0xac, 0xe8, 0x68, 0x3c, 0x6c, 0x38, 0xa8, 0xec, 0xaa, 0xae, 0x3a, 0x3e,
                        0x6a, 0x6e, 0xea, 0xee, 0xa6, 0xa3, 0x33, 0x36, 0x66, 0x63, 0xe3, 0xe6,
                        0xe1, 0xa4, 0x61, 0x34, 0x31, 0x64, 0xa1, 0xe4, 0x8d, 0xc9, 0x49, 0x1d,
                        0x4d, 0x19, 0x89, 0xcd, 0x8b, 0x8f, 0x1b, 0x1f, 0x4b, 0x4f, 0xcb, 0xcf,
                        0x85, 0xc0, 0x40, 0x15, 0x45, 0x10, 0x80, 0xc5, 0x82, 0x87, 0x12, 0x17,
                        0x42, 0x47, 0xc2, 0xc7, 0x96, 0x93, 0x03, 0x06, 0x56, 0x53, 0xd3, 0xd6,
                        0xd1, 0x94, 0x51, 0x04, 0x01, 0x54, 0x91, 0xd4, 0x9c, 0xd8, 0x58, 0x0c,
                        0x5c, 0x08, 0x98, 0xdc, 0x9a, 0x9e, 0x0a, 0x0e, 0x5a, 0x5e, 0xda, 0xde,
                        0x95, 0xd0, 0x50, 0x05, 0x55, 0x00, 0x90, 0xd5, 0x92, 0x97, 0x02, 0x07,
                        0x52, 0x57, 0xd2, 0xd7, 0x9d, 0xd9, 0x59, 0x0d, 0x5d, 0x09, 0x99, 0xdd,
                        0x9b, 0x9f, 0x0b, 0x0f, 0x5b, 0x5f, 0xdb, 0xdf, 0x16, 0x13, 0x83, 0x86,
                        0x46, 0x43, 0xc3, 0xc6, 0x41, 0x14, 0xc1, 0x84, 0x11, 0x44, 0x81, 0xc4,
                        0x1c, 0x48, 0xc8, 0x8c, 0x4c, 0x18, 0x88, 0xcc, 0x1a, 0x1e, 0x8a, 0x8e,
                        0x4a, 0x4e, 0xca, 0xce, 0x35, 0x60, 0xe0, 0xa5, 0x65, 0x30, 0xa0, 0xe5,
                        0x32, 0x37, 0xa2, 0xa7, 0x62, 0x67, 0xe2, 0xe7, 0x3d, 0x69, 0xe9, 0xad,
                        0x6d, 0x39, 0xa9, 0xed, 0x3b, 0x3f, 0xab, 0xaf, 0x6b, 0x6f, 0xeb, 0xef,
                        0x26, 0x23, 0xb3, 0xb6, 0x76, 0x73, 0xf3, 0xf6, 0x71, 0x24, 0xf1, 0xb4,
                        0x21, 0x74, 0xb1, 0xf4, 0x2c, 0x78, 0xf8, 0xbc, 0x7c, 0x28, 0xb8, 0xfc,
                        0x2a, 0x2e, 0xba, 0xbe, 0x7a, 0x7e, 0xfa, 0xfe, 0x25, 0x70, 0xf0, 0xb5,
                        0x75, 0x20, 0xb0, 0xf5, 0x22, 0x27, 0xb2, 0xb7, 0x72, 0x77, 0xf2, 0xf7,
                        0x2d, 0x79, 0xf9, 0xbd, 0x7d, 0x29, 0xb9, 0xfd, 0x2b, 0x2f, 0xbb, 0xbf,
                        0x7b, 0x7f, 0xfb, 0xff])

round_constants = array('B', [0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33,
                              0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B,
                              0x17, 0x2E, 0x1C, 0x38, 0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29,
                              0x12, 0x24, 0x08, 0x11, 0x22, 0x04, 0x09, 0x13, 0x26, 0x0c, 0x19, 0x32, 0x25, 0x0a,
                              0x15, 0x2a, 0x14, 0x28, 0x10, 0x20])

# valid cipher configurations stored:
# block_size:{key_size: number_rounds}
__valid_setups = {64: {64: 32, 128: 36, 192: 40},
                  128: {128: 40, 256: 48, 384: 56}}

__valid_modes = ['ECB', 'CTR', 'CBC', 'PCBC', 'CFB', 'OFB']

def int_to_state(self, valid_int):
    byte_state = []
    for x in range(4):
        shift_limit = self.block_size - self.row_size
        shift_val = shift_limit - (self.row_size * x)
        word = (valid_int >> shift_val) & (2 ** self.row_size - 1)
        line_array = array('B')
        for y in range(4):
            line_array.append(word >> ((self.row_size - self.s_val) - (y*self.s_val)) & self.cell_size)
        byte_state.append(line_array)
    return byte_state

def state_to_int(self, byte_array_state):
    state_int = 0
    for row in byte_array_state:
        for cell in row:
            state_int <<= self.s_val
            state_int += cell
    return state_int

def __init__(self, key, key_size=128, block_size=128, mode='ECB', init=0, counter=0):
    """
    Initialize an instance of the Skinny block cipher.
```

```
        :param key: Int representation of the encryption key
        :param key_size: Int representing the encryption key in bits
        :param block_size: Int representing the block size in bits
        :param mode: String representing which cipher block mode the object should initialize with
        :param init: IV for CTR, CBC, PCBC, CFB, and OFB modes
        :param counter: Initial Counter value for CTR mode
        :return: None
        """

        # Setup block/word size
        try:
            self.possible_setups = self.__valid_setups[block_size]
            self.block_size = block_size
            self.word_size = self.block_size >> 1
        except KeyError:
            print('Invalid block size!')
            print('Please use one of the following block sizes:', [x for x in self.__valid_setups.keys()])
            raise

        # Setup Number of Rounds, and Key Size
        try:
            self.rounds = self.possible_setups[key_size]
            self.key_size = key_size
        except KeyError:
            print('Invalid key size for selected block size!!')
            print('Please use one of the following key sizes:', [x for x in self.possible_setups.keys()])
            raise

        # Determine Cell Bit Size
        self.s_val = self.block_size >> 4

        # Calculate Tweakkey type based off ratio of key size and block size
        self.tweak_size = self.key_size // self.block_size

        self.row_size = self.s_val*4
        self.cell_size = (2**self.s_val - 1)
        self.block_mask = ((2 ** self.block_size) - 1)

        # Parse the given iv and truncate it to the block length
        try:
            iv_int = init & self.block_mask
            self.iv = self.int_to_state(iv_int)
        except (ValueError, TypeError):
            print('Invalid IV Value!')
            print('Please Provide IV as int')
            raise

        # Parse the given Counter and truncate it to the block length
        try:
            self.counter = (iv_int + counter) & self.block_mask
        except (ValueError, TypeError):
            print('Invalid Counter Value!')
            print('Please Provide Counter as int')
            raise

        # Check Cipher Mode
        try:
            position = self.__valid_modes.index(mode)
            self.mode = self.__valid_modes[position]
        except ValueError:
            print('Invalid cipher mode!')
            print('Please use one of the following block cipher modes:', self.__valid_modes)
            raise

        # Parse the given key and truncate it to the key length
        try:
            self.key = key & ((2 ** self.key_size) - 1)
        except (ValueError, TypeError):
            print('Invalid Key Value!')
            print('Please Provide Key as int')
            raise
```

```
        # Initialize key state from input key value
        key_state = []
        for z in range(self.tweak_size):
            sub_key = self.key >> ((self.key_size - self.block_size) - (z * self.block_size))
            tweakkey = self.int_to_state(sub_key)
            key_state.append(tweakkey)

        # Pre-compile key schedule
        # Generate first round key from base input key
        round_key_xor = [key_state[0][0], key_state[0][1]]
        for twky in range(1, self.tweak_size):
            round_key_xor[0] = array('B', map(xor, round_key_xor[0], key_state[twky][0]))
            round_key_xor[1] = array('B', map(xor, round_key_xor[1], key_state[twky][1]))

        self.key_schedule = [round_key_xor]

        # Generate remaining round keys
        for x in range(self.rounds):
            round_key_xor = [array('B', [0, 0, 0, 0]), array('B', [0, 0, 0, 0])]
            for y, twky in enumerate(key_state):

