Some Applications of Hamming Weight Correlations

Fatih Balli, Andrea Caforio and Subhadeep Banik

LASEC, Ecole Polytechnique Fédérale de Lausanne, Switzerland
{fatih.balli,andrea.caforio,subhadeep.banik}@epfl.ch

Abstract. It is a well-known fact that the power consumption during certain stages of a cryptographic algorithm exhibits a strong correlation with the Hamming Weight of its underlying variables. This phenomenon has been widely exploited in the cryptographic literature in various attacks targeting a broad range of schemes such as block ciphers or public-key cryptosystems. A common way of breaking this correlation is through the inclusion of countermeasures involving additional randomness into the computation in the form of hidden (undisclosed) component functions or masking strategies that complicate the inference of any sensitive information from the gathered power traces.

In this work, we revisit the tight correlation between the Hamming Weight and the observed power consumption of an algorithm and demonstrate, in the first part, a practical reverse-engineering attack of proprietary AES-like constructions with secret internal components like the SubBytes, MixColumns and ShiftRows functions. This approach is used in some commercial products such as the Dynamic Encryption package from the communication services provider Dencrypt as an extra layer of security. We recover the encryption key alongside the hidden substitution and permutation layer as well as the MixColumns matrix on both 8-bit and 32-bit architectures.

In a second effort, we shift our attention to a masked implementation of AES, specifically the secAES proposal put forward by the French National Cybersecurity Agency (ANSSI) that concisely combines several side-channel countermeasure techniques. We show its insecurity in a novel side-channel-assisted statistical key-recovery attack that only necessitates a few hundreds of collected power traces.

Keywords: Block Cipher · Side-Channel · See-in-the-Middle · DPA · AES · Reverse Engineering · Microcontroller · Masking · Hamming Weight

1 Introduction

Over the past few years, the field of side-channel-assisted cryptanalysis has evolved into an intricate spectrum that ranges from attacks based on power measurements to the assessment of leakages through electromagnetic emission as well as timing-based procedures and advanced pattern matching strategies that deploy machine learning algorithms. The common ground, however, remains in the fact that such leakages are only observable at the implementation level (both in hardware and software) and are usually not accounted for during the design phase of a cryptographic algorithm. Among these techniques, differential power analysis (DPA) [KJJ99] stands as one of the most destructive attack vector, as it is able to dismantle implementations of cryptographic algorithms with little effort. Contrary to other power analysis strategies, in DPA, an attacker looks for leakages in the differential power trace obtained by running the target device with two or more different inputs and computing the difference in obtained power traces. Furthermore, in order to counter the

inevitable noise that comes with electromagnetic measurements, it is often necessary to average the power traces for a given input over many repetitions.

In recent efforts, differential power analysis was coupled with conventional cryptanalysis techniques that amplified its effectiveness, especially against constructions based on shift registers. Dobraunig et al. [DEKM17] showed that by controlling differences in the initialization vector of the re-keying scheme Keymill [TRS16], it becomes feasible to infer relations between neighboring bits in the state register using the accompanying power differences. The attack was later extended and successfully applied by Sim et al. [SJB21] on an improved variant of Keymill. The premise of combining conventional cryptanalysis with side-channel observations was later brought into the realm of block ciphers by Bhasin et al. [BBH+19].

Reverse Engineering. Drawing conclusions from side-channel observations becomes invariably harder when the construction in question is not fully disclosed. Kerckhoffs's principle states that any cryptosystem should be secure even if everything about the system, except the key, is public knowledge. This concept is widely embraced by cryptographers, however *security through obscurity* remains as a tempting path to follow in industry. In this rather unconventional approach, the security is sometimes expected through obscurity, and undisclosed proprietary cryptographic algorithms are still used in civil applications, e.g. GSM or Pay-TV systems, and in diplomatic or military domains. This is where side-channels can play a major role in unlocking the hidden structures in secret cryptosystems.

The first known use of side-channels to reverse-engineer was the case of the A3/8 algorithm used in GSM [Nov03]. The attack by Novak reveals the content of one of the two substitution tables, which are intended to be kept as secret, used for authentication and key agreement protocol of GSM. This was improved by Clavier to the recovery of both the tables [Cla07]. In a related work, Clavier et al. [CIW13] presented a complete reverse engineering of AES-like secret ciphers, which shared the same core structure of AES-128, but used secret SubBytes, ShiftRows and MixColumns functions instead. Developed independently around the same time, Rivain and Roche [RR13], proposed a generic attack which applies to a general class of secret substitution-permutation (SPN) ciphers, showing that this line of attack works beyond AES cipher. Note that none of the previous work demonstrate attacks in practice, but their results are based only on theoretical simulations. The first practical attack was presented by Jap and Bhasin [JB20]. The authors tried to recover the 256 entries of a secret 8-bit S-box implemented on an Atmel AT-mega328P microprocessor mounted on Arduino UNO board. The authors were able to recover only 159 of the 256 entries successfully.

A common method of using power measurements to reverse-engineer is by using the concept of side-channel collisions, which builds on the premise that the same signals (values) tend to consume the same amount of power. The attacker is required to distinguish if values processed at two instances given algorithm are the same, assuming of course that the power consumption patterns are the same, and deduce the structures of internal components (e.g., entries of unknown lookup tables) on the basis of this. This is difficult in practice, because almost all side channel measurements are corrupted with random additive noise, which is why it is very important to determine the efficacy of side-channel reverse engineering on real-world platforms.

One of the current commercial products that uses secret components in the encryption process is *Dynamic Encryption* [Knu14, Knu20], which is a proprietary algorithm used by the Danish company Dencrypt. The technique extends standard encryption algorithms like AES-256 by a few additional rounds. In the additional rounds however, secret S-boxes are used instead of the standard AES S-box. The technique is used in many of the company's products like *Dencrypt Talk* and *Dencrypt Message* [den21]. This method has been proven hard to break in a classical sense (i.e., when the attacker only has access to plaintext-ciphertext pairs and no side-channel information) [TKKL15]. However, the side channel

security of the above method is still an open question.

Masking. Masking schemes are dedicated countermeasures that aim to thwart side-channel attacks by introducing additional randomness into the cipher state. Akin to the secret-sharing methodology, masking technique partitions an intermediate secret variable x during the computation into n shares such that the knowledge of up to $n-1$ shares does not enable an attacker to infer any information about x, i.e., x is statistically independent from the observed $n-1$ shares. Since the introduction of masking as a side-channel countermeasure in 1999 [CJRR99], there has been a continuous stream of refinements and improvements such as threshold implementations that are targeted against the inherent glitches occurring in hardware implementations [NRR06] and affine masking schemes against high-order side-channel analysis [von01, FMPR11].

In this paper, we scrutinize the security of a masked AES implementation that combines several side-channel countermeasures against a novel side-channel-based statistical attack.

1.1 Contributions

In first part of this paper, we demonstrate how to practically reverse-engineer the complete round function of AES-like SPN ciphers whose internal round functions are kept as secret besides the key. We shall assume that the adversary has black box access to the 8/32-bit microcontroller, programmed with the said SPN cipher implementation, with the ability to query any plaintext and collect its corresponding power trace of encryption with a single unknown key. The underlying assumptions of the algebraic structure of the unknown cipher are as follows:

1. The construction exhibits the same order of operations as AES. The state is of size 128 bits arranged in a 4×4 byte matrix.
2. The substitution box is an unknown bijective function S^*: $\{0,1\}^8 \rightarrow \{0,1\}^8$.
3. The ShiftRows procedure is an unknown permutation $\pi \in \mathcal{S}(16)$, where $\mathcal{S}(16)$ is the permutation group on 16 elements. We assume that the bytes of the state are permuted in a fashion that the byte at position i moves to $\pi(i)$ after the execution of the permutation layer, i.e. $i \rightarrow \pi(i)$.
4. The MixColumns matrix is an unknown invertible circulant matrix of form

$$M = \begin{bmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{bmatrix} \quad M^{-1} = \begin{bmatrix} e & f & g & h \\ h & e & f & g \\ g & h & e & f \\ f & g & h & e \end{bmatrix}, \tag{1}$$

   where $a, b, c, d, e, f, g, h \in GF(2^8) \setminus \{0\}$.

In the second part, we turn our focus to a *secure* AES software implementation, whose security relies on affine masking. We propose a practical side-channel-assisted key-recovery attack against the secAES [BLPR] proposal by the French National Cybersecurity Agency (ANSSI) for 8-bit Atmel microcontrollers. Briefly, the key idea behind this implementation can be summarized in two points:

1. During the execution of each layer, the order of execution among 16 bytes are shuffled, where the shuffling permutation is dependent on the mask. This means that any cryptanalytic advance that attempts to leverage difference in power traces by introducing a difference in one of the plaintext bytes is rendered ineffective. This is because the order of byte accesses is randomized in the time axis, and as such the time of access of any particular plaintext byte is different across two different runs. The corresponding power traces do not necessarily align in time. Thus, it is not possible to use the attack introduced by Bhasin et al. [BBH+19].

2. The implementation furthermore employs affine masking to create two shares of each state byte, therefore a traditional CPA-based approaches are difficult to apply.

1.2 Outline

After this introductory section, some preliminaries such as the recently proposed See-in-the-Middle attack are briefly reiterated in Section 2. Afterwards, in Section 3, we detail our reverse engineering procedure on hidden AES-like ciphers. Section 4 then contains the write-up of the attack against the masked secAES implementations. Finally, the paper is concluded in Section 5.

2 Preliminaries

Before diving into the two main parts of this text, we describe some preliminary information on which the coming sections are based.

2.1 See-in-the-Middle

This novel side-channel analysis approach [BBH+19] can be categorized as a side-channel assisted differential plaintext attack (SCADPA). It exploits the fact that the occurrence of specific differential propagations in block cipher implementations are observable in differential power traces. For example in AES, the well-known convergence property (see Figure 1) of a diagonal plaintext differential leads to a significant leakage in the differential power traces.

