Complete Practical Side-Channel-Assisted Reverse Engineering of AES-Like Ciphers

Andrea Caforio[1], Fatih Balli[1,2], and Subhadeep Banik[1]

[1] LASEC, Ecole Polytechnique Fédérale de Lausanne, Switzerland
{andrea.caforio,subhadeep.banik}@epfl.ch
[2] CSEM, Switzerland
fatih.balli@csem.ch

Abstract. Public knowledge about the structure of a cryptographic system is a standard assumption in the literature and algorithms are expected to guarantee security in a setting where only the encryption key is kept secret. Nevertheless, undisclosed proprietary cryptographic algorithms still find widespread use in applications both in the civil and military domains. Even though side-channel-based reverse engineering attacks that recover the hidden components of custom cryptosystems have been demonstrated for a wide range of constructions, the complete and practical reverse engineering of AES-128-like ciphers remains unattempted.
In this work, we close this gap and propose the first practical reverse engineering of AES-128-like custom ciphers, i.e., algorithms that deploy undisclosed SubBytes, ShiftRows and MixColumns functions. By performing a side-channel-assisted differential power analysis, we show that the amount of traces required to fully recover the undisclosed components are relatively small, hence the possibility of a side-channel attack remains as a practical threat. The results apply to both 8-bit and 32-bit architectures and were validated on two common microcontroller platforms.

1 Introduction

Over the past few years, the field of side-channel-assisted cryptanalysis has evolved into an intricate spectrum. In this spectrum, the trace, which is the signal collected by the adversary during the execution of a cryptographic operation, can stem from various sources, such as the electromagnetic emission, the power consumption, or even the sound noise generated by the victim device [4,13,19]. Furthermore, there are many available techniques to analyze the collected traces with the goal of recovering the secret key [7,13,15].

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. Undisclosed proprietary cryptographic algorithms are still used in civil applications, e.g., GSM or Pay-TV systems, and in diplomatic or military domains. Even though *security through obscurity* is far from ideal and generally discouraged by cryptographers, from the implementation layer perspective, it is considered as an extra layer of protection against all types of attacks, including that of side-channels. In particular, one idea that we consider in this paper is to implement a custom version of a popular scheme, e.g., AES, by replacing the inner layer of operations without publicly disclosing these modifications. Obviously, the idea is that extrapolating conclusions from side-channel observations becomes significantly harder when the construction in question is not fully disclosed. Therefore, the adversary would need to collect larger amount of traces. This is exactly the approach taken by the Danish enterprise *Dencrypt* whose communication devices ship with a customized AES implementation with secret S-boxes based on the *Dynamic Encryption* proposal [12].

The first known use of side channels to reverse-engineer (SCARE) hidden structures was the case of the A3/8 algorithm used in GSM [17]. This attack reveals the contents of one of the two substitution tables, which are intended to be kept secret, used for authentication and key agreement in GSM. This was later improved by Clavier in an attack that fully recovers both tables [8]. In a related work, Clavier et al. [9] presented a theoretical 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 [21] proposed a generic reverse engineering attack which applies to a general class of undisclosed substitution-permutation ciphers, showing that this line of attack works beyond the AES constructions.

Note that none of the previous works demonstrate the mentioned attacks in practice, but instead their results are only based on theoretical simulations. More specifically, they all rely on the assumption that some side-channel observations can be made that allows the attacker to distinguish whether intermediate values of an algorithm are equal at different points during the computation. However, these works neither back up the assumption through an experimental setup, nor present a practical full-recovery attack. It is thus important to determine the efficacy of side-channel reverse engineering on real-world platforms. The first practical attack was presented by Jap and Bhasin [11]. 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 and succeeded in recovering 159 out of 256 entries. However, a practical side-channel-assisted reverse engineering attack that recovers the full description of an AES-128-like cipher remains an open problem.

Contributions. In this paper, we demonstrate the first practical side-channel-assisted reverse engineering procedure for the full description of unprotected AES-128-like ciphers that deploy undisclosed SubBytes, ShiftRows and MixColumns functions. A precise definition of such a cipher will be given shortly. Our attacks follow the side-channel-assisted differential plaintext methodology (SCADPA) pioneered by Breier et al. [6] and subsequently extended by Bhasin et al. [5], whose work enhances differential power analysis [13] with tools from conventional differential cryptanalysis. Specifically, the complete recovery routine proceeds in four consecutive steps as detailed in Table 1. This work thus closes the open question of whether such an attack is feasible, and more so on the cost of performing such attack. We validate our recovery routines on both 8-bit and 32-bit systems, namely the 8-bit ATXMEGA128D4 and 32-bit STM32F303 architectures that find wide use in the industry.

Table 1. Complexity (the number of traces) of our proposed AES-128-like side-channel-assisted reverse engineering algorithms. The parameter α denotes the required number of repetitions in order to get a stable average in the extracted power traces. On our testing equipment, $\alpha \approx 10$ was sufficient for effectively de-noising the traces.

Recovery	Platforms	Complexity	Reference
Encryption Key	8-bit, 32-bit	$\alpha \times 2^9$	Section 2.3
Partial ShiftRows	8-bit, 32-bit	$\alpha \times 32$	Section 3.1
255 MixColumns Candidates	8-bit, 32-bit	$\alpha \times 2^{20}$	Section 3.2
Full SubBytes, ShiftRows, MixColumns	8-bit, 32-bit	$\alpha \times 2^{18}$	Section 3.3

Outline. We review some preliminary material concerning side-channel-assisted reverse engineering attacks in Section 2. Section 3 details our procedures that recover the complete description of hidden components within AES-128-like ciphers. Ultimately, the paper is concluded in Section 4.

2 Preliminaries

We commence the preliminaries with a precise definition of an AES-128-like cipher and then proceed with a review of the power consumption model in microcontrollers.

Definition 1 (AES-like cipher). *Denote by* AES^* *an* AES-128*-like SPN cipher over the Rijndael finite field of the form*

$$\begin{aligned} \mathsf{AES}^* : \mathbb{F}_{256}^{4\times4} \times \mathbb{F}_{256}^{4\times4} &\mapsto \mathbb{F}_{256}^{4\times4} \\ (p, k) &\mapsto y, \end{aligned}$$

for some plaintext p *and key* k*. The round function consists of a round key addition layer* AK*, a byte-wise substitution layer* SB *defined by a lookup table* $T : \mathbb{F}_{256} \mapsto \mathbb{F}_{256}$*, a byte permutation layer* PB *over* $\mathbb{F}_{256}^{4\times4}$ *that shuffles the state bytes according to some permutation* $\Pi \in S_{16}$ *(where* S_n *is the permutation group over* n *elements) and a linear diffusion layer* MC *that multiplies the state* *by a circulant matrix $M \in \mathbb{F}_{256}^{4\times 4}$ such that*

$$M = \begin{bmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{bmatrix}, \tag{1}$$

where $a, b, c, d \in \mathbb{F}_{256} \setminus \{0\}$. Without loss of generality, the round key generation is assumed to be achieved via the regular AES-128 *key scheduling function* KS *using T as the substitution table instead of the Rijndael S-box. Finally, the sequence of operations is the same as the original* AES-128 *algorithm, i.e.,*

```
AES*(p, k) :
1: AK(p, k)
2: for $i \leftarrow 1$; $i < 10$; $i \leftarrow i + 1$ do
3:     KS(k), SB(p), PB(p), MC(p), AK(p, k)
4: KS(k), SB(p), PB(p), AK(p, k)
```

In the following, we adopt the standard column-major notation to denote the individual bytes of the state as per Definition 2.

Definition 2 (Notation). *Let $b_{i,F(j)} \in \mathbb{F}_{256}$ be the value of the i-th state byte after the computational layer $F \in \{\mathsf{AK}, \mathsf{SB}, \mathsf{PB}, \mathsf{MC}\}$ in the j-th round function for $0 \le i \le 15$ and $0 \le j \le 10$. Analogously, let $c_{i,F(j)} \in \mathbb{F}_{256}^4$ be the value of the i-th state column for $0 \le i \le 3$. A graphical depiction of this notation is given in Figure 1.*

For the experiments we conducted in the paper, on both 8-bit and 32-bit microcontrollers, the AES* algorithm is implemented in a straightforward constant-time and byte-wise manner in which each state byte is computed individually in all layers of the round function. SB and PB are realized via standard lookup tables. The field multiplication steps that are part of MC are computed with a generic Galois field multiplication routine. This type of AES-128 implementation is common for 8-bit central processing units with limited memory. In 32-bit environments, a more compact T-table implementation is sometimes also deployed that combines the substitution and diffusion layers through lookup tables. We remark that for the remainder we are mostly interested in the computation of round key additions and the byte substitutions, hence our attacks are irrespective of the actual choice of implementation for the PB and MC operations. See Figure 2 for a generic set of byte-wise instructions that implement the AK and SB layers. Note that certain implementations also merge the AK and SB layers, however in many publicly available implementations, like OpenSSL, AVR-crypto-lib and the masked secAES proposal, these layers are separated [1,2,3].