                # Perform Permutation Step
                modifed_key_rows = [array('B', [twky[2][1], twky[3][3], twky[2][0], twky[3][1]]),
                                    array('B', [twky[2][2], twky[3][2], twky[3][0], twky[2][3]])]

                if y > 0:  # Perform LFSR step on higher tweakey components
                    lfsr_rows = []
                    for mod_row in modifed_key_rows:
                        lfsr_row = array('B', [])
                        for cell in mod_row:
                            if self.s_val == 4:
                                if y == 1:
                                    lfsr_row.append(((cell << 1) ^ ((cell >> 3) ^ (cell >> 2) & 1)) & 0xF)
                                else:
                                    lfsr_row.append(((cell >> 1) ^ ((cell << 3) ^ cell & 0x8)) & 0xF)
                            else:
                                if y == 1:
                                    lfsr_row.append(((cell << 1) ^ ((cell >> 7) ^ (cell >> 5) & 1)) & 0xFF)
                                else:
                                    lfsr_row.append(((cell >> 1) ^ ((cell << 7) ^ (cell << 1) & 0x80)) & 0xFF)

                        lfsr_rows.append(lfsr_row)
                    modifed_key_rows = lfsr_rows

                # Store updated round key data
                round_key_xor[0] = array('B', map(xor, round_key_xor[0], modifed_key_rows[0]))
                round_key_xor[1] = array('B', map(xor, round_key_xor[1], modifed_key_rows[1]))

                # Update key state
                key_state[y] = [modifed_key_rows[0],
                                modifed_key_rows[1],
                                twky[0],
                                twky[1]]

            self.key_schedule.append([round_key_xor[0], round_key_xor[1]])
            self.trace=[]
            self.intermediary_values= []
            self.intermetiate_values_round_2=[]


    def encrypt(self, plaintext):

        try:
            pt_int = plaintext & self.block_mask
            plaintext_state = self.int_to_state(pt_int)
            self.ins = plaintext_state
        except (ValueError, TypeError):
            print('Invalid Plaintext Value!')
            print('Please Provide Plaintext as int')
            raise
```

```
    # Prepare Based On Mode
    if self.mode == 'ECB':
        internal_state = plaintext_state
        internal_state = self.encrypt_function(internal_state)
        ciphertext = self.state_to_int(internal_state)
        return ciphertext

    elif self.mode == 'CTR':
        internal_state = self.int_to_state(self.counter)
        self.counter += 1
        internal_state = self.encrypt_function(internal_state)
        internal_state = [array('B', map(xor, plaintext_state[x], internal_state[x])) for x in range(4)]
        ciphertext = self.state_to_int(internal_state)
        return ciphertext

    elif self.mode == 'CBC':
        internal_state = [array('B', map(xor, plaintext_state[x], self.iv[x])) for x in range(4)]
        internal_state = self.encrypt_function(internal_state)
        self.iv = internal_state
        ciphertext = self.state_to_int(internal_state)
        return ciphertext

    elif self.mode == 'PCBC':
        internal_state = [array('B', map(xor, plaintext_state[x], self.iv[x])) for x in range(4)]
        internal_state = self.encrypt_function(internal_state)
        self.iv = [array('B', map(xor, plaintext_state[x], internal_state[x])) for x in range(4)]
        ciphertext = self.state_to_int(internal_state)
        return ciphertext

    elif self.mode == 'CFB':
        internal_state = self.encrypt_function(self.iv)
        internal_state = [array('B', map(xor, plaintext_state[x], internal_state[x])) for x in range(4)]
        self.iv = internal_state
        ciphertext = self.state_to_int(internal_state)
        return ciphertext

    elif self.mode == 'OFB':
        internal_state = self.encrypt_function(self.iv)
        self.iv = internal_state
        internal_state = [array('B', map(xor, plaintext_state[x], internal_state[x])) for x in range(4)]
        ciphertext = self.state_to_int(internal_state)
        return ciphertext


def encrypt_function(self, internal_state):

    K= []
    # Run Encryption Steps For Appropriate Number of Rounds
    for round_num in range(3):
        p = []
        if round_num > 0:

            p = internal_state
            if round_num==1:
                self.intermediary_values=[p, K]
            if round_num ==2:
                self.intermediate_values_round_2=[p,K]

            internal_state = [array('B',map(xor,internal_state[0], K[0])),
                              array('B',map(xor,internal_state[1], K[1])),
                              array('B',map(xor,internal_state[2], K[2])),
                              array('B',map(xor,internal_state[3], K[3]))
                              ]


        # S-box Layer
        if self.s_val == 4:
            sbox_state = [array('B', [self.sbox4[state_nib] for state_nib in state_row]) for state_row in internal_state
        else:
            sbox_state = [array('B', [self.sbox8[state_byte] for state_byte in state_row]) for state_row in internal_st
```

```
            internal_state = sbox_state

            key_state=[array('B', [0b00000000,0b00000000,0b00000000,0b00000000]),
                       array('B', [0b00000000,0b00000000,0b00000000,0b00000000]),
                       array('B', [0b00000000,0b00000000,0b00000000,0b00000000]),
                       array('B', [0b00000000,0b00000000,0b00000000,0b00000000])]



            # AddRoundConstant
            round_constant = self.round_constants[round_num]
            c0 = round_constant & 0xF
            c1 = round_constant >> 4
            c2 = 0x2
            internal_state[0][0] ^= c0
            internal_state[1][0] ^= c1
            internal_state[2][0] ^= c2


            # AddTweakKey
            key_state[0] = array('B', map(xor, key_state[0], self.key_schedule[round_num][0]))
            key_state[1] = array('B', map(xor, key_state[1], self.key_schedule[round_num][1]))

            # Shift Rows
            internal_state = [internal_state[0],
                              array('B', [internal_state[1][3], internal_state[1][0], internal_state[1][1], internal_state[
                              array('B', [internal_state[2][2], internal_state[2][3], internal_state[2][0], internal_state[
                              array('B', [internal_state[3][1], internal_state[3][2], internal_state[3][3], internal_state[

            #separate shift rows for key-state matrix
            key_state = [key_state[0],
                              array('B', [key_state[1][3], key_state[1][0], key_state[1][1], key_state[1][2]]),
                              array('B', [key_state[2][2], key_state[2][3], key_state[2][0], key_state[2][1]]),
                              array('B', [key_state[3][1], key_state[3][2], key_state[3][3], key_state[3][0]])]
            # MixColumns
            mix_1 = array('B', map(xor, internal_state[1], internal_state[2]))
            mix_2 = array('B', map(xor, internal_state[0], internal_state[2]))
            mix_3 = array('B', map(xor, internal_state[3], mix_2))

            internal_state = [mix_3, internal_state[0], mix_1, mix_2]


            mix_1 = array('B', map(xor, key_state[1], key_state[2]))
            mix_2 = array('B', map(xor, key_state[0], key_state[2]))
            mix_3 = array('B', map(xor, key_state[3], mix_2))

            key_state = [mix_3, key_state[0], mix_1, mix_2]
            K = key_state

        return internal_state

    def decrypt(self, ciphertext):

        try:
            ct_int = ciphertext & self.block_mask
            ciphertext_state = self.int_to_state(ct_int)
        except (ValueError, TypeError):
            print('Invalid Ciphertext Value!')
            print('Please Provide Ciphertext as int')
            raise

        # Prepare Based On Mode
        if self.mode == 'ECB':
            internal_state = ciphertext_state
            internal_state = self.decrypt_function(internal_state)
            plaintext = self.state_to_int(internal_state)
            return plaintext

        elif self.mode == 'CTR':
            internal_state = self.int_to_state(self.counter)
```

```
            self.counter += 1
            internal_state = self.encrypt_function(internal_state)
            internal_state = [array('B', map(xor, ciphertext_state[x], internal_state[x])) for x in range(4)]
            plaintext = self.state_to_int(internal_state)
            return plaintext

        elif self.mode == 'CBC':
            internal_state = self.decrypt_function(ciphertext_state)
            internal_state = [array('B', map(xor, internal_state[x], self.iv[x])) for x in range(4)]
            self.iv = ciphertext_state
            plaintext = self.state_to_int(internal_state)
            return plaintext

        elif self.mode == 'PCBC':
            internal_state = self.decrypt_function(ciphertext_state)
            internal_state = [array('B', map(xor, internal_state[x], self.iv[x])) for x in range(4)]
            self.iv = [array('B', map(xor, internal_state[x], ciphertext_state[x])) for x in range(4)]
            plaintext = self.state_to_int(internal_state)
            return plaintext

        elif self.mode == 'CFB':
            internal_state = self.encrypt_function(self.iv)
            internal_state = [array('B', map(xor, ciphertext_state[x], internal_state[x])) for x in range(4)]
            self.iv = ciphertext_state
            plaintext = self.state_to_int(internal_state)
            return plaintext

        elif self.mode == 'OFB':
            internal_state = self.encrypt_function(self.iv)
            self.iv = internal_state
            internal_state = [array('B', map(xor, ciphertext_state[x], internal_state[x])) for x in range(4)]
            plaintext = self.state_to_int(internal_state)
            return plaintext

    def decrypt_function(self, internal_state):

        for round_num in range(self.rounds - 1, -1, -1):