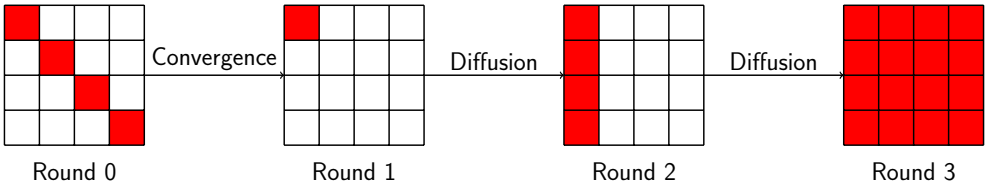
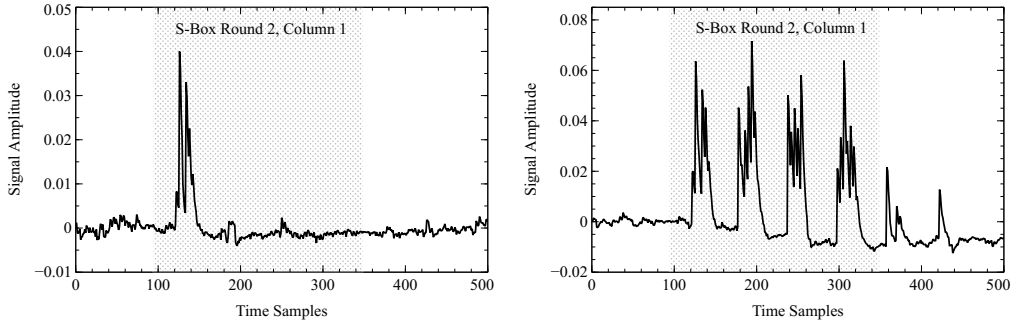


Figure 1: Three-round converging and diffusing AES differential propagation. Depicted is the block cipher state at the end of each round. Active bytes are marked in red.

The convergence in round 1 occurs with some probability and may affect any byte in the first column. Depending on which byte is active, the differential propagates to a different column in the second round. In Figure 2, the difference power traces for encryption with two plaintexts are shown. The traces zoom into the region during the substitution layer on the first state column during the second round, showing that a convergence is easily detected. The authors proceed to show that on both 8-bit and 32-bit implementations the active column is readily visible. Hence, it possible to know whether a convergence has taken place or not for a given plaintext pair by just inspecting the difference of the power trace. Then by guessing the actual differential in round 1 after the MixColumns, one can solve backwards to find a solution for the entire round key in the corresponding diagonal. The authors showed that on average, around $2^{11.5}$ plaintexts need to be queried until a convergence is observed, and thus finding the entire round key requires $\mathcal{O}(2^{11.5})$ side channel observations using the same number of plaintexts.

2.2 Setup

The contributions in Section 3 are validated experimentally on an 8-bit ATXmega128D4-AU (AVR instruction set) board and a 32-bit STM32F3 (ARM Cortex-M4) board. Both



(a) Convergence (b) Non-Convergence

Figure 2: Differential power trace (ATXmega128D4-AU) of the second round substitution layer of the first state column. Only one byte is active when a convergence occurs (a) whereas all four of them are active when there is no such occurrence (b).

microcontrollers belong to the most widely incorporated architectures in their respective domains. As the secAES implementation is 8-bit specific (written in AVR assembly language), the attack in Section 4 is validated on the ATXmega128D4-AU board. Both microcontrollers are mounted on a ChipWhisperer installation [OC14] that captures power traces through a softcore on a SPARTAN-6 FPGA that exposes an UART interface to the user as a means of communication. The target devices are clocked at a frequency of 7.38 MHz, effectively sampling four power traces per clock cycle, i.e., a sampling frequency of 29.52 MHz.

We used the avrcryptolib (8-bit) and TinyAES128C (32-bit) implementatios with randomized components in Section 3, whereas in Section 4, the actual secAES implementation was used [BLPR] .

3 Reverse Engineering AES-like Ciphers

Before diving into the two main parts of the reverse engineering attack, we reiterate a few side-channel phenomena that are directly linked to Hamming Weights of the underlying parameters.

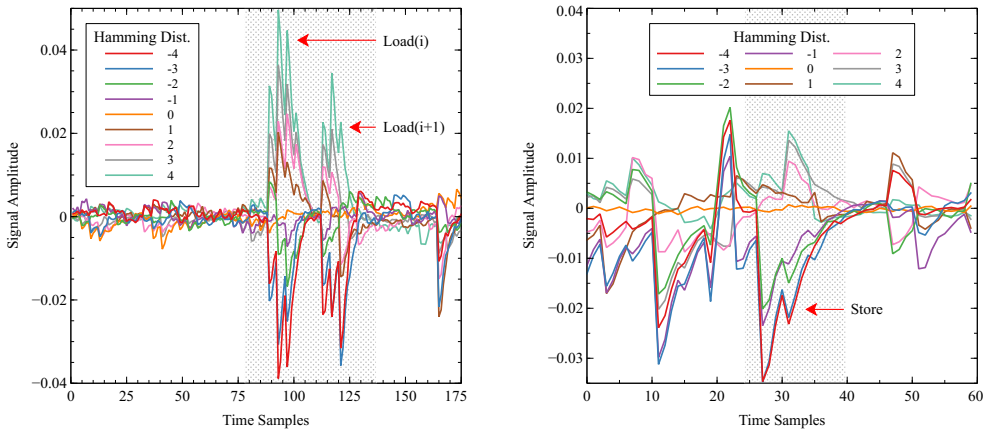
Definition 1 (Hamming Weight)**.** Given an n-bit element $x \in \mathbb{F}_n$, let $0 \leq W(x) \leq n$ be its Hamming Weight, i.e., the number of bits that are set to one in the binary representation of x.

Due to noise in power measurements, it is not possible to directly deduce the Hamming Weight for a particular value of interest, that is being updated inside the central processing unit of the microcontroller. By leveraging differential power analysis, it is nevertheless possible to find out the Hamming Weight of a particular value by the means of a different observation.

Definition 2 (Power Consumption)**.** Given a function $F : \mathbb{F}_m \mapsto \mathbb{F}_n$ that executes a sequence of instructions for an arbitrary $x \in \mathbb{F}_m$, we denote by $P(F(x))$ the power consumption of this transfer and by $\overline{P}(F(x))$ its average over multiple repetitions.

In Figure 3, we show that if $W(F(x)) > W(F(y))$ then $\overline{P}(F(x)) > \overline{P}(F(y))$ for two different $x, y \in \mathbb{F}_{256}$. More precisely, given a fixed element x with Hamming Weight $W(F(x)) = w$, we plot $\overline{P}(F(x)) - \overline{P}(F(y))$ for nine elements y with pairwise different Hamming Weights. Evidently, we note that a Hamming Weight change in the output of F is easily spotted. In general terms, if $\overline{P}(F(x))$ for an n-bit element $x \in \mathbb{F}_n$ is compared

against $\overline{P}(F(y))$ for all other elements $y \in \mathbb{F}_n$ then the obtained plot can only exhibit one of $n+1$ different arrangements of power drops and spikes.



(a) Key Whitening (b) Substitution Box

Figure 3: Averaged differential power traces (ATXmega128D4-AU) of one state byte during the initial key whitening phase (a) and at the end of the substitution layer (b)[1].

Definition 3 (Hamming Scale)**.** Consider a n-bit element $x \in \mathbb{F}_n$ with $W(x) = w$. We define the Hamming Scale H of x as

$$H(x) = \sum_{y \in \mathbb{F}_n} 1_{W(y)>W(x)} - 1_{W(y)<W(x)}.$$

In other words, $H(x)$ denotes the difference between the number elements with a higher Hamming Weight and those with a lower one. The Hamming Scale for $n = 8$ is given in Table 1, which tabulates the value of the function $H(x)$ whenever $W(x)$ is a given constant.

Table 1: Hamming Scale of an 8-bit element space according to Definition 3.

$W(x)$	0	1	2	3	4	5	6	7	8
$H(x)$	255	246	210	126	0	-126	-210	-246	-255

The existence of the observed phenomenon together with the Hamming Scale from Definition 3 immediately yields an algorithm to recover the Hamming Weight of $F(x)$, which is given in Algorithm 1. In terms of complexity, this algorithm makes $2^n \times \alpha$ function calls, where α is the number of repetitions per encryption that are required to attain a stable averaged power trace[2]. Note that knowledge of $W(F(x))$ does not necessarily mean that $F(x)$ is uniquely recovered. For example, if $W(F(x)) = 4$ and $n = 8$, then there remains $\binom{8}{4}$ candidates. However, 0 and $2^n - 1$ within $\mathbb{F}_n$ are the only elements with Hamming Weight 0 and n respectively.

Algorithm 1, can be converted into a *Hill-Climbing* routine with the same complexity that recovers the element x such that $W(x) = 0$ (or analogously $W(x) = n$). Its pseudocode is detailed in Algorithm 2.

[1]This observation also holds for 32-bit processors like an ARM Cortex-M4. In Appendix A, we give the corresponding plots for the STM32F3 microcontroller.

[2]On both 8-bit and 32-bit systems used in the paper, $\alpha \approx 10$ was sufficient for a stable average

Algorithm 1 Generic n-Bit Hamming Weight Recovery

Input: $x \in \mathbb{F}_m$, $F : \mathbb{F}_m \mapsto \mathbb{F}_n$
Output: $W(F(x))$
▷ *Execute the operation $F(x)$ of a n-bit value x and obtain an averaged power trace. In order to attain a stable average, several hundreds repetitions may be required.*
$p \leftarrow \overline{P}(F(x))$
$h \leftarrow 0$
for $y \in \mathbb{F}_m \setminus \{x\}$ **do**
▷ *Distinguish whether the averaged differential power trace incurs a power spike or drop and adjust the Hamming Scale counter h accordingly. Do nothing when the power difference is below some threshold ϵ representing elements of equal Hamming Weight.*
    $d \leftarrow \overline{P}(F(y)) - p$
    if $|d| < \epsilon$ **then continue**
    if $d > 0$ **then** $h \leftarrow h + 1$
    else $h \leftarrow h - 1$
▷ *Compare the calculated scale counter h with the Hamming Scale table from Definition 3 and return the corresponding Hamming Weight i.e., if $|H(x) - h|$ is minimal, then $W(x)$ is the best estimation for the Hamming Weight of $F(x)$.*
return $\arg\min_x |h - H(x)|$

Algorithm 2 Generic Zero Hamming Weight Recovery

Input: $F : \mathbb{F}_m \mapsto \mathbb{F}_n$
Output: $x \in \mathbb{F}_m$ such that $W(F(x)) = 0$
$x \leftarrow_U \mathbb{F}_m$
$p \leftarrow \overline{P}(F(x))$
for $y \in \mathbb{F}_m \setminus \{x\}$ **do**
    $d \leftarrow \overline{P}(F(y)) - p$
    if $|d| < \epsilon$ **then continue**
    if $d < 0$ **then**
        $x \leftarrow y$
        continue
return x

3.1 Finding the Key

The first key whitening operation is invariably carried out by fetching the corresponding plaintext and key bytes from the memory to some CPU register via some form of LOAD instruction. After XOR of the state and the key bytes, the result is stored back to memory with STORE instruction. In most instances, they would be performed by the code snippet (running in a loop 16 times) as follows:

```
LD  R1, [ADDR PT]  ; load plaintext byte
LD  R2, [ADDR KEY] ; load key byte
EOR R1, R2         ; execute exclusive-or
ST  R1, [ADDR PT]  ; store back to memory
```