2.1 Setup

The reverse engineering procedures proposed in this work have been validated on existing platforms. In particular, we utilized the following two microcontrollers:

$$\begin{array}{cccc} c_0 & c_1 & c_2 & c_3 \\ \left[\begin{array}{c} p_0 \\ p_1 \\ p_2 \\ p_3 \end{array}\right. & \begin{array}{c} p_4 \\ p_5 \\ p_6 \\ p_7 \end{array} & \begin{array}{c} p_8 \\ p_9 \\ p_{10} \\ p_{11} \end{array} & \left.\begin{array}{c} p_{12} \\ p_{13} \\ p_{14} \\ p_{15} \end{array}\right] \end{array} \xrightarrow[\text{Round 0}]{\mathsf{AK}} \begin{array}{cccc} c_{0,\mathsf{AK}(0)} & c_{1,\mathsf{AK}(0)} & c_{2,\mathsf{AK}(0)} & c_{3,\mathsf{AK}(0)} \\ b_{0,\mathsf{AK}(0)} & b_{4,\mathsf{AK}(0)} & b_{8,\mathsf{AK}(0)} & b_{12,\mathsf{AK}(0)} \\ b_{1,\mathsf{AK}(0)} & b_{5,\mathsf{AK}(0)} & b_{9,\mathsf{AK}(0)} & b_{13,\mathsf{AK}(0)} \\ b_{2,\mathsf{AK}(0)} & b_{6,\mathsf{AK}(0)} & b_{10,\mathsf{AK}(0)} & b_{14,\mathsf{AK}(0)} \\ b_{3,\mathsf{AK}(0)} & b_{7,\mathsf{AK}(0)} & b_{11,\mathsf{AK}(0)} & b_{15,\mathsf{AK}(0)} \end{array} \xrightarrow[\text{Round 1}]{\mathsf{SB}}$$

$$\begin{array}{cccc} c_{0,\mathsf{SB}(1)} & c_{1,\mathsf{SB}(1)} & c_{2,\mathsf{SB}(1)} & c_{3,\mathsf{SB}(1)} \\ b_{0,\mathsf{SB}(1)} & b_{4,\mathsf{SB}(1)} & b_{8,\mathsf{SB}(1)} & b_{12,\mathsf{SB}(1)} \\ b_{1,\mathsf{SB}(1)} & b_{5,\mathsf{SB}(1)} & b_{9,\mathsf{SB}(1)} & b_{13,\mathsf{SB}(1)} \\ b_{2,\mathsf{SB}(1)} & b_{6,\mathsf{SB}(1)} & b_{10,\mathsf{SB}(1)} & b_{14,\mathsf{SB}(1)} \\ b_{3,\mathsf{SB}(1)} & b_{7,\mathsf{SB}(1)} & b_{11,\mathsf{SB}(1)} & b_{15,\mathsf{SB}(1)} \end{array} \xrightarrow[\text{Round 1}]{\mathsf{PB}} \begin{array}{cccc} c_{0,\mathsf{PB}(1)} & c_{1,\mathsf{PB}(1)} & c_{2,\mathsf{PB}(1)} & c_{3,\mathsf{PB}(1)} \\ b_{0,\mathsf{PB}(1)} & b_{4,\mathsf{PB}(1)} & b_{8,\mathsf{PB}(1)} & b_{12,\mathsf{PB}(1)} \\ b_{1,\mathsf{PB}(1)} & b_{5,\mathsf{PB}(1)} & b_{9,\mathsf{PB}(1)} & b_{13,\mathsf{PB}(1)} \\ b_{2,\mathsf{PB}(1)} & b_{6,\mathsf{PB}(1)} & b_{10,\mathsf{PB}(1)} & b_{14,\mathsf{PB}(1)} \\ b_{3,\mathsf{PB}(1)} & b_{7,\mathsf{PB}(1)} & b_{11,\mathsf{PB}(1)} & b_{15,\mathsf{PB}(1)} \end{array} \xrightarrow[\text{Round 1}]{\mathsf{MC}}$$

$$\begin{array}{cccc} c_{0,\mathsf{MC}(1)} & c_{1,\mathsf{MC}(1)} & c_{2,\mathsf{MC}(1)} & c_{3,\mathsf{MC}(1)} \\ b_{0,\mathsf{MC}(1)} & b_{4,\mathsf{MC}(1)} & b_{8,\mathsf{MC}(1)} & b_{12,\mathsf{MC}(1)} \\ b_{1,\mathsf{MC}(1)} & b_{5,\mathsf{MC}(1)} & b_{9,\mathsf{MC}(1)} & b_{13,\mathsf{MC}(1)} \\ b_{2,\mathsf{MC}(1)} & b_{6,\mathsf{MC}(1)} & b_{10,\mathsf{MC}(1)} & b_{14,\mathsf{MC}(1)} \\ b_{3,\mathsf{MC}(1)} & b_{7,\mathsf{MC}(1)} & b_{11,\mathsf{MC}(1)} & b_{15,\mathsf{MC}(1)} \end{array} \xrightarrow[\text{Round 1}]{\mathsf{AK}} \begin{array}{cccc} c_{0,\mathsf{AK}(1)} & c_{1,\mathsf{AK}(1)} & c_{2,\mathsf{AK}(1)} & c_{3,\mathsf{AK}(1)} \\ b_{0,\mathsf{AK}(1)} & b_{4,\mathsf{AK}(1)} & b_{8,\mathsf{AK}(1)} & b_{12,\mathsf{AK}(1)} \\ b_{1,\mathsf{AK}(1)} & b_{5,\mathsf{AK}(1)} & b_{9,\mathsf{AK}(1)} & b_{13,\mathsf{AK}(1)} \\ b_{2,\mathsf{AK}(1)} & b_{6,\mathsf{AK}(1)} & b_{10,\mathsf{AK}(1)} & b_{14,\mathsf{AK}(1)} \\ b_{3,\mathsf{AK}(1)} & b_{7,\mathsf{AK}(1)} & b_{11,\mathsf{AK}(1)} & b_{15,\mathsf{AK}(1)} \end{array} \xrightarrow[\text{Round 2}]{\mathsf{SB}}$$

Fig. 1. Byte and column notations for the first two rounds. The notation scheme progresses similarly for later rounds.

- **ATXMEGA128D4.** An 8-bit microcontroller featuring a 2-stage-pipelined AVR processing unit. It offers 128 KB of flash memory and can be clocked at a maximum frequency of 32 MHz.
- **STM32F303.** A 32-bit microcontroller featuring a 3-stage-pipelined ARM Cortex-M4 processing unit. It offers 256 KB of flash memory and can be clocked at a maximum frequency of 72 MHz.

The two target microcontrollers are mounted on a ChipWhisperer CW308 board [18] that clocks them at a frequency of 7.37 MHz. Power traces are captured via the ChipWhisperer CW1173 board through a 10-bit 105 MS/s ADC. A key aspect of this setup, is that power traces are captured synchronously with the target clock, in other words, four samples per clock cycles are obtained at a frequency of roughly 30 MHz. Synchronous sampling, in contrast to asynchronous sampling performed by ordinary oscilloscopes, reduces the number of samples that are required for precise measurements and thus accelerates attacks that necessitate the processing of a large number of traces. This is reflected in the fact that taking an average over $\alpha \approx 10$ repetitions of an experiment was sufficient to effectively de-noise the power traces and attain a stable average.

```
; Round key addition
LD R1, [ADDR PT]
LD R2, [ADDR KEY]
XOR R1, R2
ST R1, [ADDR PT]
```

```
; Byte substitution
LD R1, [ADDR STATE]
ADD R2, R1, [ADDR SBOX]
LD R3, R2
ST R3, [ADDR STATE]
```

Fig. 2. Generic assembly of the AK (left) and SB (right) layers in AES* operating on a single byte. Note that statements within square brackets are akin to a function call, e.g., `[ADDR PT]` computes the plaintext address. It should be straightforward to convert the given snippets to valid assembly for any 8-bit or 32-bit architecture.