            # Inverse Mix Columns
            mix_1 = array('B', map(xor, internal_state[0], internal_state[3]))
            mix_2 = array('B', map(xor, internal_state[1], internal_state[3]))
            mix_3 = array('B', map(xor, internal_state[2], mix_2))
            internal_state = [internal_state[1], mix_3, mix_2, mix_1]

            # Inverse Shift Rows
            internal_state = [internal_state[0],
                              array('B', [internal_state[1][1],
                                          internal_state[1][2],
                                          internal_state[1][3],
                                          internal_state[1][0]]),
                              array('B', [internal_state[2][2],
                                          internal_state[2][3],
                                          internal_state[2][0],
                                          internal_state[2][1]]),
                              array('B', [internal_state[3][3],
                                          internal_state[3][0],
                                          internal_state[3][1],
                                          internal_state[3][2]])]

            # Inverse AddTweakKey
            internal_state[0] = array('B', map(xor, internal_state[0], self.key_schedule[round_num][0]))
            internal_state[1] = array('B', map(xor, internal_state[1], self.key_schedule[round_num][1]))

            # Inverse AddRoundConstant
            round_constant = self.round_constants[round_num]
            c0 = round_constant & 0xF
            c1 = round_constant >> 4
            c2 = 0x2
            internal_state[0][0] ^= c0
            internal_state[1][0] ^= c1
            internal_state[2][0] ^= c2
```

```
            # Inverse S-box Layer
            if self.s_val == 4:
                sbox_state = [array('B', [self.sbox4_inv[state_nib] for state_nib in state_row]) for state_row in internal_
            else:
                sbox_state = [array('B', [self.sbox8_inv[state_byte] for state_byte in state_row]) for state_row in internal

            internal_state = sbox_state
        return internal_state


    def deep_copy(self, inp):
        copy= []
        for i in range(4):
            row=[]
            for j in range(4):
                row[i][j]=inp[i][j]
            copy[i]=row
        return copy

if __name__ == "__main__":

    p = SkinnyCipher(0x17401096d712b2adcc0143a91dddb11c)
    d = p.encrypt(0x5768de09fd1f69fd2a90de397270597a)
    w = p.decrypt(0x1de2136fb373e0522cc2351306e9f62d)
    print('Encrypt:', format(d, '#018X'))
    print('Decrypt:', format(w, '#018X'))
```

A.2 Helper Library

Listing A.2: Helper library for the experiments

```
import random
import string
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.ticker import StrMethodFormatter
import numpy as np


def hw(int_no):

    c = 0
    while(int_no):
        int_no &= (int_no - 1)
        c += 1
    return c

def gen_plaintexts(n):
    plaintexts=[]
    for i in range(n):
        plaintext=""
        for j in range(16):
            plaintext+=str(np.random.randint(0, 16))
        plaintexts.append(int(plaintext))
    return plaintexts

# Majority voting algorithm
def majority_vote(votes):
    # votes = list of integer votes
    votes_table = {}
    for vote in votes:
        if vote in votes_table:
```

```
            votes_table[vote] += 1
        else:
            votes_table[vote] = 1
    return votes_table

# Get list of all corresponding key dependant sboxes
def determine_kdi(target_nibble):
    if target_nibble in [0,4,12]:
        return [0,4,12]
    elif target_nibble in [1,5,13]:
        return [1,5,13]
    elif target_nibble in [2,6,14]:
        return [2,6,14]
    elif target_nibble in [3,7,15]:
        return [3,7,15]

def recover_pk(val):

    # Inverse Mix Col
    mix_1 = val[0] ^ val[3]
    mix_2 = val[1] ^ val[3]
    mix_3 = val[2] ^ mix_2
    internal_state = [val[1], mix_3,mix_2, mix_1]

    # Inverse Shift Rows
    internal_state = [internal_state[0],
                      [internal_state[1][1], internal_state[1][2], internal_state[1][3], internal_state[1][0]],
                      [internal_state[2][2], internal_state[2][3], internal_state[2][0], internal_state[2][1]],
                      [internal_state[3][3], internal_state[3][0], internal_state[3][1], internal_state[3][2]],]

    return internal_state

def plot_multiple_attacks(plot_0,plot_1,plot_2,plot_3, label_0,label_1,label_2,label_3,outfile):

    #xaxs=np.array(list(range(0,len(plot_0),1)))
    #plot_label= "TK1_0 recovery rate | experiments: "+ str(len(keys))+", std: "+str(std)
    plt.figure()
    plt.ylabel('Success rate')
    plt.xlabel('Traces')
    plt.plot(np.array(plot_0), label=label_0, c="red")
    plt.plot(np.array(plot_1), label=label_1, c="blue")
    plt.plot(np.array(plot_2), label=label_2, c="green")
    plt.plot(np.array(plot_3), label=label_3, c="black")
    plt.grid(axis='y')
    plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.0f}')) # No decimal places
    plt.legend()
    plt.draw()
    plt.savefig(outfile)


def plot_multiple_attacks(plot_0,plot_1,plot_2,plot_3, label_0,label_1,label_2,label_3,outfile):
    plot_0.insert(0, 0.0)
    plot_1.insert(0, 0.0)
    plot_2.insert(0, 0.0)
    plot_3.insert(0, 0.0)
    #xaxs=np.array(list(range(0,len(plot_0),1)))
    #plot_label= "TK1_0 recovery rate | experiments: "+ str(len(keys))+", std: "+str(std)
    plt.figure()
    plt.ylabel('Success rate')
    plt.xlabel('Traces')
    plt.plot(np.array(plot_0), label=label_0, c="red")
    plt.plot(np.array(plot_1), label=label_1, c="blue")
    plt.plot(np.array(plot_2), label=label_2, c="green")
    plt.plot(np.array(plot_3), label=label_3, c="black")
    plt.grid(axis='y')
    plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.0f}')) # No decimal places
    plt.legend()
    plt.draw()
    plt.savefig(outfile)

def plot_single_attack(plot_0, label_0, outfile):
```

```
    plt.figure()
    plt.ylabel('Success rate')
    plt.xlabel('Traces')
    plt.plot(np.array(plot_0), label=label_0, c="black")
    plt.legend()
    plt.draw()

    plt.savefig(outfile)

def plot_TK_recovery_rates(success_rate_unanimous,success_rate_majority_vote,success_rate_individual,succcess_rate_simultanc
    # Plot the results of each experiment and save to file

    fig, axs = plt.subplots(1,4,figsize=(28,7),sharey=True)
    #fig.suptitle('success-discard rate')
    plt.subplots_adjust(bottom=0.2)

    axs[0].set_title('unanimous')
    axs[1].set_title('majority vote')
    axs[2].set_title('individual')
    axs[3].set_title('simultanous')

    axs[0].plot(np.array(success_rate_unanimous), label="valid and correct", c="red")
    axs[1].plot(np.array(success_rate_majority_vote), label="valid and correct", c="red")
    axs[2].plot(np.array(success_rate_individual), label="valid and correct", c="red")

    #std_string="experiments: "+str(len(keys))+" \n std: "+ str(sigma)
    #axs[3].plot([], [], ' ', label=std_string) #dirty hack to get std into legend
    axs[3].plot(np.array(succcess_rate_simultanous), label="valid and correct", c="red")


    axs[3].legend(bbox_to_anchor=(1.1, 1.05))

    for ax in axs.flat:
        ax.set(xlabel='traces', ylabel='probability')

    # Hide x labels and tick labels for top plots and y ticks for right plots.
    for ax in axs.flat:
        ax.label_outer()

    fig.savefig(outfile, bbox_inches='tight')

def plot_required_traces(plot_0,plot_1,plot_2,plot_3, label_0,label_1,label_2,label_3,outfile,std):
    plt.figure()
    plt.ylabel('Traces')
    plt.xlabel($\sigma$)
    plt.plot(std,np.array(plot_0), label=label_0, c="red")
    plt.plot(std,np.array(plot_1), label=label_1, c="blue")
    plt.plot(std,np.array(plot_2), label=label_2, c="green")
    plt.plot(std,np.array(plot_3), label=label_3, c="black")
    plt.grid(axis='y')
    plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.0f}')) # No decimal places
    plt.legend()
    plt.draw()
    plt.savefig(outfile)
```

A.3 DPA Utilities

Listing A.3: DPA utilities