Consider what happens to the differential power trace during the execution of these 4 instructions, when it is queried with plaintexts $PT \oplus PT' = \Delta \cdot \mathbf{e}_i$, where $\mathbf{e}_i$ is the i-th unit vector over bytes, i.e. $\mathbf{e}_0 = [\texttt{0x01}, \texttt{0x00}, \texttt{0x00}, \ldots, \texttt{0x00}]$ and $\mathbf{e}_1 = [\texttt{0x00}, \texttt{0x01}, \texttt{0x00}, \ldots, \texttt{0x00}]$ etc. Note that the difference in the power consumption comes from the changes in the

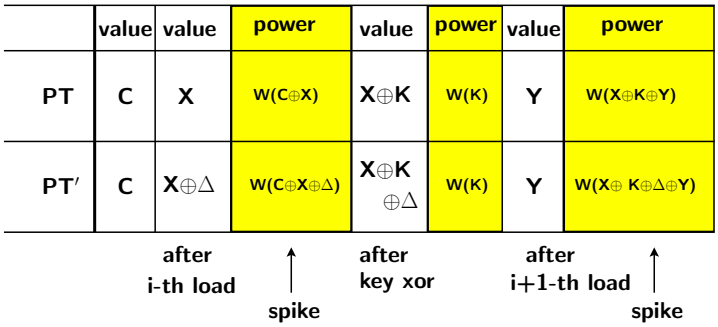


Figure 4: Values in the register `r1` during the i and $i+1$-th load cycles

register `r1` which accommodates the differential state byte during the 2 queries. Let us observe the change in the value of the registers.

As can be seen in Figure 4, the sequence of loads during the i, $(i+1)$-th load cycles give rise to 2 peaks in the differential (provided the noise has been filtered out by averaging sufficient samples). This double peak can be observed in the differential trace shown in Figure 3(a). During the $(i+1)$-th load, the register gets overwritten with the constant $(i+1)$-th plaintext byte Y and thus the amplitude of the peak in the differential power trace is proportional to the difference of the Hamming Weights $W(X \oplus K \oplus Y) - W(X \oplus K \oplus Y \oplus \Delta)$ where K is the i-th key byte. Let this time instant be τ, and the amplitude of the averaged power trace at τ for when the i-th plaintext byte has value x be denoted as $\overline{P}_{x,\tau}$. We have established that $\overline{P}_{x_1,\tau} - \overline{P}_{x_2,\tau} = c \cdot (W(x_1 \oplus K \oplus Y) - W(x_2 \oplus K \oplus Y))$, where c is some constant. Since Y is a plaintext byte under the control of the adversary, we can set it to zero which gives $\overline{P}_{x_1,\tau} - P_{x_2,\tau} = c \cdot (W(x_1 \oplus K) - W(x_2 \oplus K))$. The adversary can collect the set $\mathbb{P} = \{\overline{P}_{i,\tau},\ \forall\ i \in \mathrm{GF}(2^8)\}$. Note that $\overline{P}_{i,\tau} - \overline{P}_{K,\tau} = c \cdot W(i \oplus K)$. Thus the strategy of the adversary is as follows:

1. For each $x \in \mathrm{GF}(2^8)$
   - Compare the scalar $\overline{P}_{x,\tau}$ with the set $\mathbb{P}$, i.e compute the vector V_x whose i-th element is $\overline{P}_{i,\tau} - \overline{P}_{x,\tau}$.
   - Compute the vector H_x whose i-th element is $W(x \oplus i)$.
   - Find correlation coefficient ρ_x between H_x and V_x.

By our previous analysis we already know that for $x = K$, we will obtain a very high correlation coefficient, which gives us the value of K.

Complexity: Recovering each byte of the whitening key requires $2^8 \times \alpha$ encryptions using Algorithm 2, hence the total number of plaintexts one needs to query to recover all 16 bytes of the key is around $2^{12} \times \alpha$.

3.2 Finding π

Of the three secret components in the design, finding π is probably the simplest given a differential power trace of two encryptions, provided the trace has been averaged over an adequate number of runs to get rid of the noise (which is assumed to be zero mean). The process is as follows: take two plaintexts PT, PT_i such that $PT \oplus PT_i = \mathbf{e}_i$. The adversary obtains the differential power trace of PT, PT_i (averaged over a few runs to cancel noise). Ideally the simplest way to guess $\pi(i)$ would be to observe the ShiftRows region of the differential trace and deduce the relative position in the time axis of the single peak in the trace for all $i \in [0, 15]$. However this method is not always reliable for the following reasons:

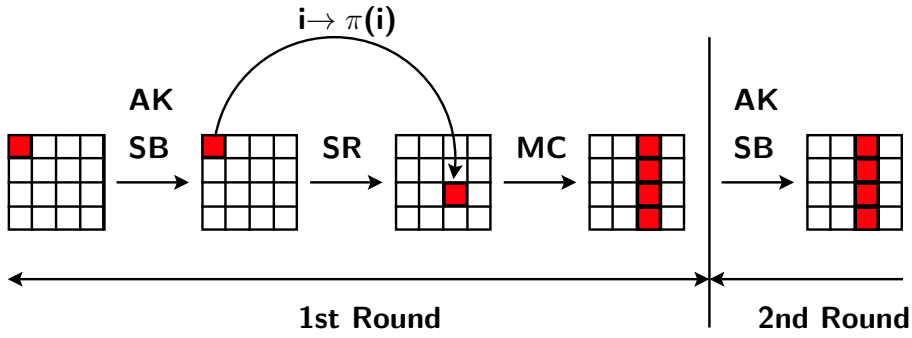


Figure 5: In this example π maps $i = 0$ to column 2 which manifests as 4 spikes in the differential trace after 2nd round substitution layer.

1. Depending on the implementation, the ShiftRows and MixColumns operation may be merged together so that a distinct region in the trace segregating the ShiftRows operation may not be deducible.
2. Even if the ShiftRows region is demarcated, any particular implementation may swap bytes in a specific order depending on the algebraic description of π.
3. The active position i may be a fixed point of π, due to which no operation the i-th byte in the ShiftRows operation is necessary.

Instead, we will look to observe the peaks in the differential trace after the 2nd round substitution layer. If the permutation function π is such that i is mapped to the j-th column (for any $0 \leq j \leq 3$), i.e. $\pi(i) \in \{4j, 4j+1, 4j+2, 4j+3\}$ then after the 1st round MixColumns the j-th column becomes active, which shows up as a sequence of 4 spikes after the 2nd round substitution layer in the differential trace. The relative order in the time axis of these peaks tells us the value of j such that $4j \leq \pi(i) \leq 4j+3$, for each i. In other words, we are able to deduce which column each byte is mapped to after the ShiftRows operation.

Complexity. If α is the number of plaintexts we need to query for the de-noised trace of one plaintext, the above procedure can pe simply run 16 times for differential at each $\mathbf{e}_i$ and so we need only around $17 \times \alpha$ plaintext queries.

3.3 Finding M

Note that we do not yet have the exact description of π, but we aim to find the precise π and the matrix M together in this part. Let u_i be such that $\pi(u_i) = i, \forall i \in [0, 15]$, i.e., $u_{4i}, u_{4i+1}, u_{4i+2}, u_{4i+3}$ are the bytes in the state that get mapped to the i-th column after the first round ShiftRows. We have already determined the values of u_0, u_1, u_2, u_3 upto a permutation of the 4 elements. As such, this means we have narrowed down the exact values of u_i for $i = 0 \rightarrow 3$ to a set of $4! = 24$ candidates. As such the following subroutine is repeated 24 times for each of the possible values of the u_i.

Given the unknown matrix from (1), we proceed in multiple steps with differentials on the certain specific locations after the substitution layer of the first round. Note that our Hamming Scale algorithm (Algorithm 1) allows us to deduce (after de-noising) the Hamming Weight of each byte computed after the substitution layer. We take additional advantage of the fact that there is only a unique byte value with Hamming Weight 8, i.e. `0xFF` or 255. We can filter all plaintext pairs that result in the difference $x \cdot \mathbf{e}_i$ (after the first round SubBytes), for all $i \in [0, 15]$ and for some byte value x, by simply examining spikes in the differential trace after the first round substitution layer, and the Hamming

Scale analysis allows us to determine $W(x)$. Since there is only one byte (255) with Hamming Weight 8, we can also determine plaintext pairs that generate differences of type $255 \cdot \mathbf{e}_i$. Since the there is no inter-mixing of bytes after the first round it is also relatively straightforward to filter plaintext pairs that produce differences of type $\sum x_i \cdot \mathbf{e}_i$ (after 1st round substitution layer), if we already know plaintext pairs that generate the individual $x_i \cdot \mathbf{e}_i$ differences.

Example 1. If PT=`[00,00,00,00, 00,00,00,00, 00,00,00,00, 00,00,00,00]` and PT'= `[23,00,00,00, 00,00,00,00, 00,00,00,00, 00,00,00,00]` are such that after the first substitution layer the difference of the internal state is `FF`$\cdot\mathbf{e}_0$, and PT''= `[00,00,A1,00, 00,00,00,00, 00,00,00,00, 00,00,00,00]` is such that the corresponding difference of state (between PT, PT'') is `05`$\cdot\mathbf{e}_2$, then it is easy to see that $PT''' =$ `[23,00,A1,00, 00,00,00,00, 00,00,00,00, 00,00,00,00]` and PT produce the difference `FF`$\cdot\mathbf{e}_0$ + `05`$\cdot\mathbf{e}_2$. This is true, even if the key and S-box functions are unknown.

Denote by Δ the four-tuple $[\delta_0, \delta_1, \delta_2, \delta_3]$. Note that we can recover such differences up to the their Hamming Weight using Algorithm 1. We will try to concentrate of differential plaintext pairs that produce differences of the form $\sum_{i=0}^{3} \delta_i \cdot \mathbf{e}_{u_i}$, so that they get mapped to the first column after the first round ShiftRows. The idea is to observe in the differential trace after the 2nd substitution layer at least one inactive byte (which requires one inactive byte after the first round MixColumns). This also results in a completely inactive column after the 3rd round SubBytes as seen in Figure 6. This can be distinguished in the corresponding regions in the differential trace by a lack of spikes. In general we try to distinguish the following special cases.