2.2 Power Leakage Model

Power leakage simulators for micro-controllers have been developed in the past for numerous systems. In the context of leakage models, SILK (simple leakage simulator) is one of the first power simulators that generates power traces given a C file as input [22]. The simulator, however, is not specific to any particular architecture. Reparaz also described a simulator generating power traces from a high-level C description of a cryptographic algorithm [20]. ELMO (Emulator for Power Leakage for Cortex M0) was introduced by McCann et al. for the Cortex-M0 and M4 processor families [16] whose program takes as input a compiled binary object file. Le Corre et al. proposed the first leakage simulator MAPS (Micro-Architectural Power Simulator) for the ARM Cortex-M3 Processors [10]. This work accounts for the the inter-instruction dependency of the power consumption by utilizing a more refined micro-architectural model of the target processor. Specifically, it models all pipeline registers and validates these models through simulations with an HDL description of the target micro-architecture.

There are two common cases of dynamic power consumption that we exploit, as they correlate with the intermediate values computed in the processor's core:

1. Register-type instructions typically read two values from the register file, compute an arithmetic or logical operation on them, and eventually store the result back in a register. This naturally causes the value stored in the destination register to be updated. Let us use $\mathsf{R1} \leftarrow \mathsf{R2} \oplus \mathsf{R3}$ as an example register-type instruction XOR, and denote the value of R1 before and after the execution of the XOR by a and b respectively. Then, some portion of the dynamic power consumption depends on the amount of bits that needs to be flipped when R1 goes through the transition $a \rightarrow b$. Therefore, in the collected power trace, if we focus on the special point in time that corresponds to this instruction's execution, we can find the correlation between the Hamming weight of $a \oplus b$, i.e., $H(a \oplus b)$, and the consumed power. This was referred to as Hamming-distance model by Mangard et al. [14].
2. Memory-type instructions either bring a value from the memory into a register, or store a register value in a specified memory location. These operations cause the memory bus to be driven with the data to be stored (the bus is

usually pre-charged to a value that is either all zero or all one logic values). Let us use $[\mathsf{ADDR\ PT}] \leftarrow \mathsf{R1}$ as an example of memory-type instruction, where $[\mathsf{ADDR\ PT}]$ denotes the address of the plaintext byte in the memory. Then, the execution of the store instruction causes a dynamic power consumption that correlates with the amount of logic one values in $\mathsf{R1}$, if the bus is initially pre-charged to all zeroes. In other words, $H(\mathsf{R1})$ correlates with the measured power value at particular point in time that corresponds to the store instruction. This was referred to as Hamming-weight model by Mangard et al. [14].

As our main motivation in this paper is not to investigate the relationship between the power consumption and the intermediate values, but rather use the established model as an abstract tool, this intuition will suffice for the remainder.

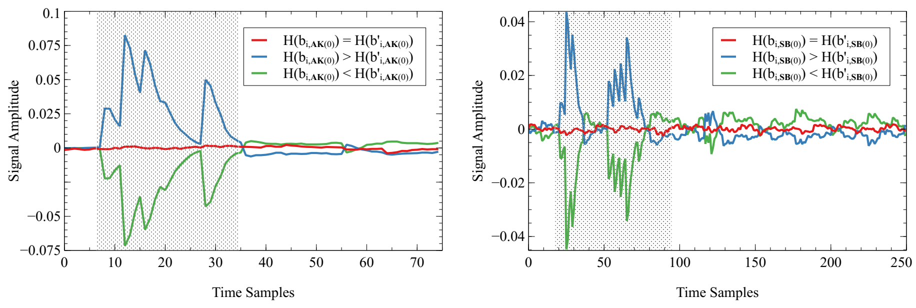
Definition 3 (Power Trace). *Let* $\mathrm{Exp}(b_{i,F(j)})$ *be an experiment that obtains a power trace from the computation of the value* $b_{i,F(j)}$*, i.e., the* i*-th state byte of the* j*-th round during the computational layer* F *for* $i \in [0, 15]$*,* $j \in [1, 10]$ *and* $F \in \{\mathsf{AK}, \mathsf{SB}, \mathsf{PB}, \mathsf{MC}\}$*. Since computing any particular layer* F *is typically carried out by multiple instructions, let us denote by* $E(b_{i,F(j)})$ *the power signal recorded during the computation of byte* $b_{i,F(j)}$*, e.g., at the moment it is placed on an initially reset bus. Similarly, we define* $\overline{E}(b_{i,F(j)})$ *as the averaged power signal over multiple runs.* [3]

An experimental observation is that $\overline{E}(b_{i,\mathsf{AK}(j)}) > \overline{E}(b'_{i,\mathsf{AK}(j)})$ if and only if $H(b_{i,\mathsf{AK}(j)}) > H(b'_{i,\mathsf{AK}(j)})$ for a large enough number of repetitions where $H(b_{i,F(j)})$ is the Hamming weight of the i-th state byte of the j-th round after the layer F. This follows from the Hamming weight model of power consumption. Analogously, $\overline{E}(b_{i,\mathsf{SB}(j)}) > \overline{E}(b'_{i,\mathsf{SB}(j)})$ if and only if $H(b_{i,\mathsf{SB}(j)}) > H(b'_{i,\mathsf{SB}(j)})$. This observation is validated in Figure 3 for $\mathsf{AK}(0)$ and $\mathsf{SB}(1)$ on our custom AES^* implementation but can also be observed on most byte-based implementations on both 8-bit and 32-bit architectures.

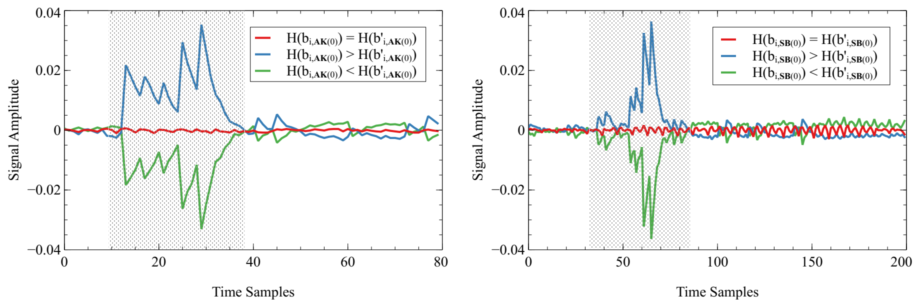
2.3 Key Recovery

Naturally, the first step of reverse-engineering an undisclosed AES^* structure involves recovering the encryption key. This is a straightforward procedure as it is directly possible to target the whitening key addition before the first round function, meaning that we measure the power trace $\overline{E}(b_{i,\mathsf{AK}(0)})$ for each state byte. We have $b_{i,\mathsf{AK}(0)} = p_i + k_i$, consequently if $\overline{E}(b_{i,\mathsf{AK}(0)}) < \overline{E}(b'_{i,\mathsf{AK}(0)})$ for all $b'_{i,\mathsf{AK}(0)} \in \mathbb{F}_{256} \setminus \{b_{i,\mathsf{AK}(0)}\}$, then $b_{i,\mathsf{AK}(0)} = 0$, or in other words, $p_i = k_i$. The key recovery algorithm thus tests whether a plaintext p_i yields $H(b_{i,\mathsf{AK}(0)}) = 0$. The entire key recovery routine for one byte is given in Algorithm 1. We remark that Algorithm 1 can be modified into a procedure that recovers the Hamming

[3] For the remainder of this text, we assume that a signal $\overline{E}(b_{i,F(j)})$ corresponds to a plaintext p, while $\overline{E}(b'_{i,F(j)})$ refers to p'.



(a) ATXMEGA128D4



(b) STM32F303

Fig. 3. Differential power traces $\overline{E}(b_{i,\mathsf{AK}(0)}) - \overline{E}(b'_{i,\mathsf{AK}(0)})$ and $\overline{E}(b_{i,\mathsf{SB}(1)}) - \overline{E}(b'_{i,\mathsf{SB}(1)})$ corresponding to different Hamming weight distances on 8-bit ATXMEGA128D4 and 32-bit STM32F303 architectures.

weight of $b_{i,\mathsf{AK}(0)}$ and $b_{i,\mathsf{SB}(1)}$ with identical complexity by simply counting how many traces exhibit a higher, lower and equal power consumption as shown in Algorithm 2. This property will be useful in Section 3.2 and Section 3.3.

The lookup table L in Algorithm 2 is related to the distribution of the Hamming weight of a random variable over $\mathbb{F}_{256}$, which was mentioned in [14, Table 4.1], where $L^{-1}(i) = \sum_{b\in\mathbb{F}_{256}} 1_{H(b)>H(i)} - \sum_{b\in\mathbb{F}_{256}} 1_{H(b)<H(i)}$. For any byte $b \in \mathbb{F}_{256}$, it essentially counts the difference of the number of $b' \in \mathbb{F}_{256} \setminus \{b\}$ for which $\overline{E}(b) > \overline{E}(b')$ and $\overline{E}(b) < \overline{E}(b')$. Since $\overline{E}$ is correlated with the Hamming weight, the method faithfully recovers $H(b)$ using L if the power traces are adequately de-noised. A slightly modified version of Algorithm 1 can be used to uniquely identify $b_{i,\mathsf{AK}(0)}$, $b_{i,\mathsf{SB}(1)}$ such that their Hamming weight is either zero or eight. As $t_{\min}$ already represents the byte whose Hamming weight is zero. Similarly, $t_{\max} = \arg\max J$ is equal to the byte with Hamming weight eight.