```
import numpy as np
import scipy as sp
import matplotlib
import matplotlib.pyplot as plt
```

```
import math
import sys

import skinny
import helpers

from scipy import stats
from array import array
from operator import xor

np.set_printoptions(threshold=sys.maxsize)

def compute_intemediate_values(P,K): #Plaintexts list, K int
    known_RK=[]
    IS=[] #List of each p's IS'
    CP=[] #P'

    for plaintext in P:
        #valid_modes = ['ECB', 'CTR', 'CBC', 'PCBC', 'CFB', 'OFB']
        cipher = skinny.SkinnyCipher(K,64,64) #TK1
        cipher.encrypt(plaintext)
        p=cipher.intermediary_values[0]
        RK=cipher.intermediary_values[1]

        p_xor_k = [array('B',map(xor,p[0], RK[0])),
                   array('B',map(xor, p[1], RK[1])),
                   array('B',map(xor, p[2], RK[2])),
                   array('B',map(xor, p[3], RK[3]))
                  ]
        p_xor_k=np.matrix(p_xor_k)

        s=[]
        sbox4 = array('B', [12, 6, 9, 0, 1, 10, 2, 11, 3, 8, 5, 13, 4, 14, 7, 15])
        for x in np.nditer(p_xor_k):
            s.append(sbox4[x])

        IS.append(np.matrix(s))
        CP.append(np.asmatrix(p).flatten())

    return [IS, CP, np.matrix(RK)]

def gen_traces(states, std):
    traces = []
    for i in range(len(states)):
        t=[]
        for j in range(16):
            mu=helpers.hw(states[i].item(j))
            t.append(np.random.normal(mu,std))
        traces.append(np.array(t).flatten())
    return np.array(traces)

# Formulate key hypothesis, V for single sbox
def compute_v(sbox_index,list_of_interm_p):
    sbox4 = array('B', [12, 6, 9, 0, 1, 10, 2, 11, 3, 8, 5, 13, 4, 14, 7, 15])
    V = []
    for i in range(len(list_of_interm_p)):
        row=[]
        for j in range(16):
            hyp_val = list_of_interm_p[i].item(sbox_index) ^ j
            row.append(helpers.hw(sbox4[hyp_val]))
        V.append(row)
    return V

# Maximum-log-likelihood distinguisher
def distinguisher(V,traces,target_nibble):
    std=0.5
    scores=[]
    for i in range(16):
        score=0
        for (t,v) in zip(traces,V):
            mu=v.item(i)
```

```
            d=sp.stats.norm(mu,std)
            l=t[target_nibble]
            score+= math.log(d.pdf(l))
        scores.append(score)
    return scores

# MLL via multivariate distribution of all same-key-dependent s-boxes
def distinguisher_multivariate(V,T,kdi):
    V_0=V[0]
    V_1=V[1]
    V_2=V[2]
    std=0.5 #std of distinguisher
    scores=[]
    cov_matrix= np.identity(3) * std**2

    # Selection function
    for i in range(16):
        score=0
        for j in range(len(T)):

            mu=[V_0[j].item(i),V_1[j].item(i),V_2[j].item(i)]
            Sigma=cov_matrix
            distrib = sp.stats.multivariate_normal(mu, Sigma)
            traces=[T[j][kdi[0]],T[j][kdi[1]],T[j][kdi[2]]]
            score+= math.log(distrib.pdf(traces))
        scores.append(score)
    return scores

# Standard univariate DPA attack
def individual_atk(T,clear_text_nibbles,target_nibble):
    V = np.matrix(compute_v(target_nibble,clear_text_nibbles))
    scores=distinguisher(V,T,target_nibble)

    return(scores.index(max(scores)))

def simultanous_atk(T,clear_text_nibbles,target_nibble):
    kdi=helpers.determine_kdi(target_nibble)
    V = []
    scores=[]
    for ind in kdi:
        V.append(np.matrix(compute_v(ind,clear_text_nibbles)))
    scores=distinguisher_multivariate(V,T,kdi)

    return(scores.index(max(scores)))

def majority_vote_atk(T,clear_text_nibbles,target_nibble):
    kdi=helpers.determine_kdi(target_nibble)
    V = []
    scores=[]
    for ind in kdi:
        V.append(np.matrix(compute_v(ind,clear_text_nibbles)))
        scores.append(distinguisher(V[kdi.index(ind)],T,ind))

    argmax_scores = [score.index(max(score)) for score in scores]

    votes_table = helpers.majority_vote(argmax_scores)
    k_cand = max(votes_table,key=votes_table.get)

    if votes_table.get(k_cand) > 1:
        return k_cand # If there are a winner, return winner
    else:
        return -1 # else Discard voting

def unanimous_attack(T,clear_text_nibbles,target_nibble):
    kdi=helpers.determine_kdi(target_nibble)
    V = []
    scores=[]
    for ind in kdi:
        V.append(np.matrix(compute_v(ind,clear_text_nibbles)))
        scores.append(distinguisher(V[kdi.index(ind)],T,ind))
```

```
    argmax_scores = [score.index(max(score)) for score in scores] #Argmax of list of scores

    if len(set(argmax_scores))== 1:
        return argmax_scores[0]
    else:
        return -1



# Returning scores, no argmax
# MLL via multivariate distribution of all same-key-dependent s-boxes
def simultanous_atk_scores(T,clear_text_nibbles,target_nibble):
    kdi=helpers.determine_kdi(target_nibble)
    V = []
    for ind in kdi:
        V.append(np.matrix(compute_v(ind,clear_text_nibbles)))
    return distinguisher_multivariate(V,T,kdi)

def majority_vote_atk_scores(T,clear_text_nibbles,target_nibble):
    kdi=helpers.determine_kdi(target_nibble)
    V = []

    for ind in kdi:
        V.append(np.matrix(compute_v(ind,clear_text_nibbles)))
        scores.append(distinguisher(V[kdi.index(ind)],T,ind))

    argmax_scores = [score.index(max(score)) for score in scores]

    votes_table = helpers.majority_vote(argmax_scores)
    k_cand = max(votes_table,key=votes_table.get)

    if votes_table.get(k_cand) > 1:
        return k_cand # If there are a winner, return winner
    else:
        return -1 # else Discard voting

def unanimous_attack_scores(T,clear_text_nibbles,target_nibble):
    kdi=helpers.determine_kdi(target_nibble)
    V = []
    scores=[]
    for ind in kdi:
        V.append(np.matrix(compute_v(ind,clear_text_nibbles)))
        scores.append(distinguisher(V[kdi.index(ind)],T,ind))

    a=np.matrix(scores)

    argmax_scores = [score.index(max(score)) for score in scores] #Argmax of list of scores

    if len(set(argmax_scores))== 1:
        return a.sum(axis=0)
    else:
        return [-1]

def individual_atk_scores(T,clear_text_nibbles,target_nibble):
    V = np.matrix(compute_v(target_nibble,clear_text_nibbles))
    return distinguisher(V,T,target_nibble)
```

A.4 DPA Attacks Against SKINNY-64-64

Listing A.4: Attack execution

```
import numpy as np
import scipy as sp
```

```
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import math
import sys

import skinny
import helpers
import dpautils

from scipy import stats
from array import array
from operator import xor

np.set_printoptions(threshold=sys.maxsize)

#Parameters as program args
N=int(sys.argv[1]) # 20
std=float(sys.argv[2]) # 0.5
number_of_experiments=int(sys.argv[3]) # 200

TK1_0=[0,1,2,3] #

keys=np.random.randint(2147483647, 9223372036854775807, size=number_of_experiments, dtype=np.int64)

P=helpers.gen_plaintexts(N)

# Precompute intermediate values and power traces for different keys

intermediate_values=[]
for key in keys:
    val=dpautils.compute_intemediate_values(P,int(key))
    interm_values=val[0]
    clear_text=val[1]
    TK1=val[2].A1
    T = dpautils.gen_traces(interm_values, std)
    intermediate_values.append([interm_values,clear_text,TK1,T])

simultanous_success_rate_pr_n=[]
Ns=list(range(1,N,1))
for n in Ns:
    probs=[]

    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]

        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.simultanous_atk(t,ct,i)
            TK1_0_guess.append(nibble_guess)

        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            probs.append(1)
        else:
            probs.append(0)
    simultanous_success_rate_pr_n.append(np.average(probs))

majority_vote_success_rate_pr_n=[]
Ns=list(range(1,N,1))
for n in Ns:
    probs=[]