Step 1: $\delta_0 = 255,\ \delta_1 = x_1,\ \delta_2 = 0,\ \delta_3 = 0$, for some $w_1 \in \mathrm{GF}(2^8)$ whose exact value is only known up to its Hamming Weight, i.e., $W(x_1) = w_1$. Note that as per the previous example, an efficient way of getting such a difference is by first finding plaintext pairs $PT,\ PT'$ such that $PT \oplus PT' = y \cdot \mathbf{e}_{u_0}$ such that they yield a difference $255 \cdot \mathbf{e}_{u_0}$ after the first round SubBytes and then sequentially querying pairs $PT,\ PT''$ such that $PT \oplus PT'' = y \cdot \mathbf{e}_{u_0} \oplus x \cdot \mathbf{e}_{u_1}$ for all non-zero byte values x. After this, we make use of See-in-the-Middle observation in the second round and check whether the differential has caused the first byte after the 2nd SubBytes to be inactive and thus diffused to a single inactive column after the 3rd SubBytes. When the first byte after the 2nd SubBytes is inactive we have that the first byte of $M \cdot \Delta$ is zero which yields

$$255a \oplus x_1 b = 0 \quad \Rightarrow \quad 255a = x_1 b \tag{2}$$

If the first byte after the 2nd substitution layer is not inactive, we repeat with a different plaintext differential in the u_1 location until one is found.

Step 2: $\delta_0 = x_2,\ \delta_1 = 255,\ \delta_2 = 0,\ \delta_3 = 0$, for some $x_2 \in \mathrm{GF}(2^8)$. Again, only $W(x_2) = w_2$ is known. In this step, we now attempt to find such an w_2 such that it yields the same inactive column as in first step. In our case, this would be the first one, from which we then infer

$$255b \oplus x_2 a = 0 \quad \Rightarrow \quad 255b = x_2 a \tag{3}$$

Consequently, by combining (2) and (3), we get

$$x_1 x_2 = 255^2. \tag{4}$$

Step 3+4: The first two steps are repeated for other differentials $x_3, x_4 \in \mathrm{GF}(2^8)$ for which $W(x_3) = w_3$ and $W(x_4) = w_4$ such that the 2nd byte is inactive in the second round. When this happens we must have the 2nd byte of $M \cdot \Delta$ is 0 for both $\Delta = [255, x_3, 0, 0]$ and $\Delta = [x_4, 255, 0, 0]$. These yield

$$255d = x_3 a, \quad x_4 d = 255a. \tag{5}$$

Combining these equations as above we get:

$$x_3 x_4 = 255^2. \tag{6}$$

Step 5+6: Another repetition for differentials $x_5, x_6 \in \mathrm{GF}(2^8)$ with $W(x_5) = w_5$ and $W(x_6) = w_6$. If now the third byte is inactive, it yields

$$255c = x_5 d, \quad x_6 c = 255d. \tag{7}$$

Combining these, we get

$$x_5 x_6 = 255^2. \tag{8}$$

Step 7+8: A final repetition for which the fourth byte is inactive using the differentials $x_7, x_8 \in \mathrm{GF}(2^8)$ with $W(x_7) = w_7$ and $W(x_8) = w_8$, we get

$$255b = x_5 c, \quad x_6 b = 255c. \tag{9}$$

Combining these we get

$$x_7 x_8 = 255^2. \tag{10}$$

The combination of (2), (3), (5), (7), (9) additionally yields

$$x_1 x_3 x_5 x_7 = x_2 x_4 x_6 x_8 = 255^4. \tag{11}$$

Step 9+10: One can indeed other structures of plaintext differences to get inactive bytes after the 2nd round sub-bytes. For example an inactive 1st byte for $\Delta = [x_9, 0, 255, 0]$ and an inactive 3rd byte for $\Delta = [255, 0, x_{10}, 0]$ yields

$$x_9 a = 255c, \quad 255a = x_{10} c. \tag{12}$$

which yields

$$x_9 x_{10} = 255^2. \tag{13}$$

The combination of (2), (3), (9), (12) gives:

$$x_1 x_7 x_9 = x_2 x_8 x_{10} = 255^3 \tag{14}$$

A schematic of the MixColumns recovery algorithm is depicted in Figure 2. By using all the obtained equations together with the known Hamming Weights it becomes possible to narrow down the number of circulant candidate matrices defined by $[a, b, c, d]$ to around 2^8. This is how:

1. Note that the only information the adversary has is $w_i = W(x_i)$ for all $i \in [1, 10]$ and the fact that the x_i's satisfy Equations (4), (6), (8), (10), (11), (13), (14).
2. For all bytes b_i such that $W(b_i) = w_i$ for $i \in [1, 10]$
   - If the b_i satisfy Equations (4), (6), (8), (10), (11), (13), (14), then retain them as candidates for x_i.
   - If not move to the next set of b_i.

If the initial values of u_0, u_1, u_2, u_3 are correctly guessed, then the filters defined by the above equations are enough to obtain a unique value for the x_i's. From here we can get $2^8 - 1$ solutions for M as follows: we freely choose a to be any non-zero byte. Then b, c, d are obtained from above as $b = 255^{-1} \cdot x_2 \cdot a$, $c = 255^{-1} \cdot x_9 \cdot a$ and $d = 255^{-1} \cdot x_3 \cdot a$.

For the 23 incorrect initial guesses of u_0, u_1, u_2, u_3 the situation is slightly more complicated. For exactly 20 other incorrect guesses the above algorithm returns no solution

which implies that our guess was incorrect. However for the remaining 3 guesses in which the starting u_0, u_1, u_2, u_3 are rotations of the correct guess, the algorithm also yields a unique solution. The solutions in the latter cases are row rotated versions of M in the opposite direction.

To understand why this happens, let P_t be the 4×4 permutation matrix that rotates the a column vector by t locations (for $0 \leq t \leq 3$) in some direction. Let $\mathbf{v}_i$ be the i-th column after the π operation. Note that if M is a circulant matrix then $M \cdot P_t^{-1}$ is also a circulant matrix, in which the rows of M are rotated t locations in the opposite direction. Since $M \cdot \mathbf{v}_i = \left(M \cdot P_t^{-1}\right) \cdot (P_t \cdot \mathbf{v}_i)$, this explains that any starting guess of u_0, u_1, u_2, u_3 that is a rotation of the correct guess also yields a set of solutions for the matrix M that is a row-rotated version of the correct matrix. However since $M \cdot P_\$$ is not a circulant matrix for any random permutation matrix $P_\$$ that does not strictly rotate, then trying to run the above algorithm would yield a contradiction in some stage in the 10-stage filter and hence yield no solutions.

The next question is then how to recover t and P_t? The answer is, it is not necessary. We repeat the above algorithm to for the three other columns of the state, i.e. all possible guesses of $U_i = [u_{4i}, u_{4i+1}, u_{4i+2}, u_{4i+3}] \in [4i, 4i+3]$ for $1 \leq i \leq 3$. For each column (which is to say for each i) we get 4 rotationally equivalent initial guesses that yield solutions for M. We first select the guesses for the 4 sets of initial guesses U_i that yield the same set of $2^8 - 1$ solutions for M up to multiplication by the free variable a. Let π be the permutation over $[0, 15]$ so formed by this guess, and let P_π be the 16×16 permutation matrix corresponding to π. Thereafter for any t, consider the block diagonal matrix $B_t = Diag[P_t, P_t, P_t, P_t]$, then it is easy to see that the block cipher consisting of the ShiftRows matrix $B_t \cdot P_\pi$ and MixColumns matrix $M \cdot P_t^{-1}$ are functionally equivalent for any t. For example, AES would be functionally the same if the matrix B_1 were applied to the state after ShiftRows and the MixColumns matrix changed to $Circ(1, 2, 3, 1)$. Thus any choice of t works equally well, and we recover the linear layer up to the choice of the free variable a.

Complexity. For each of the columns, the complexity can be queried as follows. To generate differentials of the form $255 \cdot \mathbf{e}_{u_0}$, $255 \cdot \mathbf{e}_{u_1}$ and $255 \cdot \mathbf{e}_{u_2}$ after the first SubBytes (using Algorithm 2) requires $3 \times \alpha \times 2^8$ queries. Thereafter, for generating the filter equations, we have to use differentials of type $\Delta = [255, x, 0, 0]$, $[x, 255, 0, 0]$, $[255, 0, x, 0]$, $[x, 0, 255, 0]$ which requires around $4 \times \alpha \times 2^8$ plaintext queries. The filtering for the 24 possibilities of the u_i can be done separately after generating the traces. Thus the total number of queries required in this part are $4 \times (7 \times \alpha \times 2^8) = 28 \times \alpha \times 2^8$.

3.4 Finding S^*

Ultimately, to recover the hidden s-box, we again limit ourselves to differentials that map to a single column after the ShiftRows operation. Therefore take the plaintext bytes pt_{u_i} and key bytes ky_{u_i} for $i \in [0, 3]$, i.e. those which map to the 0-th column after key addition, s-box layer and ShiftRows operation. For ease of notation, the corresponding plaintext bytes are denoted by $x_i = pt_{u_i}$ which are added to the recovered whitening key bytes $k_i = ky_{u_i}$. It is straightforward to derive that when only the *first* byte of the column is active after the MixColumns operation (as in Figure 1, i.e., all the other bytes in the column inactive), this only occurs for a specific set of differentials, namely

$$\begin{aligned}
S^*(x_0 \oplus k_0) + S^*(x_0 \oplus k_0 \oplus \delta_0) &= e\lambda \\
S^*(x_1 \oplus k_1) + S^*(x_1 \oplus k_1 \oplus \delta_1) &= h\lambda \\
S^*(x_2 \oplus k_2) + S^*(x_2 \oplus k_2 \oplus \delta_2) &= g\lambda \\
S^*(x_3 \oplus k_3) + S^*(x_3 \oplus k_3 \oplus \delta_3) &= f\lambda,
\end{aligned} \tag{15}$$