Parallelization. Due to the fact that it takes around $\alpha \times 2^8$ traces to recover a single key byte using Algorithm 1, it should take $\alpha \times 2^{12}$ for the complete 16-byte key. However, it is possible to parallelize the key recovery procedure for multiple key bytes at once. The idea is to have an index set $I \subset [0, 15]$, and query the 2^8 plaintexts $p_{i,j} = i,\ \forall j \in I$ and $p_{i,j} = 0,\ \forall j \notin I$, instead of a singleton j

Algorithm 1 Recover i-th Key Byte

```
   ▷ Choose a plaintext $p$ and initialize an empty array $J$ of size 256.
1: $p \in \mathbb{F}_{256}^{4\times 4}$, $J \leftarrow \{\cdot\}$
2: for $t \in \mathbb{F}_{256}$ do
   ▷ Replace $i$-th byte of $p$ with $t$, encrypt $p$ and obtain a stable power trace.
3:     $p_i \leftarrow t$, $e \leftarrow \overline{E}(b_{i,\mathsf{AK}(0)})$, $J(t) \leftarrow e$
4: $t_{\min} = \arg\min J$
5: return $t_{\min}$
```

(here $p_{i,j}$ implies the j-th byte of the i-th plaintext for $i \in [0, 255]$). The key recovery algorithm again tests whether a plaintext $p_{i,j}$ yields $H(b_{i,\mathsf{AK}(j)}) = 0$ for some $j \in I$. We have observed that if I does not contain consecutive indices then the power peaks corresponding to the round key addition are reasonably spaced apart in the time axis, allowing for efficient identification of the j-th peak only by visual inspection. As a consequence, if we repeat the process for $I = \{0, 2, 4, \ldots, 14\}$ and then $\{1, 3, 5, \ldots, 15\}$ we can recover the entire key in two runs.

Complexity. Since by parallelization we recover eight key bytes using $\alpha \times 2^8$ traces, we need $\alpha \times 2^9$ traces for the complete key.

The reader will note that it is possible to further accelerate the proposed key recovery procedure by utilizing bit-wise differentials. Let $p = 0$ be the all-zero plaintext with corresponding power trace for the first byte after the key addition $\overline{E}(b_{0,\mathsf{AK}(0)})$. Similarly, let $p' = p + (1 \ll j)$ for $j \in [0, 7]$ be the plaintext where all bits are set to zero except the j-th bit of the first plaintext byte with respective power trace $\overline{E}(b'_{0,\mathsf{AK}(0)})$. Clearly, if $\overline{E}(b_{0,\mathsf{AK}(0)}) < \overline{E}(b'_{0,\mathsf{AK}(0)})$, then the j-th bit of k_0 is zero. On the other hand, an inequality $\overline{E}(b_{0,\mathsf{AK}(0)}) > \overline{E}(b'_{0,\mathsf{AK}(0)})$ indicates that that the j-th bit of k_0 is equal to one. Repeating this for all j yields the

Algorithm 2 Recover Hamming Weight $H(b)$ for $b \in \{b_{i,\mathsf{AK}(0)}, b_{i,\mathsf{SB}(1)}\}$

```
   ▷ Initialize a lookup table $L$ and choose a plaintext $p$ for which we want to calculate
   either $H(b_{i,\mathsf{AK}(0)})$ or $H(b_{i,\mathsf{SB}(1)})$.
1: $L \leftarrow \{\ 255 : 0,\ 246 : 1,\ 210 : 2,\ 126 : 3,\ 0 : 4,\ -126 : 5$
           $-210 : 6,\ -246 : 7,\ -255 : 8\ \}$
2: $p \in \mathbb{F}_{256}^{4\times 4}$, $e \leftarrow \overline{E}(b)$, $h \leftarrow 0$
3: for $t \in \mathbb{F}_{256}$ do
   ▷ Replace the $i$-th byte of $p$ with $t$ and extract the averaged power trace. Count
   how many $t$ have a larger/smaller Hamming weight.
4:     $p_i \leftarrow t$, $e' \leftarrow \overline{E}(b)$
5:     if $e' < e$ then $h \leftarrow h - 1$.
6:     else if $e' > e$ then $h \leftarrow h + 1$.
7: Find $h_0$ in the set $\{255, 246, 210, 126, 0, -126, -210, -246, -255\}$ such that $|h - h_0|$
   is minimized.
8: return $L(h_0)$
```

full key byte k_0 in $\alpha \times 2^3$ encryptions, which again can be parallelized in an analogous fashion as done before with Algorithm 1 in order to recover multiple key bytes in a single iteration.

3 Reverse-Engineering AES-Like Ciphers

Having established the preliminaries, we proceed with our recovery algorithms for the byte permutation PB, the matrix M of the diffusion layer MC and ultimately the lookup table T of the nonlinear substitution layer.

3.1 Partial Π Recovery

The key recovery algorithm exploited the correlation between the Hamming distance of two values and their respective power consumption. This connection implies that any differential introduced in the plaintext that is diffusing through the rounds of the cipher incurs either a power spike or drop at specific points. The utilization of this phenomenon in attacks is a relatively recent addition to the large assortment of side-channel assisted cryptanalytic attacks and was first introduced by Breier et al. with an attack on $\mathsf{PRESENT}$[6]. The detection of differentially active bytes and columns lays the groundwork for our algorithms that recover Π in the byte permutation layer PB, M as part of the linear diffusion layer MC and the S-box T in the substitution layer.

Definition 4 (Differential Activity). *Denote by $\delta_{i,F(j)} \in \{\square, \blacksquare\}$ an indicator that signals whether a state byte is differentially active (with $\blacksquare$ representing an active byte). Analogously, let $\Delta_{i,F(j)} \in \{\square, \blacksquare\}$ be an indicator for differentially active columns.*

A direct approach that uniquely recovers Π consists in injecting a difference in a single plaintext byte $p_i + p'_i = d$ such that $\delta_{j,\mathsf{PB}(1)} = \blacksquare$ is observable in the differential power trace $\overline{E}(b_{j,\mathsf{PB}(1)}) - \overline{E}(b'_{j,\mathsf{PB}(1)})$ at some byte position $j \in \{0, \ldots, 15\}$. However, this method might not be reliable in certain implementations for the following reasons:

1. Depending on the implementation, the PB and MC operations may be combined together so that a distinct region in the trace segregating the PB layer may not be deducible.
2. Even if the PB region is clearly separated, 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 PB operation is necessary.

Instead, we will observe the peaks in the differential traces during round key addition $\overline{E}(b_{i,\mathsf{AK}(1)}) - \overline{E}(b'_{i,\mathsf{AK}(1)})$ of the first round or the substitution layer of the second round $\overline{E}(b_{i,\mathsf{SB}(2)}) - \overline{E}(b'_{i,\mathsf{SB}(2)})$. If the permutation function Π is such that i-th byte 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 first round MC, the j-th column becomes active, which shows up as a sequence of four spikes after the second 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 PB operation. The diffusion of a single active plaintext byte into an active column $\Delta_{j,\mathsf{AK}(1)} = \blacksquare$ is shown in Figure 4. Furthermore, the experimental detection of an active column on actual hardware is given in the plots of Figure 5.

At this point, we do not yet have the precise description of Π but only the the column to which each byte is mapped. The full permutation is recovered alongside the diffusion matrix M and the S-box T in the following sections.

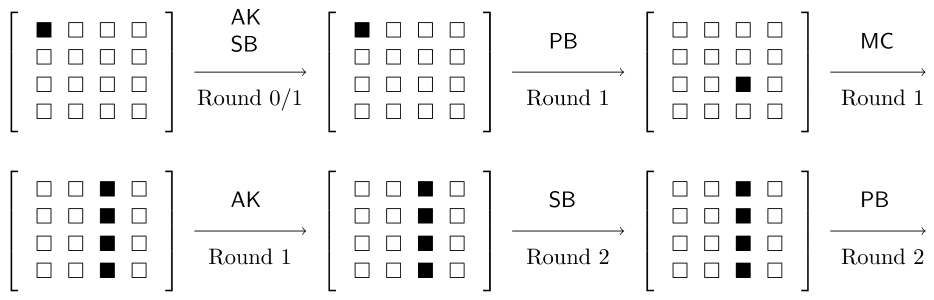
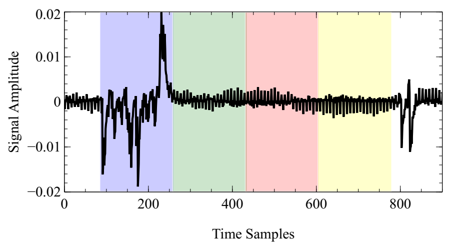


Fig. 4. Diffusion of a single active byte during the initial computational layers with $\Pi(0) = 10$. $\delta_{l,\mathsf{AK}(1)} = \blacksquare$ and $\delta_{l,\mathsf{SB}(2)} = \blacksquare$ for $8 \leq l \leq 11$ are observable as four spikes in the differential power trace (see Figure 5), i.e., $\overline{E}(b_{l,\mathsf{AK}(1)})$ - $\overline{E}(b'_{l,\mathsf{AK}(1)})$ and $\overline{E}(b_{l,\mathsf{SB}(2)})$ - $\overline{E}(b'_{l,\mathsf{SB}(2)})$.