    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
```

```
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]
        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.majority_vote_atk(t,ct,i)
            TK1_0_guess.append(nibble_guess)

        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            probs.append(1)
        else:
            probs.append(0)
    majority_vote_success_rate_pr_n.append(np.average(probs))

unanimous_success_rate_pr_n=[]
Ns=list(range(1,N,1))
for n in Ns:
    probs=[]

    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]
        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.unanimous_attack(t,ct,i)
            TK1_0_guess.append(nibble_guess)

        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            probs.append(1)
        else:
            probs.append(0)
    unanimous_success_rate_pr_n.append(np.average(probs))

standard_attack_success_rate_pr_n=[]
Ns=list(range(1,N,1))
for n in Ns:
    probs=[]

    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]
        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.individual_atk(t,ct,i)
            TK1_0_guess.append(nibble_guess)

        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            probs.append(1)
        else:
            probs.append(0)
    standard_attack_success_rate_pr_n.append(np.average(probs))

success_rates=  [ simultanous_success_rate_pr_n,
                  majority_vote_success_rate_pr_n,
                  unanimous_success_rate_pr_n,
```

```
                    standard_attack_success_rate_pr_n]

outfile="TK1-10-recovery-rate-std-"+str(std)+".png"
np.save(outfile,success_rates)

# Plotting
helpers.plot_multiple_attacks(simultanous_success_rate_pr_n, majority_vote_success_rate_pr_n,
                  unanimous_success_rate_pr_n, individual_attack_success_rate_pr_n,
                  "simultanous","majority vote",
                  "unanimous","standard",
                 outfile)
```

A.5 Power Trace Generation

Listing A.5: Generate required traces

```
import numpy as np
import scipy as sp
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib.ticker import StrMethodFormatter
import math
import sys
import copy
import helpers
import dpautils

from scipy import stats
from array import array
from operator import xor

np.set_printoptions(threshold=sys.maxsize)

N=20
number_of_experiments=50
keys=np.random.randint(2147483647, 9223372036854775807, size=number_of_experiments, dtype=np.int64)
alfa=0.005
threshold=0.995

P=helpers.gen_plaintexts(N)

target_nibble=0

STD=np.arange(0.1, 3, 0.1)

all_intermediate=[]
for std in STD:
    # Precompute intermediate values and power traces for different keys
    intermediate_values=[]
    required_traces=[]
    for key in keys:
        val=dpautils.compute_intemediate_values(P,int(keys[0]))
        interm_values=val[0]
        clear_text=val[1]
        TK1=val[2].A1
        T = dpautils.gen_traces(interm_values, std)
        intermediate_values.append([interm_values,clear_text,TK1,T])

    i=1
    while True:
        res=[]
        for j,key in enumerate(keys):
```

```
        clear_text = intermediate_values[j][1]
        TK1 = intermediate_values[j][2]
        T = intermediate_values[j][3]
        t=T[:i]
        ct=clear_text[:i]
        nibble_guess=dpautils.simultanous_atk(t,ct,target_nibble)

        #_____SUCCESS/FAIL_________________
        if(TK1[target_nibble] == nibble_guess):
            res.append(1)
        else:
            res.append(0)

    if np.average(res)>=float(threshold):
        required_traces.append(i)
        break
    i+=1
i=1
while True:
    res=[]
    for j,key in enumerate(keys):
        clear_text = intermediate_values[j][1]
        TK1 = intermediate_values[j][2]
        T = intermediate_values[j][3]
        t=T[:i]
        ct=clear_text[:i]
        nibble_guess=dpautils.majority_vote_atk(t,ct,target_nibble)

        #_____SUCCESS/FAIL_________________
        if(TK1[target_nibble] == nibble_guess):
            res.append(1)
        else:
            res.append(0)

    if np.average(res)>=float(threshold):
        required_traces.append(i)
        break
    i+=1
i=1
while True:
    res=[]
    for j,key in enumerate(keys):
        clear_text = intermediate_values[j][1]
        TK1 = intermediate_values[j][2]
        T = intermediate_values[j][3]
        t=T[:i]
        ct=clear_text[:i]
        nibble_guess=dpautils.unanimous_attack(t,ct,target_nibble)

        #_____SUCCESS/FAIL_________________
        if(TK1[target_nibble] == nibble_guess):
            res.append(1)
        else:
            res.append(0)

    if np.average(res)>=float(threshold):
        required_traces.append(i)
        break
    i+=1

i=1
while True:
    res=[]
    for j,key in enumerate(keys):
        clear_text = intermediate_values[j][1]
        TK1 = intermediate_values[j][2]
        T = intermediate_values[j][3]
        t=T[:i]
        ct=clear_text[:i]
        nibble_guess=dpautils.individual_atk(t,ct,target_nibble)
```

```
            #_____SUCCESS/FAIL_________________
            if(TK1[target_nibble] == nibble_guess):
                res.append(1)
            else:
                res.append(0)

        if np.average(res)>=float(threshold):
            required_traces.append(i)
            break
        i+=1
    all_intermediate.append(required_traces)

    np.save("required_traces",all_intermediate)
```

A.6 Determining The Confidence Of The Distinguishers

Listing A.6: Modelling distinguisher confidence

```
import numpy as np
import scipy as sp
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
from matplotlib.ticker import StrMethodFormatter
import math
import sys

import skinny
import helpers
import dpautils

from scipy import stats
from array import array
from operator import xor

np.set_printoptions(threshold=sys.maxsize)

# Take in parameters as args
N=int(sys.argv[1]) #20
std=float(sys.argv[2]) #0.5
number_of_experiments=int(sys.argv[3]) #200

TK1_0=[0,1,2,3] # 3x traces
TK1_1=[4,5,6,7]
TK1_2_unsorted=[0,1,2,3] # 3x traces
Tk1_3_unsorted=[8,9,10,11]

keys=np.random.randint(2147483647, 9223372036854775807, size=number_of_experiments, dtype=np.int64)

P=helpers.gen_plaintexts(N)

# Precompute intermediate values and power traces for different keys

intermediate_values=[]
for key in keys:
    val=dpautils.compute_intemediate_values(P,int(key))
    interm_values=val[0]
    clear_text=val[1]
    TK1=val[2].A1
    T = dpautils.gen_traces(interm_values, std)
    intermediate_values.append([interm_values,clear_text,TK1,T])

simultanous_success_rate_pr_n=[]
```

```
Ns=list(range(1,N,1))
for n in Ns:
    correct=[]
    incorrect=[]
    discarded=[]

    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]

        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.simultanous_atk(t,ct,i)
            TK1_0_guess.append(nibble_guess)

        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            correct.append(1)
            incorrect.append(0)
        else:
            incorrect.append(1)
            correct.append(0)
    simultanous_success_rate_pr_n.append([np.average(correct),np.average(incorrect)])

majority_vote_success_rate_pr_n=[]
Ns=list(range(1,N,1))
for n in Ns:
    correct=[]
    incorrect=[]
    discarded=[]

    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]
        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.majority_vote_atk(t,ct,i)
            TK1_0_guess.append(nibble_guess)

        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            correct.append(1)
            discarded.append(0)
            incorrect.append(0)

        elif -1 in TK1_0_guess:
            discarded.append(1)
            correct.append(0)
            incorrect.append(0)
        else:
            incorrect.append(1)
            correct.append(0)
            discarded.append(0)
    majority_vote_success_rate_pr_n.append([np.average(correct),np.average(incorrect),np.average(discarded)])

unanimous_success_rate_pr_n=[]
Ns=list(range(1,N,1))
for n in Ns:
    correct=[]
    incorrect=[]
    discarded=[]
```

```
    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]
        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.unanimous_attack(t,ct,i)
            TK1_0_guess.append(nibble_guess)


        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            correct.append(1)
            discarded.append(0)
            incorrect.append(0)

        elif -1 in TK1_0_guess:
            discarded.append(1)
            correct.append(0)
            incorrect.append(0)
        else:
            incorrect.append(1)
            correct.append(0)
            discarded.append(0)
    unanimous_success_rate_pr_n.append([np.average(correct),np.average(incorrect),np.average(discarded)])

individual_attack_success_rate_pr_n=[]
Ns=list(range(1,N,1))
for n in Ns:
    correct=[]
    incorrect=[]

    for i,key in enumerate(keys):
        clear_text = intermediate_values[i][1]
        TK1 = intermediate_values[i][2]
        T = intermediate_values[i][3]
        TK1_0_guess=[]