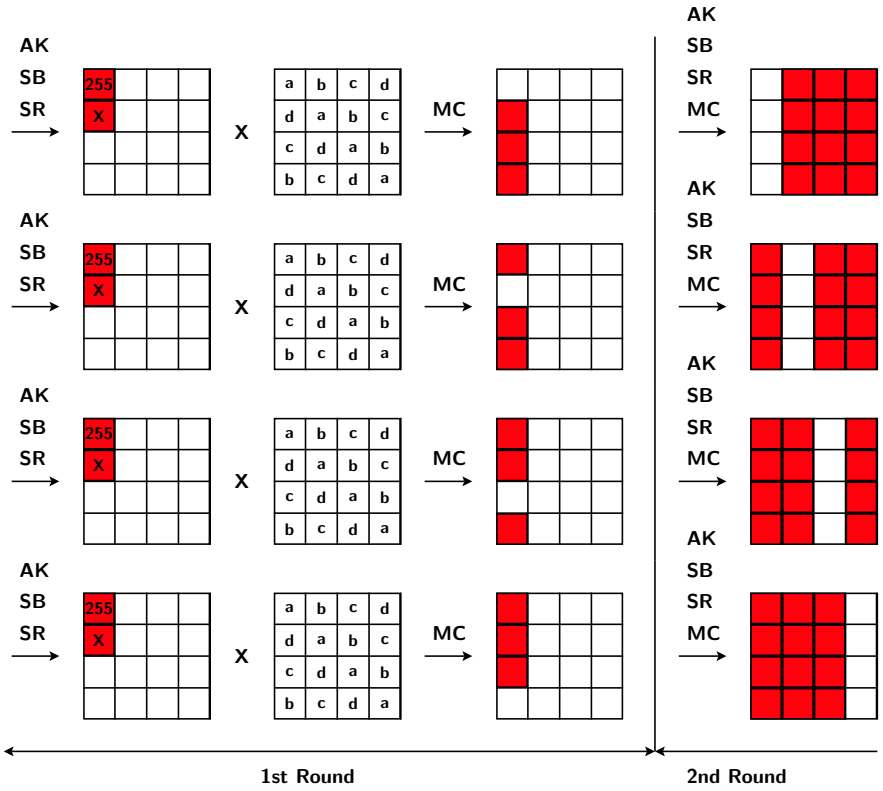


Figure 6: MixColumns recovery schematic. An inactive first column yields the equation $255a = bx$. For an inactive second column, we get $255b = cx$. Through an inactive third column, we receive $255c = dx$. Finally, an inactive fourth column yields $255d = ax$.

where $\lambda \in \mathrm{GF}(2^8)$ is any non-zero byte value, and as already mentioned e, f, g, h are the coefficients in the inverse MixColumns matrix. The first step of our recovery attack is concerned with finding a tuple $x_0, x_1, x_2, x_3 \in \mathrm{GF}(2^8)$ such that $S^*(x_0 \oplus k_0) = S^*(x_1 \oplus k_1) = S^*(x_2 \oplus k_2) = S^*(x_1 \oplus k_3) = 0$, which simplifies the system of equations to

$$\begin{aligned} S^*(x_0 \oplus k_0 \oplus \delta_0) &= e\lambda \\ S^*(x_1 \oplus k_1 \oplus \delta_1) &= h\lambda \\ S^*(x_2 \oplus k_2 \oplus \delta_2) &= g\lambda \\ S^*(x_3 \oplus k_3 \oplus \delta_3) &= f\lambda. \end{aligned} \tag{16}$$

Algorithm 2 together with the Hamming Scale observation after s-box can be used to find such a tuple.

We proceed to look for the occurrence of convergences using the See-in-the-Middle technique by varying the differentials $\delta_1, \delta_2, \delta_3, \delta_4$. Once it is found, we recover the Hamming Weight of $S(x_i + k_i + \delta_i)$ making use of Algorithm 1. Suppose that we receive the following tuple of Hamming Weights for the four equations

$$\begin{aligned} W(S^*(x_0 \oplus k_0 \oplus \delta_0)) &= w_0 \\ W(S^*(x_1 \oplus k_1 \oplus \delta_1)) &= w_1 \\ W(S^*(x_2 \oplus k_2 \oplus \delta_2)) &= w_2 \\ W(S^*(x_3 \oplus k_3 \oplus \delta_3)) &= w_3. \end{aligned} \tag{17}$$

Since at this point x_i, k_i, δ_i are known to the attacker, all he needs to fill up the 256

entries of S^* is some method to convert the weights w_i recovered above into actual values. However, the attacker knows that the actual values are related by Equation (16) which he can leverage as follows. The attacker pre-computes a table T whose λ-th entry is the tuple $T[\lambda] = [W(e\lambda), W(h\lambda), W(g\lambda), W(f\lambda)]$ for all $0 < \lambda < 256$. The attacker can infer the value of λ if $T[\lambda] = [w_0, w_1, w_2, w_3]$ for some table entry.

For random values of e, f, g, h, from computer simulations we have found that more than 200 entries of T are unique. If the fingerprint $[w_0, w_1, w_2, w_3]$ is a unique entry in the table, we recover four substitution table elements. Otherwise, we can repeat the procedure for a different differential. We were able to recover all S-box entries within a few repetitions of the above procedure.

Complexity. Recovering the zero-image in the first step such that $S^*(x_0 \oplus k_0) = S^*(x_1 \oplus k_1) = S^*(x_2 \oplus k_2) = S^*(x_3 \oplus k_3) = 0$ requires $4 \times 2^8 = 2^{10}$ encryptions. Afterwards, for each four elements of the s-box, the See-in-the-Middle convergence requires an additional $\alpha \times \beta \times 2^{11.5}$ encryptions where β is the reciprocal of the probability that a unique Hamming Weight fingerprint is found in the pre-computed table. Note that on average $2^{11.5}$ plaintexts are required to observe a See-in-the-Middle convergence on active byte as explained in Section 2.1. If the coefficients e, f, g, h are chosen uniformly at random then computer simulations show that $\beta \approx 1.3$. This step needs to be repeated $\frac{256}{4} = 64$ times to recover all the entries of S^*. Hence the number of total encryptions to recover the full substitution table for a given MixColumns matrix is $\alpha \times 2^{10} + \alpha \times \beta \times 2^{17.5}$.

3.5 Final Computation

Note that we have 255 solutions of the matrix M and hence M^{-1} and so above procedure has to be repeated 255 times to obtain 255 solutions of the M, S^*. Since every thing else has been recovered, thereafter using any known plaintext, ciphertext pair we can check if the given solution of M, S^* indeed produce the corresponding plaintext-ciphertext pair. With high probability 254 incorrect solutions of M, S^* are eliminated in this step and we successfully recover all the secret components of the block cipher.

4 secAES Implementation

The secAES implementation [BLPR] was proposed by Benadjila et al. and uses a combination of affine masking and shuffling to protect the key as well as the plaintext against side-channel attacks[3]. Briefly, this implementation relies on two techniques (note that $[n]$ denotes the set $\{0, 1, \ldots, n-1\}$ in this section):

- Each cipher state byte s_i is stored in the form of $a \times s_i \oplus m_i$ with masks $a \in \mathsf{GF}(2^8)$ and $m_i \in \{0,1\}^8$ for $i \in [16]$. Here, a is fixed throughout a single encryption call and is shared among 16 bytes, whereas each m_i is unique to particular byte [FMPR11].

- The order of execution among bytes for each layer is randomized according to some permutation $\pi \in \mathcal{S}(16)$. The permutation π is again fixed throughout the single encryption call, and is mostly shared between different layers such as AddRoundKey, SubBytes and ShiftRows [RPD09].

In the following section, we give a longer description of the implementation.

[3] We use Version 2 of the implementation that incorporates additional security measures that is available at https://github.com/ANSSI-FR/secAES-ATmega8515/tree/master/src/Version2.

4.1 Details of secAES

Inputs. secAES receives three sequences of bytes $P[i]$, $K[i]$ and $R[j]$ as input, for $i \in [16], j \in [18]$. Each respectively corresponds to plaintext, key and initial random mask.

Intermediate Variables. During the execution, the following intermediate variables are used:

- Three permutations $\pi_0, \pi_1, \pi_2 : [16] \rightarrow [16]$ are initialized based on the first two bytes of the mask, i.e. $R[0]||R[1]$. A fixed permutation G is used in the Even-Mansour fashion to construct them, and the algorithm buildIndicesPermutation is described in Figure 7.
- The *multiplicative mask* byte $a \in \mathsf{GF}(2^8) \setminus \{0\}$ is computed depending on the value of $R[0]||R[1]$. This is computed with buildMultiplicativeMask.
- The S-box masking bytes b_{in} and b_{out} are chosen as $R[16]$ and $R[17]$ respectively.
- The S-box is realized through masked and precomputed S-box table of 256 bytes, hence we denote it by L. We use $L[i]$ to denote i-th element in this table, for $i \in [256]$.
- The input array P of size 16 is used for storing the internal cipher state. For memory efficiency, the same array is used for producing the ciphertext at the end of the encryption.
- The initial key array K of size 16 is used to store the intermediate round key.
- The array M of size 16 is used to store the intermediate mask state, which is dynamically updated. This is in contrast to R, which is fixed throughout encryption.

Pre-processing. During this phase, first, the permutations π_0, π_1, π_2 are established with buildIndicesPermutation depending on the two-byte mask value $R[0]||R[1]$. Then, the multiplicative mask a is computed with buildMultiplicativeMask. This is followed by computeAndStoreMaskedSbox, which calculates the entries of the mask-dependent S-box table, based on $b_{\mathsf{in}} = R[16]$, $b_{\mathsf{out}} = R[17]$ and a. Computation of each S-box table entry is also randomized. Then, loadAndMaskInput transforms the given input such that each state byte $S[i]$ is transformed into $a \times S[i] \oplus M[i]$, hence *affine masking* is used. In the same fashion, loadAndMaskKey transforms each key byte $K[i]$ into $a \times K[i]$. Note that the order of byte accesses in both loadAndMaskInput and loadAndMaskKey are randomized by the permutation π_0 which is computed with first two mask bytes $R[0]$ and $R[1]$.

These algorithms are given in Figure 7. This precomputation phase takes up about the half of the whole encryption duration.

AES Layers. Given that each state byte is stored in the form of $a \times S[i] \oplus M[i]$ and the key byte is stored as $a \times K[i]$, the individual layers of AES must be reconsidered to play along with the format. Here, AddRoundKey is straightforwardly executed by simply XORing key bytes into the state. With SubBytes, each state byte is first unmasked from $M[i]$ share, passed through S-box table L, and then remasked again with $M[i]$.