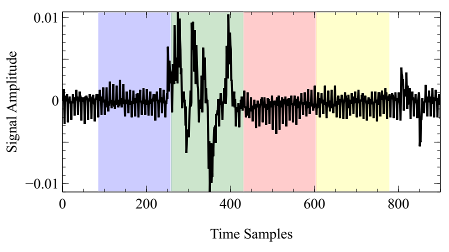
Complexity. Recovering Π up to column permutations exhibits a worst-case complexity of $\alpha \times 32$ traces, i.e., two averaged power traces are required for each state byte.

3.2 Finding 255 Candidates for M

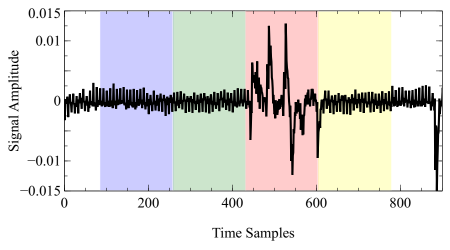
Given the unknown matrix from (1), we proceed in multiple steps with differentials on the certain specific locations after the substitution layer of the first round. More specifically, we are interested in plaintext differentials that diffuse to two active bytes after the PB operation of the first round, e.g., $\delta_{0,\mathsf{PB}(1)} = \blacksquare$, $\delta_{1,\mathsf{PB}(1)} = \blacksquare$, $\delta_{2,\mathsf{PB}(1)} = \square$, $\delta_{3,\mathsf{PB}(1)} = \square$. With some probability, such a difference leads to three active bytes in the first state column after the MC layer of the first round, e.g., $\delta_{0,\mathsf{MC}(1)} = \square$, $\delta_{1,\mathsf{MC}(1)} = \blacksquare$, $\delta_{2,\mathsf{MC}(1)} = \blacksquare$, $\delta_{3,\mathsf{MC}(1)} = \blacksquare$. In the following, let d_0, d_1, d_2, d_3 denote the four differentials in the first column after



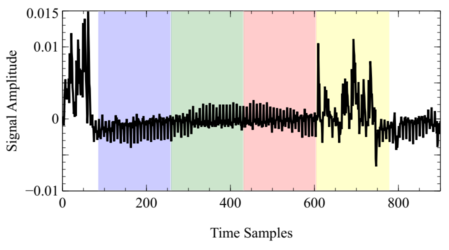
(a) $\Delta_{0,\mathsf{SB}(2)} = $ ■



(b) $\Delta_{1,\mathsf{SB}(2)} = $ ■



(c) $\Delta_{2,\mathsf{SB}(2)} = $ ■



(d) $\Delta_{3,\mathsf{SB}(2)} = $ ■

Fig. 5. Differential power traces $\overline{E}(b_{i,\mathsf{SB}(2)})$ - $\overline{E}(b'_{i,\mathsf{SB}(2)})$ for $0 \leq i \leq 15$ on the 32-bit STM32F303 platform for different active state columns. The color coding indicates the time frame during which a state column is computed (blue for the first and yellow for the fourth column). The plots for the ATXMEGA128D4 architecture are given in the appendix.

the PB layer of the first round, i.e.,

$$d_0 = b_{0,\mathsf{PB}(1)} + b'_{0,\mathsf{PB}(1)},\ \ d_1 = b_{1,\mathsf{PB}(1)} + b'_{1,\mathsf{PB}(1)},$$
$$d_2 = b_{2,\mathsf{PB}(1)} + b'_{2,\mathsf{PB}(1)},\ \ d_3 = b_{3,\mathsf{PB}(1)} + b'_{3,\mathsf{PB}(1)}.$$

In order to detect whether two active bytes in the first column activate three bytes after the multiplication by M, we can check the differential power traces $\overline{E}(b_{i,\mathsf{AK}(1)}) - \overline{E}(b'_{i,\mathsf{AK}(1)})$ or $\overline{E}(b_{i,\mathsf{SB}(2)}) - \overline{E}(b'_{i,\mathsf{SB}(2)})$ for $0 \leq i \leq 3$ for the occurrence of spikes and drops. It is important to remark that since we only know to which column a byte is shifted during PB it is not possible to infer which two bytes are actually active in the first column. Further note that we do not yet have the exact description of Π. 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 PB. We have already determined the values of u_0, u_1, u_2, u_3 up to a permutation of the 4 elements. As such, this means we have narrowed down the exact values of u_i for $i = 0 \to 3$ to a set of $4! = 24$ candidates.

Now assume that we lock one of the 24 possible choices of the four-tuple of indices u_0, u_1, u_2, u_3 and proceed in the following way:

1. Fix plaintext bytes $p_{u_2} = p'_{u_2},\ p_{u_3} = p'_{u_3}$ to some values in $\mathbb{F}_{256}$.

2. Use Algorithm 2 to find plaintext bytes p_{u_0}, p_{u_1} for which $H(b_{u_0,\mathsf{SB}(1)}) = 0$ and $H(b_{u_1,\mathsf{SB}(1)}) = 0$.
3. Similarly, find p'_{u_0} that yields $H(b'_{u_0,\mathsf{SB}(1)}) = 8$, which gives us a differential $d_0 = 255$.
4. Subsequently, iterate over all $p'_{u_1} \in \mathbb{F}_{256} \setminus \{p_{u_1}\}$ and check whether $\delta_{0,\mathsf{MC}(1)} = \square$. This can be done by checking for the absence of any peaks in the differential power traces $\overline{E}(b_{i,\mathsf{AK}(1)}) - \overline{E}(b'_{i,\mathsf{AK}(1)})$ or $\overline{E}(b_{i,\mathsf{SB}(2)}) - \overline{E}(b'_{i,\mathsf{SB}(2)})$.
5. Such an occurrence only happens for a single p'_{u_1} for which we then calculate the Hamming weight $H(b'_{u_1,\mathsf{SB}(1)}) = H(d_1) = H(x_1) = w_1$.

Consequently, we have $d_0 = 255,\ d_1 = x_1,\ d_2 = 0,\ d_3 = 0$, which corresponds to the relation

$$255a + x_1 b = 0 \ \rightarrow \ 255a = x_1 b.$$

By appropriately choosing different differentials d_0, d_1, d_2, d_3, it is possible to infer more relations for the same choice of indices u_0, u_1, u_2, u_3 as shown in Table 2.

Table 2. Ten choices of differentials d_0, d_1, d_2, d_3 to obtain relations between the x_i and the unknown M coefficients. Note that only the Hamming weight of the x_i, i.e., $H(x_i) = w_i$ are known but not their actual values. A graphical schematic of the first four steps is given in Figure 6.

Step	d_0	d_1	d_2	d_3	$\delta_{i,\mathsf{MC}(1)} = \square$	Relation
1	255	x_1	0	0	$i = 0$	$255a = x_1 b$ (2)
2	x_2	255	0	0	$i = 0$	$255b = x_2 a$ (3)
3	255	x_3	0	0	$i = 1$	$255d = x_3 a$ (4)
4	x_4	255	0	0	$i = 1$	$255a = x_4 d$ (5)
5	255	x_5	0	0	$i = 2$	$255c = x_5 d$ (6)
6	x_6	255	0	0	$i = 2$	$255d = x_6 c$ (7)
7	255	x_7	0	0	$i = 3$	$255b = x_7 c$ (8)
8	x_8	255	0	0	$i = 3$	$255c = x_8 b$ (9)
9	255	0	x_9	0	$i = 0$	$255c = x_9 a$ (10)
10	x_{10}	0	255	0	$i = 0$	$255a = x_{10} c$ (11)

The ten inferred relations from the side-channel observations can be combined with each other to yield a set of filter equations as listed in Table 3.