        # Attack TK1 nibble i (first row of TK1)
        for i in range(4):
            t=T[:n]
            ct=clear_text[:n]
            nibble_guess=dpautils.individual_atk(t,ct,i)
            TK1_0_guess.append(nibble_guess)


        #_____SUCCESS/FAIL_________________
        if(np.array_equal(TK1[0:4],TK1_0_guess)):
            correct.append(1)
            incorrect.append(0)
        else:
            incorrect.append(1)
            correct.append(0)
    individual_attack_success_rate_pr_n.append([np.average(correct),np.average(incorrect)])


simultanous_confidence=np.matrix(simultanous_success_rate_pr_n)
correct=simultanous_confidence[:,0]
incorrect=simultanous_confidence[:,1]
outfile="confidence_simultanous.png"
plt.figure()
plt.plot(correct, label="valid and correct", c="red")
plt.plot(incorrect, label="valid and incorrect", c="blue")
plt.grid(axis='y')
plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.0f}')) # No decimal places
plt.legend()
```

```
plt.draw()
plt.savefig(outfile)

majority_voting_confidence=np.matrix(majority_vote_success_rate_pr_n)
correct=majority_voting_confidence[:,0]
incorrect=majority_voting_confidence[:,1]
discarded=majority_voting_confidence[:,2]
outfile="confidence_majority_vote.png"
plt.figure()
plt.plot(correct, label="valid and correct", c="red")
plt.plot(incorrect, label="valid and incorrect", c="blue")
plt.plot(discarded, label="discarded", c="green")
plt.grid(axis='y')
plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.0f}')) # No decimal places
plt.legend()
plt.draw()
plt.savefig(outfile)

unanimous_confidence=np.matrix(unanimous_success_rate_pr_n)
correct=unanimous_confidence[:,0]
incorrect=unanimous_confidence[:,1]
discarded=unanimous_confidence[:,2]
outfile="confidence_unanimous.png"
plt.figure()
plt.plot(correct, label="valid and correct", c="red")
plt.plot(incorrect, label="valid and incorrect", c="blue")
plt.plot(discarded, label="discarded", c="green")
plt.grid(axis='y')
plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.0f}')) # No decimal places
plt.legend()
plt.draw()
plt.savefig(outfile)

individual_confidence=np.matrix(individual_attack_success_rate_pr_n)
correct=individual_confidence[:,0]
incorrect=individual_confidence[:,1]
outfile="confidence_individual.png"
plt.figure()
plt.plot(correct, label="valid and correct", c="red")
plt.plot(incorrect, label="valid and incorrect", c="blue")
plt.grid(axis='y')
plt.gca().xaxis.set_major_formatter(StrMethodFormatter('{x:,.0f}')) # No decimal places
plt.legend()
plt.draw()
plt.savefig(outfile)
```

A.7 Generation Of Second Round Intermediate Values

Listing A.7: Generation of second round intermediate values

```
import numpy
from array import array
from operator import xor

def linear_operations(internal_state):
    # Reshape internal state info 4x4 matrix
    internal_state=np.array(val).reshape(4,4)

    internal_state = [internal_state[0],
    array('B', [internal_state[1][3], internal_state[1][0], internal_state[1][1], internal_state[1][2]]),
    array('B', [internal_state[2][2], internal_state[2][3], internal_state[2][0], internal_state[2][1]]),
    array('B', [internal_state[3][1], internal_state[3][2], internal_state[3][3], internal_state[3][0]])
```

```
    ]


    # MixColumns
    mix_1 = array('B', map(xor, internal_state[1], internal_state[2]))
    mix_2 = array('B', map(xor, internal_state[0], internal_state[2]))
    mix_3 = array('B', map(xor, internal_state[3], mix_2))

    internal_state = [mix_3, internal_state[0], mix_1, mix_2]

    return internal_state

def add_const(internal_state,round_num):
    internal_state=np.array(internal_state).reshape(4,4)
    round_constants = array('B', [
    0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E, 0x3C, 0x39, 0x33, 0x27,
    0x0E, 0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E, 0x1C, 0x38,
    0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04, 0x09,
    0x13, 0x26, 0x0c, 0x19, 0x32, 0x25, 0x0a, 0x15, 0x2a,
    0x14, 0x28, 0x10, 0x20
    ])

        round_constant = round_constants[round_num]
        c0 = round_constant & 0xF
        c1 = round_constant >> 4
        c2 = 0x2
        internal_state[0][0] ^= c0
        internal_state[1][0] ^= c1
        internal_state[2][0] ^= c2
        return internal_state

 def compute_intermediate_results_for_round2(internal_state):
    internal_states_post_round_1=[]
    for p in clear_text:
        s=[]
        # AddRoundTweakey
        p_xor_k=np.bitwise_xor(p,SK)

        # SubBytes
        sbox4 = array('B', [12, 6, 9, 0, 1, 10, 2, 11, 3, 8, 5, 13, 4, 14, 7, 15])
        for x in np.nditer(p_xor_k):
            s.append(sbox4[x])

        # ShiftRows and MixCols then add round constant of the second round
        s = linear_operations(add_const(s,1))
        internal_states_post_round_1.append(np.matrix(s))
      return internal_states_post_round_1
```

Appendix B

Other Appendices

B.1 Distinguishing Scores For Fig.5.2

Listing B.1: Distinguishing scores for fig 5.2

```
[[ -10.2   -14.4   -15.75  -16.    -16.52  -17.7   -20.37  -27.8   -28.2
   -31.74  -33.07  -36.21  -40.79  -41.25  -41.75  -44.26  -48.19  -56.59
   -64.94  -65.55  -65.79  -66.29  -74.68  -83.1   -84.55  -85.36  -90.76
   -91.26  -91.96  -98.85  -99.09 -106.15 -107.19 -109.54 -110.84 -114.06
  -116.34 -116.57 -116.82 -119.44 -121.39 -121.65 -124.05 -124.31 -126.14
  -126.61 -127.69 -128.26 -128.59]
 [  -3.27   -3.83   -7.19   -7.45   -7.97  -11.62  -11.86  -28.88  -39.65
   -40.89  -44.26  -54.23  -55.01  -60.71  -64.71  -64.94  -87.75  -90.07
   -92.36  -92.96 -102.02 -103.05 -105.36 -107.68 -114.26 -119.24 -145.51
  -151.06 -155.71 -157.3  -164.84 -166.51 -180.65 -189.13 -195.34 -195.67
  -197.94 -200.39 -200.64 -202.51 -202.75 -212.11 -214.18 -215.89 -232.88
  -233.96 -235.04 -246.9  -248.3 ]
 [  -3.27   -7.47   -7.82   -8.07   -8.59  -14.54  -17.21  -24.64  -35.4
   -35.79  -42.11  -43.6   -44.27  -49.98  -61.46  -63.97  -67.9   -70.21
   -72.5   -73.1   -82.16  -93.61 -102.   -104.32 -120.04 -133.18 -138.58
  -144.13 -156.73 -163.62 -171.16 -172.83 -173.87 -192.47 -207.6  -210.83
  -213.01 -221.68 -221.93 -223.8  -225.75 -226.02 -226.25 -235.61 -237.44
  -243.13 -246.97 -258.83 -265.3 ]
 [  -1.4   -13.23  -16.6   -25.81  -31.27  -32.45  -34.28  -36.12  -58.06
   -59.3   -60.64  -63.78  -64.56  -78.88  -82.88  -84.84  -88.77  -90.91
   -93.07 -105.15 -124.62 -125.12 -143.6  -145.73 -152.31 -157.29 -162.69
  -168.24 -172.89 -173.18 -190.38 -193.25 -198.84 -207.32 -213.53 -214.95
  -217.23 -219.68 -227.   -234.5  -242.16 -244.97 -247.04 -247.3  -254.71
  -255.79 -256.87 -280.38 -281.78]
 [  -3.27   -3.83   -14.2   -16.94  -17.92  -21.57  -21.82  -38.84  -42.42
   -48.52  -51.89  -52.21  -57.09  -58.17  -69.66  -69.89  -81.26  -83.58
   -85.86  -90.21  -92.86  -96.83  -99.14 -101.46 -101.78 -102.43 -116.26
  -130.34 -131.09 -132.68 -134.57 -136.24 -141.83 -142.06 -142.43 -142.76
  -145.03 -153.71 -155.49 -155.74 -155.97 -158.79 -166.68 -169.5  -176.91
  -180.76 -181.85 -186.06 -189.33]
 [  -0.33   -1.26   -1.61   -1.87   -2.38   -2.8    -3.05   -4.89   -5.29
   -5.68   -6.03   -9.17   -9.85  -10.31  -10.81  -11.05  -11.53  -11.76
  -11.99  -12.59  -12.84  -13.34  -13.56  -13.79  -15.24  -16.05  -17.02
  -18.04  -18.74  -20.33  -20.57  -20.84  -21.33  -23.68  -24.97  -25.3
```