An important implication of this particular S-box implementation is that both the cipher state $S[i]$ and the internal mask state $M[i]$ must be kept in sync, otherwise one can not correctly compute the S-box layer. Therefore, ShiftRows is performed on both the cipher state, as well as the mask state in parallel. The double application idea is similar for the linear operation defined by MixColumns. A temporary orderFromOneToTwo is done before and afterwards MixColumns, which XORs a column of masks among them to increase the order of mask application during MixColumns. It must also be noted that the byte orders for each layers are randomized, so that the leak of information is minimized.

buildIndicesPermutation:

1: $G \leftarrow [12, 5, 6, 11, 9, 0, 10, 13, 3, 14, 15, 8, 4, 7, 1, 2]$
2: $\mathsf{SR} \leftarrow [0, 13, 10, 7, 4, 1, 14, 11, 8, 5, 2, 15, 12, 9, 6, 3]$
3: $m_1||m_0||m_3||m_2 \leftarrow R[0]||R[1]$ where each m_i is a nibble
4: $\pi_0[i] \leftarrow G[G[G[G[i \oplus m_0] \oplus m_1] \oplus m_2] \oplus m_3]$ for $i \in [16]$
5: $\pi_1[i] \leftarrow G[G[G[\pi_0[i] \oplus m_1] \oplus m_2] \oplus m_3]$ for $i \in [16]$
6: $\pi_2[i] \leftarrow \mathsf{SR}[\pi_0[i]]$ for $i \in [16]$

buildMultiplicativeMask:

1: $a_0 \leftarrow \pi_0[2]||\pi_1[2]$ and $a_1 \leftarrow \pi_0[3]||\pi_1[3]$
2: If $a_0 \neq 0$, then $a \leftarrow a_0$, otherwise $a \leftarrow a_1$

computeAndStoreMaskedSbox:

1: **for** $i = 0$ to 255 **do**
2: $\quad j \leftarrow (i + R[15]) \bmod 256$
3: $\quad L[j] \leftarrow a \times \mathsf{S\text{-}box}[(j \oplus b_{\mathsf{in}}) \times a^{-1}] \oplus b_{\mathsf{out}}$

loadAndMaskInput:

1: $M[i] \leftarrow R[i + 2]$ for $i \in [16]$
2: **for** $i = 0$ to 15 **do**
3: $\quad j \leftarrow \pi_0[j]$
4: $\quad S[j] \leftarrow (a \times P[j]) \oplus M[j]$

loadAndMaskKey:

1: **for** $i = 0$ to 15 **do**
2: $\quad j \leftarrow \pi_0[j]$
3: $\quad K[j] \leftarrow (a \times K[j])$

Figure 7: The initialization and mask processing takes roughly 167000 clock cycles, and consumes about the half of the operations during AES encryption.

4.2 Exposure of secAES

In this section, we draw the reader's attention to a few particular observations regarding secAES implementation. These points make secAES scheme a good showcase to implement Hamming Weight based attack.

1. In the classical threshold/masked implementations, both plaintext and ciphertext values are represented with n shares. The device, which is assumed to be vulnerable to an attacker with up to $n-1$ probes, receives n shares together as input, and returns n shares back after encryption. Therefore, neither plaintext nor ciphertext appears in clear form on the device. Furthermore, there is a uniform distribution property that ensures that $n-1$ shares do not leak any information. secAES does not fit into this description, because there are large number of internal computations dependent on the mask value, these are namely S-box table computations, dynamic masking of the key and the plaintext states as well as calculation of the permutations used in the random ordering of operations. For this reason, the device actually receives plaintext and the mask in their original form, performs the rather heavyweight masking operations on the vulnerable device for each encryption. This makes mask dependent computations to be exposed to the attacker.

2. secAES uses considerably fewer number of mask bytes to protect itself against side-channel attacks. This is made possible by the fact that each state and key byte is multiplied by the same value, that is the multiplicative mask a. This means that, for each AES invocation, each plaintext and key byte must be multiplied with a in

AESEncryption:
1: buildIndicesPermutation
2: buildMultiplicativeMask
3: computeAndStoreMaskedSbox
4: loadAndMaskInput
5: loadAndMaskKey
6: addRoundKey
7: **for** $i = 0$ to 9 **do**
8: maskedSubBytes
9: shiftRows(S, π_0)
10: shiftRows(M, π_1)
11: **if** $i \neq 9$ **then**
12: orderFromOneToTwo
13: mixColumns(S, π_0)
14: mixColumns(M, π_1)
15: orderFromOneToTwo
16: keyExpansion
17: addRoundKey
18: **for** $i = 0$ to 15 **do**
19: $S[i] \leftarrow a^{-1} \times (S[i] \oplus M[i])$

keyExpansion:
1: $u \leftarrow a \times \mathsf{RC}[\mathsf{round}]$
2: $K[0] \leftarrow K[13] \oplus u \oplus L[K[0] \oplus b_{\mathsf{in}}] \oplus b_{\mathsf{out}}$
3: $K[1] \leftarrow K[14] \oplus L[K[1] \oplus b_{\mathsf{in}}] \oplus b_{\mathsf{out}}$
4: $K[2] \leftarrow K[15] \oplus L[K[2] \oplus b_{\mathsf{in}}] \oplus b_{\mathsf{out}}$
5: $K[3] \leftarrow K[12] \oplus L[K[3] \oplus b_{\mathsf{in}}] \oplus b_{\mathsf{out}}$
6: **for** $i = 4$ to 15 **do**
7: $K[i] \leftarrow K[i] \oplus K[i-4]$

addRoundKey:
1: **for** $i = 0$ to 15 **do**
2: $j \leftarrow \pi_0[i]$
3: $S[j] \leftarrow S[j] \oplus K[j]$

maskedSubBytes:
1: **for** $i = 0$ to 15 **do**
2: $j \leftarrow \pi_0[i]$
3: $z \leftarrow S[j] \oplus M[j]$
4: $z \leftarrow L[z \oplus b_{\mathsf{in}}] \oplus b_{\mathsf{out}}$
5: $S[j] \leftarrow z \oplus M[j]$

shiftRows(X, Π):
1: **for** $i = 0$ to 15 **do**
2: $j \leftarrow \Pi[i]$
3: $X'[j] \leftarrow X[j]$
4: **for** $i = 0$ to 15 **do**
5: $j \leftarrow \Pi[i]$
6: $v \leftarrow \pi_2[i]$
7: $X[v] \leftarrow X'[j]$

orderFromOneToTwo:
1: **for** $i = 0$ to 15 **do**
2: $j \leftarrow \pi_0[i]$
3: $S[j] \leftarrow S[j] \oplus M[(j+4) \bmod 16]$

mixColumns(X, π):
1: $r \leftarrow \pi[0] \cdot \mathsf{0x03}$
2: **for** $i = 0$ to 3 **do**
3: $j \leftarrow 4(i \oplus r)$
4: $X[j : j+4] \leftarrow (\mathcal{M} \times X[j : j+4]^{\tau})^{\tau}$

Figure 8: secAES pseudocode.

$\mathsf{GF}(2^8)$. This means that there are large number of operations that depend on the same value of a. In other words, there are many points in the collected power traces that correlate to a.

For this reason, after collecting power traces, we focus our attention to the time intervals during which the mask-based pre-computation operations take place, instead of the actual AES encryption rounds. In particular, we look at loadAndMaskInput and loadAndMaskKey, as each invocations of these algorithm trigger 16 $\mathsf{GF}(2^8)$ multiplications.

From an attacker's perspective whose goal is to recover the key, there are two main challenges to overcome: the multiplicative mask a that masks each of the key bytes, and the randomized order of bytes for the multiplication operations, i.e., π_0. Both the multiplicative mask a and the randomizing permutation π_0 are determined by $R[0]||R[1]$, hence we make it our initial goal to recover this two-byte mask value. Our idea is that, if we can recover $R[0]||R[1]$ with good probability, then we can compute both a and the permutation π_0, and correctly correlate Hamming Weight observations to key byte candidates.

As clear in Figure 9, given a single power trace, the regions whose Hamming Weight correlate to the collected power measurements can be easily distinguished. We will simply refer to the sets of power measurements that corresponds to GF256_mul routine as *slots*. Each of these algorithms contains exactly 16 slots, whose position in time is fixed, but the values processed within these slots are randomized with the permutation π_0.

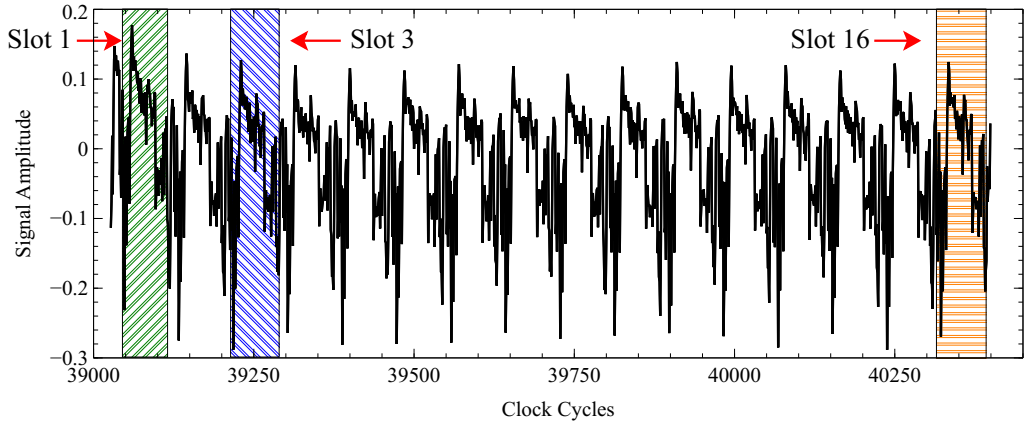


Figure 9: The single power trace in the region of loadAndMaskInput. 16 slots corresponding to the field multiplication algorithm GF256_mul is visible, and in particular slots 1, 3 and 16 are marked.

4.3 GF(2^8) Multiplication

For efficiency, multiplication of two bytes x and y is done with the use of log and $\log^{-1}$ tables. Namely, $x \times y = \log^{-1}((\log x + \log y) \bmod 255)$, where log is defined with respect to a generator g such that $\langle g \rangle = \mathsf{GF}(2^8) \setminus \{0\}$. We denote these tables with arrays LOG and ALOG, each of which consists of 256 byte values[4]. The simplified equivalent of this algorithm that we reconstructed from the original AVR assembly source code looks like following:

GF256_mul(r16, r17):

1: Push r0, r1, r18, r30, r31 to stack
2: r5 ← 0 if r16 · r17 = 0, otherwise r5 ← 1
3: r16 ← LOG[r16]
4: r18 ← LOG[r17]
5: r16 ← (r16 + r18) mod 255
6: r16 ← ALOG[r16]
7: r1||r0 ← r16 · r5
8: r16 ← r0
9: Pop r31, r30, r18, r1, r0 from the stack
10: **return**

In this algorithm, r0–r31 represent the byte registers inside the microprocessor and " · " represents simple integer multiplication. This algorithm evidently computes r16 × r17 for the two values stored in these registers and places the result back in r16. In order to handle the exceptional case where either of the two input bytes is 0 in a constant-time fashion, a multiplication-based check is used. In other words, r5 is set to 0, if r16 · r17 = 0, and otherwise r5 is set to 1. Later, r5 is used again in the multiplication to decide whether the final value must be overwritten with zero. In short, this algorithm takes the same number of clock cycles to execute regardless of its input values.

An interesting property regarding the implementation of secAES is that the caller of GF256_mul incidentally sets r0 = 0 in its scope, in order to use this register for carry addition. Therefore, the pushed and popped values of r0 is fixed to 0, and this relationship holds as long as the function is invoked from either of loadAndMaskInput and loadAndMaskKey. Between lines 7–10, r0 stores the results of the GF(2^8) multiplication

[4]One can notice that LOG[0] and ALOG[255] are irrelevant entries that are never accessed.