It is possible to computationally verify that, given the filters from Table 3 alongside the set of recovered Hamming weights $H(x_i) = w_i$, there will always be a unique solution for all x_i whenever the indices u_0, u_1, u_2, u_3 are correctly guessed. In particular, filtering out wrong x_i proceeds in the following loop:

1. Select a set of ten bytes $b_1, \ldots, b_{10}$ with $b_i \in \mathbb{F}_{256}$ such that $H(b_i) = w_i$.
2. If the selected set satisfies the filter equations in Table 3, then retain them as the solution for the x_i and return. Otherwise, repeat from the first step.

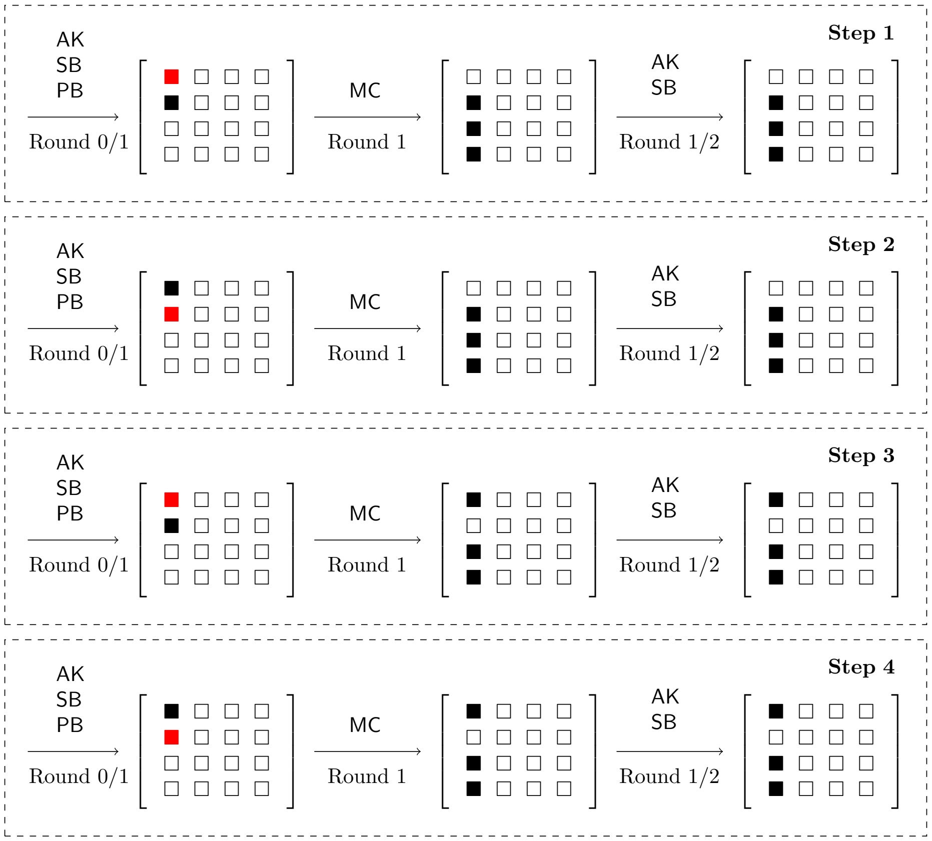
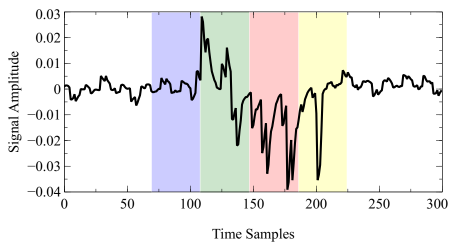
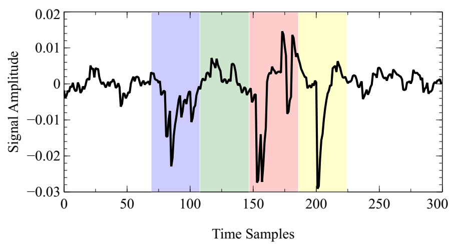


Fig. 6. First four steps of the linear diffusion layer recovery. Red squares indicate differentials of value 255.

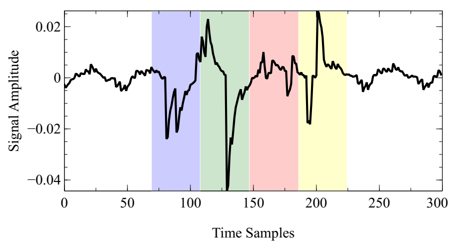
From here we can get $2^8 - 1$ solutions for M as follows: we freely choose a to be any non-zero byte. Then $b,\ c,\ d$ are obtained from above as $b = 255^{-1} \cdot x_2 \cdot a$, $c = 255^{-1} \cdot x_9 \cdot a$ and $d = 255^{-1} \cdot x_3 \cdot a$. For the 23 incorrect initial guesses the situation 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 three 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 remaining solutions in the latter cases are row rotated versions of M in the opposite direction. To understand why this happens, let Π_t be the 4×4 permutation matrix that rotates a column vector by t locations for $0 \le t \le 3$ in some direction. Let $c_{i,\mathsf{PB}(1)}$ be the i-th column after PB of the first round. Note that if M is a circulant matrix, then $M \cdot \Pi_t^{-1}$ is also a circulant matrix, in which the rows of M are rotated t locations in the opposite direction. Since $M \cdot c_{i,\mathsf{PB}(1)} = \left(M \cdot \Pi_t^{-1}\right) \cdot \left(\Pi_t \cdot c_{i,\mathsf{PB}(1)}\right)$, this explains that any starting guess of



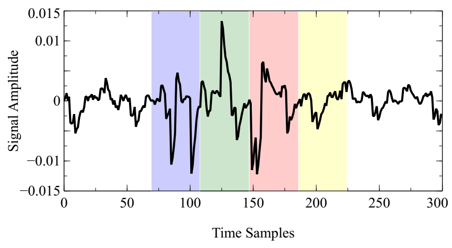
(a) $\delta_{0,\mathsf{AK}(1)} = \square$



(b) $\delta_{1,\mathsf{AK}(1)} = \square$



(c) $\delta_{2,\mathsf{AK}(1)} = \square$



(d) $\delta_{3,\mathsf{AK}(1)} = \square$

Fig. 7. Differential power traces $\overline{E}(b_{i,\mathsf{AK}(1)})$ - $\overline{E}(b'_{i,\mathsf{AK}(1)})$ for $0 \leq i \leq 3$ on the 8-bit ATXMEGA128D4 platform with a single inactive byte in the first column. The color coding indicates the four key additions of the first column (blue for the first and yellow for the fourth). The plots for the STM32F303 architecture are given in the appendix.

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.

The next question is then how to recover t and Π_t? The answer is, it is not necessary, because it is straightforward to see that for any value of t, it yields an algebraically equivalent block cipher. 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 we get four rotationally equivalent initial guesses that yield solutions for M. We first select the guesses for the four 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.

Complexity. Identifying plaintext bytes p_{u_0}, p_{u_1} and p_{u_2} that facilitate the zero images $H(b_{u_0,\mathsf{SB}(1)}) = 0$, $H(b_{u_1,\mathsf{SB}(1)}) = 0$ and $H(b_{u_2,\mathsf{SB}(1)}) = 0$ using Algorithm 1 requires $3 \times \alpha \times 2^8$ traces as it only needs to be done in the first and ninth step. Similarly, it requires $3 \times \alpha \times 2^8$ to find plaintext bytes that yield $d_i = 255$. In the worst case, it takes $10 \times \alpha \times 2^8$ traces to find the occurrence of an inactive byte in the first column in each step and ultimately another $10 \times \alpha \times 2^8$ encryptions to find the x_i. Hence, the ten steps have a cumulative worst-case complexity of $26 \times \alpha \times 2^8$ traces. Finally, the whole procedure is repeated 4×24 times for each state column and each choice of u_0, u_1, u_2, u_3, yielding a total worst-case complexity of $4 \times 24 \times 26 \times \alpha \times 2^8 \approx \alpha \times 2^{20}$ traces.

Table 3. Nine filter equations derived from the obtained relations in Table 2.

Combination	Filter	Combination	Filter
(2), (3)	$x_1x_2 = 255^2$	(4), (5)	$x_3x_4 = 255^2$
(6), (7)	$x_5x_6 = 255^2$	(8), (9)	$x_7x_8 = 255^2$
(10), (11)	$x_9x_{10} = 255^2$	(2), (8), (10)	$x_1x_7x_9 = 255^3$
(3), (9), (11)	$x_2x_8x_{10} = 255^3$	(3), (5), (7), (9)	$x_1x_3x_5x_7 = 255^4$
(4), (6), (8), (10)	$x_2x_4x_6x_8 = 255^4$	-	-

3.3 Substitution Layer Recovery

Ultimately, to recover the hidden substitution table, we fix one of the 255 candidates of M recovered in the previous section and limit ourselves once again to plaintext differentials that diffuse onto a single column after the PB operation and then converge into a single active byte after MC as shown in Figure 8.