```
  −25.53  −25.76  −26.01  −28.63  −31.15  −32.86  −33.09  −33.36  −33.6
  −34.07  −34.54  −35.1   −35.44]
[  −1.4   −13.23  −13.59  −16.32  −17.31  −18.49  −18.74  −20.58  −24.16
  −24.55  −25.89  −26.2   −26.87  −27.96  −50.93  −51.17  −62.54  −64.68
  −66.84  −71.18  −73.84  −77.81  −96.29  −98.42  −99.87 −100.68 −114.52
 −120.07 −120.77 −122.36 −124.25 −127.12 −132.71 −135.06 −136.36 −136.68
 −145.   −163.89 −165.68 −165.93 −173.59 −176.4  −176.63 −179.45 −186.86
 −187.32 −193.03 −197.24 −197.58]
[  −0.33   −0.9    −1.25   −3.02   −7.07   −7.49  −16.58  −16.83  −27.59
  −27.98  −28.34  −29.82  −30.5   −36.2   −47.69  −56.46  −60.39  −60.62
  −60.84  −61.71  −70.77  −82.21  −84.52  −84.75 −100.46 −113.61 −119.01
 −120.04 −132.63 −148.82 −156.36 −156.64 −162.23 −180.82 −195.95 −206.09
 −206.31 −214.98 −217.7  −219.57 −219.81 −222.62 −222.85 −232.22 −239.63
 −245.32 −245.78 −257.64 −264.1 ]
[  −3.27   −7.47  −13.8   −14.06  −14.57  −14.99  −17.66  −25.09  −25.49
  −36.18  −36.53  −38.01  −50.48  −50.95  −54.94  −57.45  −61.38  −63.69
  −65.98  −66.58  −66.83  −78.27  −86.67  −88.99 −117.83 −143.15 −148.55
 −162.63 −187.17 −194.06 −194.3  −195.97 −210.11 −242.83 −270.88 −274.11
 −274.34 −276.79 −277.04 −279.66 −281.61 −290.97 −299.54 −308.9  −325.89
 −340.19 −340.66 −341.22 −356.75]
[  −1.4   −24.87  −26.22  −28.95  −29.94  −33.59  −35.41  −42.84  −46.42
  −49.96  −53.33  −56.47  −61.05  −62.13  −66.13  −68.09  −68.57  −70.71
  −72.88  −77.22  −79.87  −80.37 −112.93 −115.07 −121.65 −126.63 −127.6
 −133.15 −137.8  −138.08 −139.98 −142.85 −169.54 −178.02 −184.23 −185.65
 −187.93 −190.38 −192.17 −192.41 −209.79 −229.7  −232.1  −232.36 −262.93
 −264.01 −265.1  −269.31 −270.71]
[  −0.33   −4.53   −5.88   −6.13   −6.65   −7.83   −9.66  −17.08  −17.49
  −21.02  −22.36  −23.84  −28.42  −28.88  −29.89  −31.86  −35.78  −36.01
  −36.24  −36.84  −37.09  −48.53  −56.92  −57.15  −63.73  −68.71  −74.11
  −75.14  −79.79  −80.07  −80.31  −80.59  −81.62  −90.1   −96.31  −97.74
 −106.06 −108.07 −108.32 −110.94 −112.89 −113.15 −115.55 −124.92 −126.74
 −127.82 −133.53 −134.09 −135.49]
[  −1.4   −5.6   −8.96  −10.73  −14.78  −18.43  −25.83  −56.44  −67.21
  −68.45  −71.82  −72.14  −72.91  −78.62  −82.61  −90.31 −101.68 −103.81
 −105.98 −106.84 −115.9  −119.87 −128.27 −130.4  −136.98 −141.96 −155.8
 −161.35 −166.   −168.99 −176.53 −179.41 −193.55 −202.02 −208.23 −214.76
 −217.03 −219.48 −222.2  −224.07 −226.02 −235.38 −237.45 −240.26 −257.25
 −258.33 −259.41 −271.27 −272.67]
[  −6.47  −18.3   −19.64  −22.38  −23.37  −23.78  −24.03  −41.05  −44.63
  −48.17  −48.52  −48.84  −53.41  −54.49  −58.49  −58.72  −59.21  −67.26
  −75.36  −79.7   −82.36  −86.33 −104.8  −112.84 −114.3  −115.11 −116.08
 −142.68 −143.38 −144.97 −146.86 −156.34 −161.93 −164.28 −165.57 −165.9
 −174.22 −176.67 −178.46 −178.7  −186.37 −189.18 −191.58 −194.39 −201.8
 −202.27 −207.97 −212.19 −212.52]
[  −0.33   −0.9    −1.25   −3.02   −7.07   −7.49  −10.16  −17.59  −21.17
  −21.56  −21.91  −28.57  −29.25  −30.33  −30.83  −33.34  −34.38  −34.61
  −34.84  −35.7   −38.36  −61.27  −63.58  −63.81  −79.52  −92.67  −93.2
 −107.28 −119.88 −126.77 −128.66 −128.93 −143.07 −161.67 −176.8  −180.03
 −180.25 −180.49 −183.2  −183.45 −183.68 −193.05 −193.28 −213.2  −230.19
 −235.88 −236.34 −240.55 −247.02]
[  −0.33  −12.17  −15.53  −17.31  −21.35  −32.24  −32.49  −49.51  −53.09
  −54.33  −64.72  −65.03  −65.81  −66.9   −78.38  −78.62  −89.99  −90.21
  −90.44  −91.3   −93.96  −97.93 −116.41 −116.63 −123.22 −128.2  −142.03
 −156.11 −160.76 −162.35 −164.24 −164.51 −170.1  −178.58 −184.79 −185.11
 −203.48 −212.15 −214.87 −215.11 −222.78 −225.59 −227.65 −230.47 −237.88
 −238.96 −253.29 −257.5  −258.9 ]
[  −0.33   −4.53   −4.89  −12.18  −23.76  −24.17  −26.    −27.84  −29.06
  −29.45  −29.8   −30.11  −30.79  −34.63  −35.13  −37.1   −37.58  −37.81
  −38.04  −43.16  −45.    −48.97  −57.37  −57.59  −73.31  −86.45  −87.42
  −88.44 −101.04 −101.33 −103.92 −104.19 −105.23 −123.82 −138.96 −140.38
 −148.7  −148.93 −158.12 −167.11 −169.06 −169.32 −169.55 −172.37 −174.19
 −179.88 −185.59 −186.51 −192.97]]
```

B.2 TK1 Permutations For SKINNY-64-64 (32 rounds)

Listing B.2: TK1-permutations for SKINNY-64-64, 1-indexed

```
----------
round: 1
----------
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
----------
round: 2
----------
[[ 9 15  8 13]
 [10 14 12 11]
 [ 0  1  2  3]
 [ 4  5  6  7]]
----------
round: 3
----------
[[ 1  7  0  5]
 [ 2  6  4  3]
 [ 9 15  8 13]
 [10 14 12 11]]
----------
round: 4
----------
[[15 11  9 14]
 [ 8 12 10 13]
 [ 1  7  0  5]
 [ 2  6  4  3]]
----------
round: 5
----------
[[ 7  3  1  6]
 [ 0  4  2  5]
 [15 11  9 14]
 [ 8 12 10 13]]
----------
round: 6
----------
[[11 13 15 12]
 [ 9 10  8 14]
 [ 7  3  1  6]
 [ 0  4  2  5]]
----------
round: 7
----------
[[ 3  5  7  4]
 [ 1  2  0  6]
 [11 13 15 12]
 [ 9 10  8 14]]
----------
round: 8
----------
[[13 14 11 10]
 [15  8  9 12]
 [ 3  5  7  4]
 [ 1  2  0  6]]
----------
round: 9
----------
[[ 5  6  3  2]
 [ 7  0  1  4]
 [13 14 11 10]
 [15  8  9 12]]
----------
round: 10
----------
[[14 12 13  8]
 [11  9 15 10]
 [ 5  6  3  2]
```

```
 [ 7  0  1  4]]
```

---

round: 11

```
[[ 6  4  5  0]
 [ 3  1  7  2]
 [14 12 13  8]
 [11  9 15 10]]
```

---

round: 12