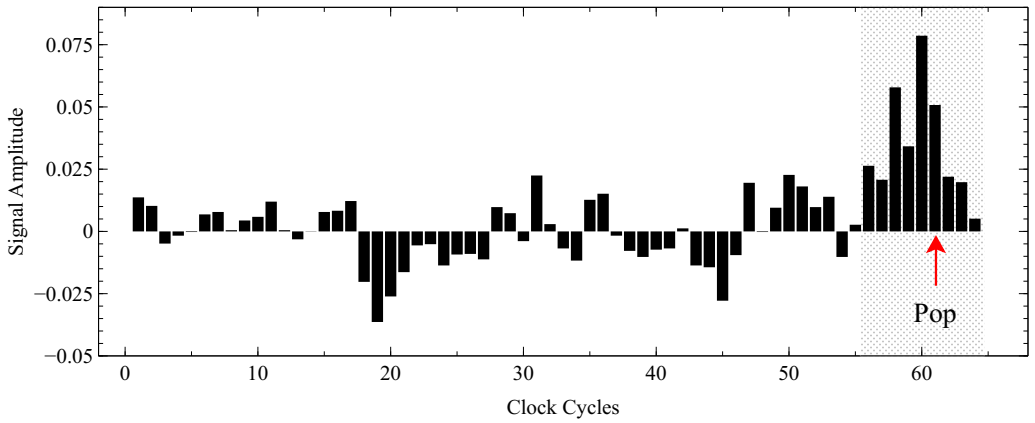


Figure 10: The difference of energy consumed during two runs of GF256_mul. The plaintext is chosen carefully such that the Hamming Weight of the multiplication results is respectively 0 and 8, for these two traces.

briefly, until the popped 0 value from the stack is overwritten to r0. Hence, here we find a transition that directly depends on the Hamming Weight of the field multiplication result.

The correlation of Hamming Weight to power measurement can be empirically verified in Figure 10. In this figure, each bar corresponds to the difference of energy consumed during a full clock cycle. Without loss of generality, we look at the third slot of two power traces during the loadAndMaskInput routine. We choose the mask and plaintext carefully so that the Hamming Weight of the field multiplication result $a \times P[i]$ is respectively 0 and 8 for this slot. Then, we can roughly find out the particular clock cycles/instruction that show dependence on the Hamming Weight of this result. We repeat this experiment many times to ensure that a particularly chosen time point is not due to a noise, but shows up reliable across multiple measurements. Our experiments concluded with the fact that the 60th clock cycle is one such points of relevance. In fact, this happens to be the very clock cycle during which r0 switches from the multiplication result to 0.

Before proceeding to details of an actual key-recovery attack, let us illustrate how well sampled power points, whose exact point in time chosen carefully, correlate with the Hamming Weight of the field multiplication result. In order to see this correlation, we collect power traces for 4096 encryption (again for the third slot, for the sake of our example) of loadAndMaskInput. For each encryption, we know the used plaintext and the mask bytes $R[0]||R[1]$ in advance which allows us to determine both a and π_0 i.e., the permutation that determines in which order each of the bytes are accessed. Hence we compute the Hamming Weight of $a \times P[i]$ for some byte index i for which multiplication of $P[i]$ is done in this third slot according to the order established by π_0. We construct nine different sets of power measurements $\mathcal{P}_h = \{y_r : W(a \times P[i]) = h\}$ for $h \in [9]$, and use the obtained power measurements y_r to determine a mean and variance for each set $\mathcal{P}_h$ assuming that they are samples from a normal random variable (note that y_r denotes the amplitude of the power measurement during the r-th run).

The outcome of this experiment is given in Figure 11. For instance, it is clear that, by looking at a power value obtained from a single trace, we can deduce whether an the result of $a \times P[i]$ has a Hamming Weight of 0 or 8 with a good confidence. However, among neighboring Hamming Weights, this information is harder to deduce. Nonetheless, once equipped with the mean and variance of these distributions, it is feasible to do a thorough analysis over multiple power traces. However to construct a valid attack, we need to ensure that the attacker will not know the value of any of the mask bytes used for any encryption apriori. Hence we need to device a method that allows the attacker to deduce at least the

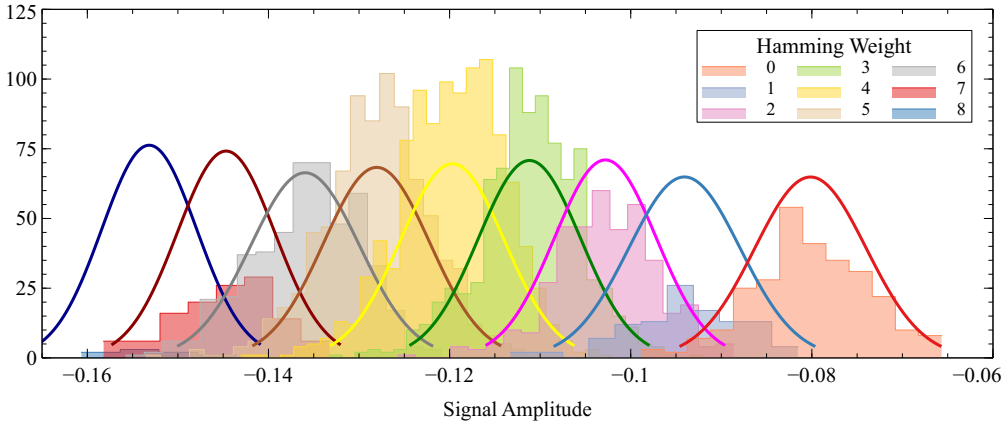


Figure 11: Power values that corresponds to the third invocation of GF256_mul from loadAndMaskInput. Each color corresponds to different Hamming Weight for the result of $a \times P[i]$. In the background, the histogram of actual power values are given. The drawn lines in front show the ideal normal distribution constructed from collected values.

first 2 mask bytes from the power traces alone. Hence, we will first explore how we can recover the first two bytes of the mask, $R[0]||R[1]$.

4.4 Mask Recovery

Let $\mathcal{N}_{j,h}$ denote the idealized normal distribution of power measurements for Hamming Weight h at slot j. Typically, the mean of the first two slots are slightly higher than the rest of the fourteen slots, and for this reason, we treat each slot separately. We will denote the probability density functions with $\mathcal{N}_{j,h}.\mathsf{PDF}(x)$ for a measurement x. For instance, in Figure 11, the distributions $\mathcal{N}_{3,h}$ of the slot 3 are given.

We set the 16 bytes of the plaintext as $P[\cdot] = [1, 2, \ldots, 16]$, so that each plaintext byte is unique and allows us to extract more information from a single trace. Then, for the multiplicative mask value a, we should observe power measurements where the results of the field multiplication are $a, 2 \times a, 3 \times a, \ldots, 16 \times a$. However, these measurements will appear in the slots with a randomized order, because of π_0.

Pre-computation. Let $r = R[0]||R[1]$. Let P denote the fixed plaintext as above such that $P[i] = i + 1$. Let a and π_0 be functions of r, as their value depend only on r (see buildIndicesPermutation and buildMultiplicativeMask in Figure 7). We compute a look-up table I with 2^{16} entries, where each entry is a list of ideal Hamming Weights in 16 slots in corrected order. In other words, $I[r] = \mathsf{Permute}_{\pi_0(r)}([W(a(r)), W(2 \times a(r)), \ldots, W(16 \times a(r))])$, where $\mathsf{Permute}_{\pi_0(r)}$ permutes the elements with respect to $\pi_0(r)$. For example, if $\pi_0(0) = 1$, $\pi_0(1) = 0$ and $\pi_0(r) = r$ for all other $r \in [16]$ then $I[r] = [W(2 \times a(r)), W(a(r)), W(3 \times a(r)), W(4 \times a(r)), \ldots, W(16 \times a(r))]$.

Mask Likelihood. Consider the following hypothetical probability experiment. Let $\mathcal{N}_0, \ldots, \mathcal{N}_8$ denote normal distributions. Let u be a uniformly sampled byte from $\{0, 1\}^8$. Let $h = W(u)$. Let X denote the random variable following the distribution $\mathcal{N}_h$. The main question in this experiment is, given a measurement x, what is the probability of the actual Hamming Weight $H = h'$? From Bayes' theorem:

$$\Pr[H = h' | X = x] = \frac{\mathcal{N}_{h'}.\mathsf{PDF}(x) \cdot \Pr[H = h']}{\sum_{\bar{h}=0}^{8} \mathcal{N}_{\bar{h}}.\mathsf{PDF}(x) \cdot \Pr[H = \bar{h}]}$$

Table 2: The probability of recovering the correct mask among the highest ranking k candidates. Ideally, we would like the probability for $k = 1$ to be as high as possible.

Among top k	1	2	3	4	8	16
The probability p	0.721	0.815	0.854	0.877	0.930	0.957

Therefore, given a candidate two-byte mask value r', and 16 power measurements $x_1, \ldots, x_{16}$ corresponding to each of the 16 slots, we can compute the probability of $r = r'$, where r denotes the unknown correct two-byte mask. For that, we take all 16 slots into account. Namely:

$$\Pr[r = r' | (X_1, \ldots, X_{16}) = (x_1, \ldots, x_{16})] = \prod_{j=1}^{16} \frac{\mathcal{N}_{j,I[r][j]}.\mathsf{PDF}(x) \cdot \Pr[h = I[r][j]]}{\sum_{\bar{h}=0}^{8} \mathcal{N}_{j,\bar{h}}.\mathsf{PDF}(x) \cdot \Pr[h = \bar{h}]}$$

In short, we can compute this probability for each mask candidate, and rank all the masks depending on their probability. The first candidate should give us the mask with the maximum likelihood.

The result of this maximum likelihood approach is given in Table 2. For more than half of the single power traces, we are able to recover the correct two-byte mask value $R[0]||R[1]$ as the highest ranking candidate.

4.5 Key Recovery

From the masking recovery algorithm, we are able to recover first 2 mask bytes for a large number of successive encryptions, where more than half of them are correct. Then, for each trace, we can compute both the permutation π_0 and the multiplicative mask a. For recovering the key, we will follow the same idea, but this time focus on the loadAndMaskKey algorithm, in which $a \times K[i]$ field multiplications are executed.