This convergence property was a cornerstone of the See-in-the-Middle attack on partially masked $\mathsf{AES\text{-}128}$ implementations in [5] where the authors experimentally verified that it occurs with probability 2^{-22} and thus necessitates on average $2^{11.5}$ encryptions. Mathematically, a convergence onto the first byte of the column only happens when the differential output of the substitution layer is of the following form:

$$\begin{aligned} b_{0,\mathsf{SB}(1)} + b'_{0,\mathsf{SB}(1)} &= T(p_0 + k_0) + T(p_0 + k_0 + d_0) = e\lambda, \\ b_{1,\mathsf{SB}(1)} + b'_{1,\mathsf{SB}(1)} &= T(p_1 + k_1) + T(p_1 + k_1 + d_1) = f\lambda, \\ b_{2,\mathsf{SB}(1)} + b'_{2,\mathsf{SB}(1)} &= T(p_2 + k_2) + T(p_2 + k_2 + d_2) = g\lambda, \\ b_{3,\mathsf{SB}(1)} + b'_{3,\mathsf{SB}(1)} &= T(p_3 + k_3) + T(p_3 + k_3 + d_3) = h\lambda, \end{aligned} \tag{12}$$

for all non-zero $\lambda \in \mathbb{F}_{256}$ and a four-tuple of differentials $d_0, d_1, d_2, d_3 \in \mathbb{F}_{256}$ where the parameters $e, f, g, h \in \mathbb{F}_{256}$ stem from the inverse of M, i.e.,

$$M^{-1} = \begin{bmatrix} e & f & g & h \\ h & e & f & g \\ g & h & e & f \\ f & g & h & e \end{bmatrix}.$$

The first step of our recovery procedure involves simplifying (12) by finding a four-tuple of plaintext bytes p_0, p_1, p_2, p_3 such that $T(p_i + k_i) = 0$, which yields

$$\begin{aligned} T(p_0 + k_0 + d_0) = e\lambda,\ T(p_1 + k_1 + d_1) = f\lambda, \\ T(p_2 + k_2 + d_2) = g\lambda,\ T(p_3 + k_3 + d_3) = h\lambda. \end{aligned} \tag{13}$$

Subsequently, we look for the occurrence of a convergence by varying the differentials d_0, d_1, d_2, d_3 and observing the differential power traces $\overline{E}(b_{i,\mathsf{AK}(2)}) - \overline{E}(b'_{i,\mathsf{AK}(2)})$ or $\overline{E}(b_{i,\mathsf{SB}(2)}) - \overline{E}(b'_{i,\mathsf{SB}(2)})$. Once found, the Hamming weight of the

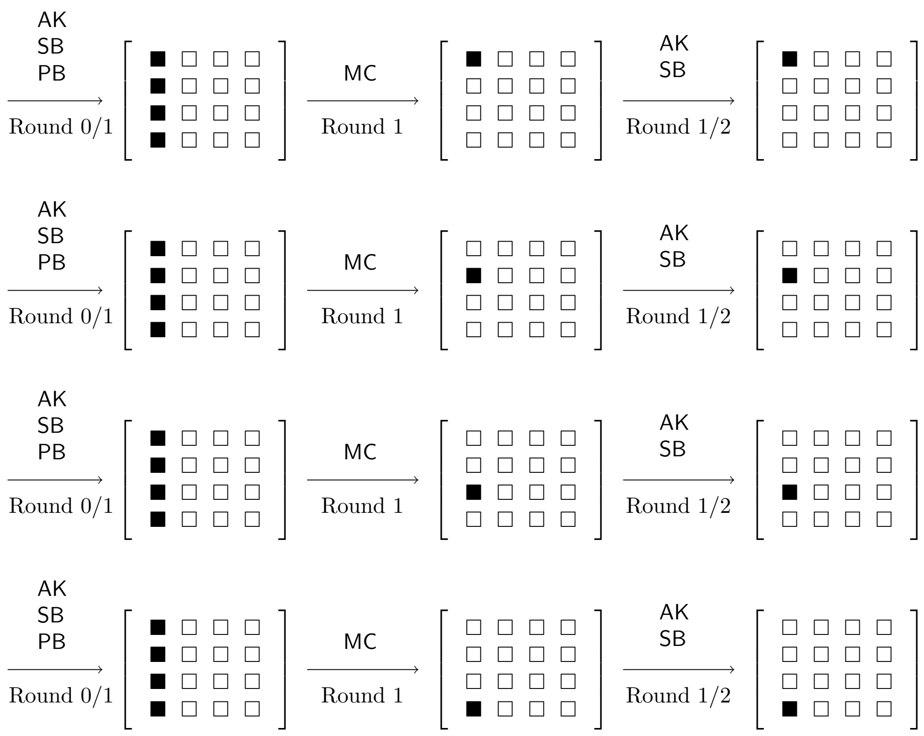


Fig. 8. Convergence of a differentially active column into a single active byte in the same column. On average, roughly $2^{11.5}$ encryptions are required for the convergence to occur.

substitution box outputs is recovered, i.e.,

$$H(T(p_0 + k_0 + d_0)) = w_0,\ H(T(p_1 + k_1 + d_1)) = w_1,$$
$$H(T(p_2 + k_2 + d_2)) = w_2,\ H(T(p_3 + k_3 + d_3)) = w_3.$$

Since at this point p_i, k_i, d_i are known, the task boils down to filling up the 256 entries of T by some method to convert the weights w_i recovered above into actual values. However, the actual values are related by (13) which can be leveraged as follows: we pre-compute a lookup table L whose λ-th entry is the tuple $L[\lambda] = [H(e\lambda), H(h\lambda), H(g\lambda), H(f\lambda)]$ for all $0 < \lambda < 256$ and infer the value of λ if $L[\lambda] = [w_0, w_1, w_2, w_3]$ for some table entry. For random values of e, f, g, h, through computer simulations we have found that more than 200 entries of L 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 entries within a few repetitions of the above procedure. Note that there are 255 candidates for M and for each one a lookup table is created yielding a potential solution for

M and T whose correctness we can verify with a plaintext-ciphertext pair from the target device.

Complexity. Recovering the zero-image in the first step requires $4 \times 2^8 = 2^{10}$ encryptions. Afterwards, for each four elements of the S-box, the convergence phenomenon 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 convergence onto a single active byte. If the coefficients e, f, g, h are chosen uniformly at random, then $\beta \approx 1.3$. This step needs to be repeated $\frac{256}{4} = 64$ times to recover all the entries of T. Hence the number of total encryptions to recover the full substitution table for a given MC matrix M is $\alpha \times 2^{10} + \alpha \times \beta \times 2^{17.5}$. Note that the power traces only need to be extracted once.

4 Future Work & Conclusion

In this work, we demonstrated the first complete and practical side-channel assisted reverse engineering attack on AES-like ciphers thus settling an open problem of whether such a recovery is feasible. The presented techniques are based on the recently introduced SCADPA methodology that combines differential power analysis with tools from conventional differential cryptanalysis. All recovery procedures were validated on two common 8-bit and 32-bit microcontrollers. Beyond the material presented in this paper, we identify the following set of open problems:

- **Non-Circulant MixColumns.** The recovery of the 255 MixColumns matrix candidates in Section 3.2 relies on the fact that M is circulant. One could also imagine an attack that is applicable to invertible non-circulant matrices in $\mathbb{F}_{256}^{4\times 4}$ as was the assumption in [9].
- **Protected Implementations.** Our recovery procedures apply to unprotected byte-wise implementations, however masking and shuffling are common side-channel countermeasures that attempt to prevent deductions from power measurements and thus also complicate any reverse engineering efforts.
- **T-Table Implementations.** The S-box recovery routine of Section 3.3 relies on the assumption that the Hamming weight of substituted bytes after the SB layer is recoverable via Algorithm 2. This may not be the case anymore in T-table implementations that merge the S-box with the MixColumns layer in a set of lookup tables.

Acknowledgements. We wish to thank Thomas Roche for helping us improve this paper. Fatih Balli and Subhadeep Banik are supported by the Swiss National Science Foundation (SNSF) through the Ambizione Grant PZ00P2_179921.