```
[[12 10 14  9]
 [13 15 11  8]
 [ 6  4  5  0]
 [ 3  1  7  2]]
```

---

round: 13

```
[[ 4  2  6  1]
 [ 5  7  3  0]
 [12 10 14  9]
 [13 15 11  8]]
```

---

round: 14

```
[[10  8 12 15]
 [14 11 13  9]
 [ 4  2  6  1]
 [ 5  7  3  0]]
```

---

round: 15

```
[[ 2  0  4  7]
 [ 6  3  5  1]
 [10  8 12 15]
 [14 11 13  9]]
```

---

round: 16

```
[[ 8  9 10 11]
 [12 13 14 15]
 [ 2  0  4  7]
 [ 6  3  5  1]]
```

---

round: 17

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

---

round: 18

```
[[ 9 15  8 13]
 [10 14 12 11]
 [ 0  1  2  3]
 [ 4  5  6  7]]
```

---

round: 19

```
[[ 1  7  0  5]
 [ 2  6  4  3]
 [ 9 15  8 13]
 [10 14 12 11]]
```

---

round: 20

```
[[15 11  9 14]
 [ 8 12 10 13]
 [ 1  7  0  5]
 [ 2  6  4  3]]
```

```
--------------------
round: 21
--------------------
[[ 7  3  1  6]
 [ 0  4  2  5]
 [15 11  9 14]
 [ 8 12 10 13]]
--------------------
round: 22
--------------------
[[11 13 15 12]
 [ 9 10  8 14]
 [ 7  3  1  6]
 [ 0  4  2  5]]
--------------------
round: 23
--------------------
[[ 3  5  7  4]
 [ 1  2  0  6]
 [11 13 15 12]
 [ 9 10  8 14]]
--------------------
round: 24
--------------------
[[13 14 11 10]
 [15  8  9 12]
 [ 3  5  7  4]
 [ 1  2  0  6]]
--------------------
round: 25
--------------------
[[ 5  6  3  2]
 [ 7  0  1  4]
 [13 14 11 10]
 [15  8  9 12]]
--------------------
round: 26
--------------------
[[14 12 13  8]
 [11  9 15 10]
 [ 5  6  3  2]
 [ 7  0  1  4]]
--------------------
round: 27
--------------------
[[ 6  4  5  0]
 [ 3  1  7  2]
 [14 12 13  8]
 [11  9 15 10]]
--------------------
round: 28
--------------------
[[12 10 14  9]
 [13 15 11  8]
 [ 6  4  5  0]
 [ 3  1  7  2]]
--------------------
round: 29
--------------------
[[ 4  2  6  1]
 [ 5  7  3  0]
 [12 10 14  9]
 [13 15 11  8]]
--------------------
round: 30
--------------------
[[10  8 12 15]
 [14 11 13  9]
 [ 4  2  6  1]
 [ 5  7  3  0]]
--------------------
```

```
round: 31
-----------
[[ 2  0  4  7]
 [ 6  3  5  1]
 [10  8 12 15]
 [14 11 13  9]]
-----------
round: 32
-----------
[[ 8  9 10 11]
 [12 13 14 15]
 [ 2  0  4  7]
 [ 6  3  5  1]]
```

Bibliography

[1] Dakshi Agrawal, Josyula R. Rao, and Pankaj Rohatgi. Multi-channel attacks. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 2–16, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45238-6.

[2] Ralph Ankele, Subhadeep Banik, Avik Chakraborti, Eik List, Florian Mendel, Siang Meng Sim, and Gaoli Wang. Related-key impossible-differential attack on reduced-round skinny. Cryptology ePrint Archive, Report 2016/1127, 2016. `https://eprint.iacr.org/2016/1127`.

[3] Ralph Ankele, Christoph Dobraunig, Jian Guo, Eran Lambooij, Gregor Leander, and Yosuke Todo. Zero-correlation attacks on tweakable block ciphers with linear tweakey expansion. Cryptology ePrint Archive, Report 2019/185, 2019. `https://eprint.iacr.org/2019/185`.

[4] Elaine B. Barker and A. Roginsky. Sp 800-131a. transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. 2011.

[5] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. `https://eprint.iacr.org/2013/404`.

[6] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 123–153, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53008-5.

[7] Luk Bettale, Emmanuelle Dottax, Laurie Genelle, and Franck Rondepierre. Differential power analysis of hmac sha-2 in the hamming weight model.

[8] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES*, 2002.

[9] Joan Daemen and Vincent Rijmen. The design of rijndael: Aes - the advanced encryption standard. 2002.

[10] Orr Dunkelman, Senyang Huang, Eran Lambooij, and Stav Perle. Single tweakey cryptanalysis of reduced-round skinny-64. Cryptology ePrint Archive, Report 2020/689, 2020. https://eprint.iacr.org/2020/689.

[11] J. Ge, Y. Xu, R. Liu, E. Si, N. Shang, and A. Wang. Power attack and protected implementation on lightweight block cipher skinny. In *2018 13th Asia Joint Conference on Information Security (AsiaJCIS)*, pages 69–74, 2018.

[12] Y. Han, X. Zou, Z. Liu, and Y. Chen. Improved differential power analysis attacks on aes hardware implementations. In *2007 International Conference on Wireless Communications, Networking and Mobile Computing*, pages 2230–2233, 2007.

[13] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Tweaks and keys for block ciphers: The tweakey framework. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 274–288, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45608-8.

[14] Jérémy Jean. Aes round function, February 2015.
URL: https://www.iacr.org/authors/tikz/.

[15] Jérémy Jean. Skinny tweakey schedule, May 2016.
URL: https://www.iacr.org/authors/tikz/.

[16] Jérémy Jean. Skinny round function, May 2016.
URL: https://www.iacr.org/authors/tikz/.

[17] S.M. Kay. *Fundamentals of Statistical Signal Processing: Detection theory*. Fundamentals of Statistical Si. Prentice-Hall PTR, 1998. ISBN 9780133457117.
URL: https://books.google.no/books?id=vA9LAQAAIAAJ.

[18] McKay Kerry, Bassham Lawrence, Turan Meltem Sönmez, and Mouha Nicky. Report on lightweight cryptography. 03 2017. doi: 10.6028/NIST.IR.8114. https://csrc.nist.gov/publications/detail/nistir/8114/final.

[19] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48405-9.

[20] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr 2011. ISSN 2190-8516. doi: 10.1007/s13389-011-0006-y.
URL: https://doi.org/10.1007/s13389-011-0006-y.

[21] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. *J. Cryptol.*, 24(3):588–613, July 2011. ISSN 0933-2790. doi: 10.1007/s00145-010-9073-y. https://doi.org/10.1007/s00145-010-9073-y.

[22] Guozhen Liu, Mohona Ghosh, and Ling Song. Security analysis of skinny under related-tweakey settings. Cryptology ePrint Archive, Report 2016/1108, 2016. https://eprint.iacr.org/2016/1108.

[23] Owen Lo, William J. Buchanan, and Douglas Carson. Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa). *Journal of Cyber Security Technology*, 1(2):88–107, 2017. doi: 10.1080/23742917.2016.1231523.
URL: https://doi.org/10.1080/23742917.2016.1231523.

[24] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. 01 2007. ISBN 978-0-387-30857-9. doi: 10.1007/978-0-387-38162-6.

[25] Stefan Mangard, Elisabeth Oswald, and Francois-Xavier Standaert. One for all - all for one: Unifying standard dpa attacks. Cryptology ePrint Archive, Report 2009/449, 2009. https://eprint.iacr.org/2009/449.

[26] Calvin McCoy. skinny cipher, 2016.
URL: https://github.com/inmcm/skinny_cipher.

[27] Sadegh Sadeghi, Tahere Mohammadi, and Nasour Bagheri. Cryptanalysis of reduced round skinny block cipher. Cryptology ePrint Archive, Report 2016/1120, 2016. https://eprint.iacr.org/2016/1120.

[28] Werner Schindler, Kerstin Lemke, and Christof Paar. Paar: A stochastic model for differential side channel cryptanalysis. In *Cryptographic Hardware and Embedded Systems — CHES 2005, Springer, LNCS 3659*, pages 30–46. Springer, 2005.

[29] Adrian Thillard, Emmanuel Prouff, and Thomas Roche. Success through confidence: Evaluating the effectiveness of a side-channel attack. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 21–36, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40349-1.

[30] D. Yang, W. Qi, and H. Chen. Impossible differential attacks on the skinny family of block ciphers. *IET Information Security*, 11(6):377–385, 2017.