Pre-computation. Suppose that we have ℓ power traces for ℓ successive encryptions, hence ℓ recovered mask values $r_1, \ldots, r_\ell$. Some of these masks are naturally incorrect, and we cannot know which ones are incorrect. For each trace, we compute $a(r_i)$ and $\pi_0(r_i)$ and store them in a table. Unlike in the mask case, where the ideal weights are pre-computed, we cannot do the same, as the key space is considerably larger than the Hamming Weight space. Instead, we look at collected power values $x_{i,j}$ that belongs to i-th trace and j-th slot in loadAndMaskKey. Then, for each slot and power trace, we compute the probability of Hamming Weight h' in advance, for each $h' \in [9]$. In other words, we construct a table J such that:

$$J[i, j, h'] = \Pr[H = h' | X_j = x_{i,j}] = \frac{\mathcal{N}_{j,h'}.\mathsf{PDF}(x_{i,j}) \cdot \Pr[H = h']}{\sum_{\bar{h}=0}^{8} \mathcal{N}_{j,\bar{h}}.\mathsf{PDF}(x_{i,j}) \cdot \Pr[H = \bar{h}]}.$$

Key Likelihood. For simplicity, let us explain our approach for a single byte position of the key. In particular, suppose that we want to recover the first byte of the key, that is $K[0]$. For $K[0]$, each power trace contains exactly one slot that corresponds to the value $a(r_i) \times K[0]$. As long as our guess on r_i is correct, we will compute $a(r_i)$ and $\pi(r_i)$ correctly, the observation we make will be useful, because we will correctly guess the slot j where the multiplication of 0-th key byte is done. Otherwise, it will contribute to our key ranking as noise.

For each byte candidate $k' \in \{0, 1\}^8$, we initially construct an empty list, $\mathcal{P}[k'] = \emptyset$. For each power trace, we will add exactly one probability term to each candidate byte's list. For

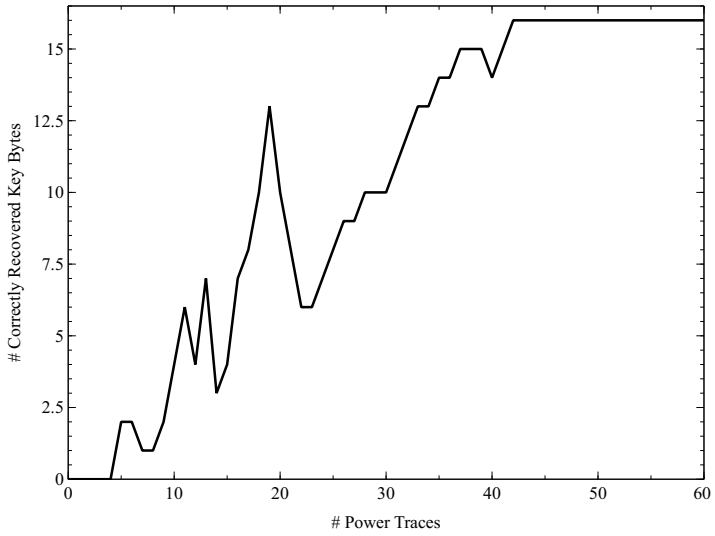


Figure 12: The number of correctly recovered key bytes (among full 16 bytes) vs. the number of power traces.

that, we first compute the Hamming Weight of $a(r_i) \times k'$, that is $u \leftarrow W(a(r_i) \times k')$. Then, we calculate the correct slot $v \leftarrow \pi_0(r_i)[0]$ in which we can find the power measurement for the key byte position 0. Then, the probability term $\{J[i, v, u]\}$ is inserted to the list $\mathcal{P}[k']$. Eventually, for ℓ power traces, each list $\mathcal{P}[k']$ contains ℓ terms, and we can compute the key byte likelihood of k' by multiplying all terms in $\mathcal{P}[k']$. If the mask is computed correctly, the list $P[k']$ is populated with a high-probability value for the correct candidate k' and with a low-probability value for the other candidates. If the mask is computed incorrectly, we expect that the terms added to the list are random. The remaining part, again, is to order key byte candidates k' in descending order, and pick the highest ranking candidate as the guess. The same algorithm is repeated for the remaining 15 byte positions of the key.

The result of this maximum likelihood is given in Figure 12. In conclusion, approximately 50 power traces are sufficient to recover all bytes of the key.

5 Final Words and Future Work

In this paper, we revisited the intrinsic connection between the Hamming Weight of intermediate cipher variables and the power consumption of an algorithm from which we extrapolate two main contributions.

1. A practical side-channel-assisted reverse engineering procedure on AES-like constructions found in commercial products such as Dencrypt Message [den21] where the round function components are secret, i.e., undisclosed encryption key as well as the SubBytes, ShiftRows and MixColumns functions.

2. A practical side-channel-assisted key-recovery attack on the 8-bit secAES proposal by the French National Cybersecurity Agency (ANSSI) that combines several side-channel countermeasures.

Therefore, a promising future line of research in this direction would be to identify for each platform, instruction sequences that when invoked in a certain order leak information

in some form or the other that make the underlying implementation vulnerable. Although, at first glance, this seems to be non-trivial task to perform, it may be possible to do so by correctly modeling the power footprint for each commonly used instruction of any given platform.

References

[BBH+19] Shivam Bhasin, Jakub Breier, Xiaolu Hou, Dirmanto Jap, Romain Poussier, and Siang Meng Sim. SITM: See-in-the-middle. *IACR TCHES*, 2020(1):95–122, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8394.

[BLPR] Ryad Benadjila, Victor Lomné, Emmanuel Prouff, and Thomas Roche. Secure AES128 for ATMega8515. Available at https://github.com/ANSSI-FR/secAES-ATmega8515.

[CIW13] Christophe Clavier, Quentin Isorez, and Antoine Wurcker. Complete SCARE of AES-like block ciphers by chosen plaintext collision power analysis. In Goutam Paul and Serge Vaudenay, editors, *INDOCRYPT 2013*, volume 8250 of *LNCS*, pages 116–135. Springer, Heidelberg, December 2013.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 398–412. Springer, Heidelberg, August 1999.

[Cla07] Christophe Clavier. An improved SCARE cryptanalysis against a secret A3/A8 GSM algorithm. In Patrick D. McDaniel and Shyam K. Gupta, editors, *Information Systems Security, Third International Conference, ICISS 2007, Delhi, India, December 16-20, 2007, Proceedings*, volume 4812 of *Lecture Notes in Computer Science*, pages 143–155. Springer, 2007.

[DEKM17] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, and Florian Mendel. Side-channel analysis of keymill. In Sylvain Guilley, editor, *COSADE 2017*, volume 10348 of *LNCS*, pages 138–152. Springer, Heidelberg, April 2017.

[den21] Dencrypt Message. https://www.dencrypt.dk/#message, 2021. [Accessed 15-April-2021].

[FMPR11] Guillaume Fumaroli, Ange Martinelli, Emmanuel Prouff, and Matthieu Rivain. Affine masking against higher-order side channel analysis. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *SAC 2010*, volume 6544 of *LNCS*, pages 262–280. Springer, Heidelberg, August 2011.

[JB20] Dirmanto Jap and Shivam Bhasin. Practical reverse engineering of secret sboxes by side-channel analysis. In *IEEE International Symposium on Circuits and Systems, ISCAS 2020, Sevilla, Spain, October 10-21, 2020*, pages 1–5. IEEE, 2020.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, August 1999.

[Knu14] Lars R. Knudsen. Dynamic encryption. *J. Cyber Secur. Mobil.*, 3(4):357–370, 2014.

[Knu20] Lars R. Knudsen. Dynamic encryption method. United States Patent, US20200021566A1, https://patents.google.com/patent/US20200021566A1, January 2020.

[Nov03] Roman Novak. Side-channel attack on substitution blocks. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *ACNS 03*, volume 2846 of *LNCS*, pages 307–318. Springer, Heidelberg, October 2003.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 06*, volume 4307 of *LNCS*, pages 529–545. Springer, Heidelberg, December 2006.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *COSADE 2014*, volume 8622 of *LNCS*, pages 243–260. Springer, Heidelberg, April 2014.

[RPD09] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-order masking and shuffling for software implementations of block ciphers. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 171–188. Springer, Heidelberg, September 2009.

[RR13] Matthieu Rivain and Thomas Roche. SCARE of secret ciphers with SPN structures. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part I*, volume 8269 of *LNCS*, pages 526–544. Springer, Heidelberg, December 2013.

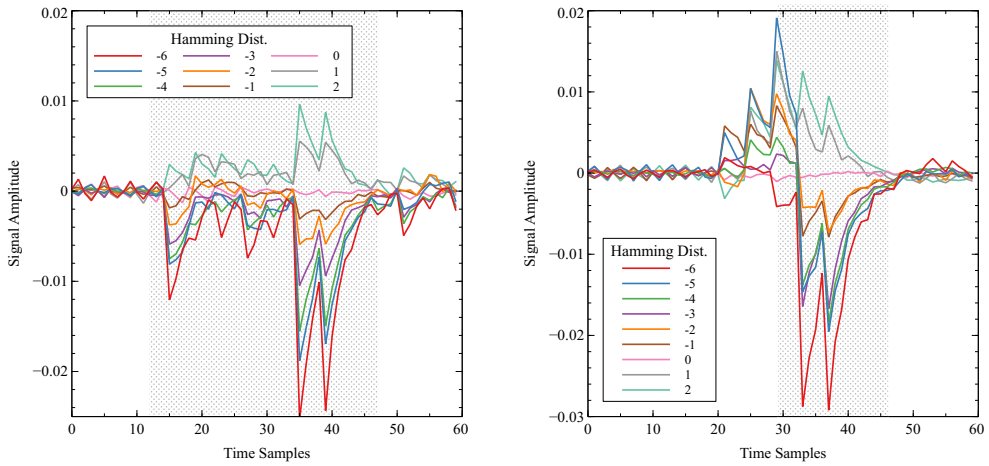
[SJB21] Siang Meng Sim, Dirmanto Jap, and Shivam Bhasin. DAPA: Differential analysis aided power attack on (non-)linear feedback shift registers. *IACR TCHES*, 2021(1):169–191, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8731.

[TKKL15] Tyge Tiessen, Lars R. Knudsen, Stefan Kölbl, and Martin M. Lauridsen. Security of the AES with a secret S-box. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 175–189. Springer, Heidelberg, March 2015.

[TRS16] Mostafa M. I. Taha, Arash Reyhani-Masoleh, and Patrick Schaumont. Keymill: Side-channel resilient key generator, A new concept for SCA-security by design - A new concept for SCA-security by design. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 217–230. Springer, Heidelberg, August 2016.

[von01] Manfred von Willich. A technique with an information-theoretic basis for protecting secret data from differential power attacks. In Bahram Honary, editor, *8th IMA International Conference on Cryptography and Coding*, volume 2260 of *LNCS*, pages 44–62. Springer, Heidelberg, December 2001.

A STM32F3 Plots



(a) Key Whitening (b) Substitution Box

Figure 13: Averaged differential power traces (STM32F3) of one state byte during the initial key whitening phase (a) and at the end of the substitution layer (b).