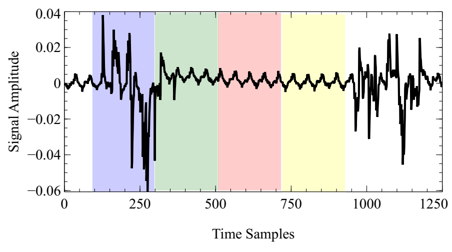
References

1. AVR-Crypto-Lib. https://wiki.das-labor.org/w/AVR-Crypto-Lib/en, accessed: 2021-07-03
2. OpenSSL. https://github.com/openssl/openssl, accessed: 2021-07-03
3. secAES. https://github.com/ANSSI-FR/secAES-ATmega8515, accessed: 2021-07-03
4. Backes, M., Dürmuth, M., Gerling, S., Pinkal, M., Sporleder, C.: Acoustic side-channel attacks on printers. In: 19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings. pp. 307–322. USENIX Association (2010), http://www.usenix.org/events/sec10/tech/full_papers/Backes.pdf
5. Bhasin, S., Breier, J., Hou, X., Jap, D., Poussier, R., Sim, S.M.: SITM: see-in-the-middle side-channel assisted middle round differential cryptanalysis on SPN block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. pp. 95–122 (2020). https://doi.org/10.13154/tches.v2020.i1.95-122
6. Breier, J., Jap, D., Bhasin, S.: SCADPA: side-channel assisted differential-plaintext attack on bit permutation based ciphers. In: Madsen, J., Coskun, A.K. (eds.) 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018. pp. 1129–1134. IEEE (2018). https://doi.org/10.23919/DATE.2018.8342180
7. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings. Lecture Notes in Computer Science, vol. 3156, pp. 16–29. Springer (2004). https://doi.org/10.1007/978-3-540-28632-5_2
8. Clavier, C.: An improved SCARE cryptanalysis against a secret A3/A8 GSM algorithm. In: McDaniel, P.D., Gupta, S.K. (eds.) Information Systems Security, Third International Conference, ICISS 2007, Delhi, India, December 16-20, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4812, pp. 143–155. Springer (2007). https://doi.org/10.1007/978-3-540-77086-2_11
9. Clavier, C., Isorez, Q., Wurcker, A.: Complete SCARE of aes-like block ciphers by chosen plaintext collision power analysis. In: Paul, G., Vaudenay, S. (eds.) Progress in Cryptology - INDOCRYPT 2013 - 14th International Conference on Cryptology in India, Mumbai, India, December 7-10, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8250, pp. 116–135. Springer (2013). https://doi.org/10.1007/978-3-319-03515-4_8
10. Corre, Y.L., Großschädl, J., Dinu, D.: Micro-architectural power simulator for leakage assessment of cryptographic software on ARM cortex-m3 processors. In: Fan, J., Gierlichs, B. (eds.) Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10815, pp. 82–98. Springer (2018). https://doi.org/10.1007/978-3-319-89641-0_5
11. Jap, D., Bhasin, S.: 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. pp. 1–5. IEEE (2020). https://doi.org/10.1109/ISCAS45731.2020.9180848
12. Knudsen, L.R.: Dynamic encryption. J. Cyber Secur. Mobil. pp. 357–370 (2014). https://doi.org/10.13052/jcsm2245-1439.341
13. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceed-

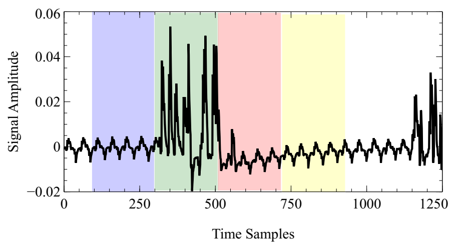
ings. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer (1999). https://doi.org/10.1007/3-540-48405-1_25

14. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
15. Mayer-Sommer, R.: Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1965, pp. 78–92. Springer (2000). https://doi.org/10.1007/3-540-44499-8_6
16. McCann, D., Oswald, E., Whitnall, C.: Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 199–216. USENIX Association (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann
17. Novak, R.: Side-channel attack on substitution blocks. In: Zhou, J., Yung, M., Han, Y. (eds.) Applied Cryptography and Network Security, First International Conference, ACNS 2003. Kunming, China, October 16-19, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2846, pp. 307–318. Springer (2003). https://doi.org/10.1007/978-3-540-45203-4_24
18. O'Flynn, C., Chen, Z.D.: Chipwhisperer: An open-source platform for hardware embedded security research. In: Prouff, E. (ed.) Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8622, pp. 243–260. Springer (2014). https://doi.org/10.1007/978-3-319-10175-0_17
19. Quisquater, J., Samyde, D.: Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In: Attali, I., Jensen, T.P. (eds.) Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2140, pp. 200–210. Springer (2001). https://doi.org/10.1007/3-540-45418-7_17
20. Reparaz, O.: Detecting flawed masking schemes with leakage detection tests. In: Peyrin, T. (ed.) Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9783, pp. 204–222. Springer (2016). https://doi.org/10.1007/978-3-662-52993-5_11
21. Rivain, M., Roche, T.: SCARE of secret ciphers with SPN structures. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I. Lecture Notes in Computer Science, vol. 8269, pp. 526–544. Springer (2013). https://doi.org/10.1007/978-3-642-42033-7_27
22. Veshchikov, N.: SILK: high level of abstraction leakage simulator for side channel analysis. In: Preda, M.D., McDonald, J.T. (eds.) Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014. pp. 3:1–3:11. ACM (2014). https://doi.org/10.1145/2689702.2689706

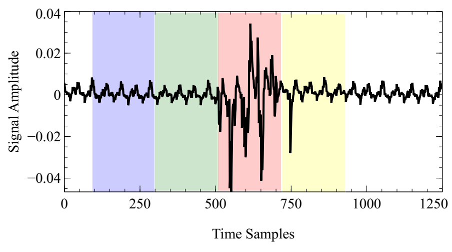
A Supplementary Plots



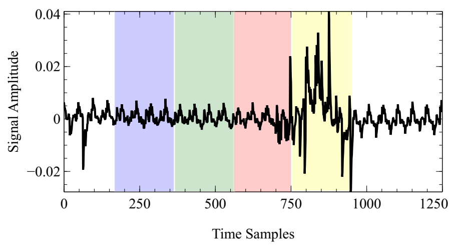
(a) $\mathit{\Delta}_{0,\mathsf{SB}(2)} = \blacksquare$



(b) $\mathit{\Delta}_{1,\mathsf{SB}(2)} = \blacksquare$

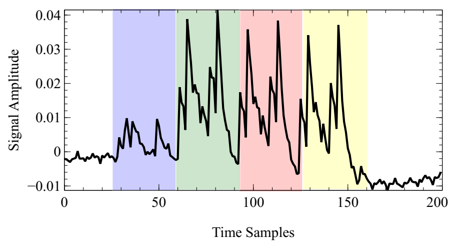


(c) $\mathit{\Delta}_{2,\mathsf{SB}(2)} = \blacksquare$

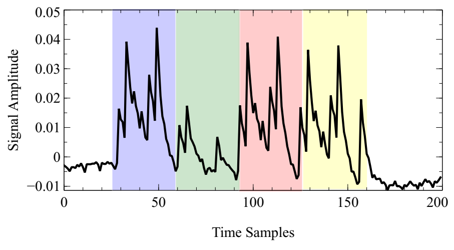


(d) $\mathit{\Delta}_{3,\mathsf{SB}(2)} = \blacksquare$

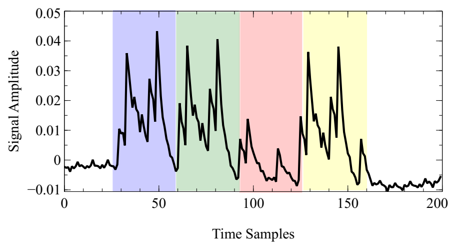
Fig. 9. Differential power traces $\overline{E}(b_{l,\mathsf{SB}(0)})$ - $\overline{E}(b'_{l,\mathsf{SB}(0)})$ on the 8-bit ATXMEGA128D4 platform for different active state columns.



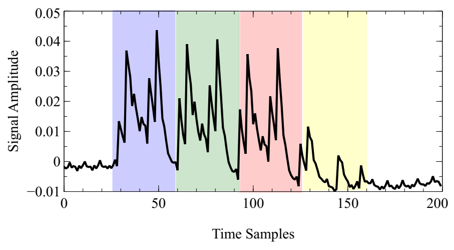
(a) $\delta_{0,\mathsf{AK}(1)} = \square$



(b) $\delta_{1,\mathsf{AK}(1)} = \square$



(c) $\delta_{2,\mathsf{AK}(1)} = \square$



(d) $\delta_{3,\mathsf{AK}(1)} = \square$

Fig. 10. Differential power traces $\overline{E}(b_{i,\mathsf{AK}(1)})$ - $\overline{E}(b'_{i,\mathsf{AK}(1)})$ for $0 \leq i \leq 3$ on the 32-bit STM32F303 platform with a single inactive byte in the first column.