Side-Channel Attacks meet Secure Network Protocols (Full Version)*

Alex Biryukov, Daniel Dinu, and Yann Le Corre

SnT, University of Luxembourg
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{alex.biryukov,dumitru-daniel.dinu,yann.lecorre}@uni.lu

Abstract. Side-channel attacks are powerful tools for breaking systems that implement cryptographic algorithms. The Advanced Encryption Standard (AES) is widely used to secure data, including the communication within various network protocols. Major cryptographic libraries such as OpenSSL or ARM mbed TLS include at least one implementation of the AES. In this paper, we show that most implementations of the AES present in popular open-source cryptographic libraries are vulnerable to side-channel attacks, even in a network protocol scenario when the attacker has limited control of the input. We present an algorithm for symbolic processing of the AES state for any input configuration where several input bytes are variable and known, while the rest are fixed and unknown as is the case in most secure network protocols. Then, we classify all possible inputs into 25 independent evaluation cases depending on the number of bytes controlled by attacker and the number of rounds that must be attacked to recover the master key. Finally, we describe an optimal algorithm that can be used to recover the master key using Correlation Power Analysis (CPA) attacks. Our experimental results raise awareness of the insecurity of unprotected implementations of the AES used in network protocol stacks.

Keywords: Side-channel attack, secure network protocol, CPA, AES

1 Introduction

Side-channel attacks use observations made during the execution of an implementation of a cryptographic algorithm to recover secret information. From the multitude of side-channel attacks, Correlation Power Analysis (CPA) [5] stands out as a very efficient and reliable technique. Its success is augmented by the minimally invasive methods employed for the acquisition of the side-channel information. Some of the most frequently used sources of side-channel leakage are the power consumption or the electromagnetic (EM) emissions of a device under attack.

* Full version of the paper published in the proceedings of ACNS 2017.

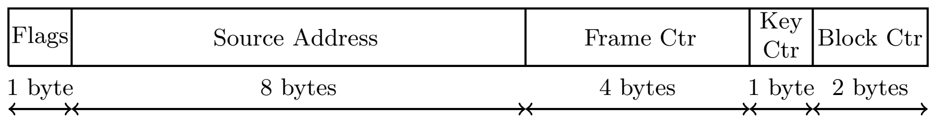


Fig. 1: The first input block for the AES-CTR and AES-CCM modes used in IEEE 802.15.4.

Nowadays, the AES [23] is the most popular symmetric cryptographic algorithm in use. It is widely deployed to secure data in transit or at rest. Various network protocols rely on the AES in different modes of operation to provide security services such as confidentiality and authenticity. The usage spectrum of the AES stretches from powerful servers and personal computers to resource constrained devices such as wireless sensor nodes. While the security of the algorithm and its implementations have been placed under scrutiny since it became the symmetric cryptographic standard, with a few notable exceptions, most of the previous work focused on the AES itself and less on the usage of the AES in complex systems.

By far, most of the experimental results reported in the side-channel literature are for implementations of the AES. They usually assume the attacker has full control of the AES input. This is not the case in a real world communication protocol, when often a major part of the input is fixed and only few bytes are variable. Moreover, sometimes the attacker cannot control these variable bytes and she has to passively observe executions of the targeted algorithm without being able to trigger encryptions of her own free will. With the notable exceptions of [16,24], the security of communication scenarios based on the AES against side-channel attacks has not been thoroughly analyzed so far. Thus, in this paper we analyze for the first time how much control of the AES input does an attacker need to recover the secret key of the cipher by performing a side-channel attack against a communication protocol.

Numerous standards for communication in the Internet of Things (IoT) such as IEEE 802.15.4 [15] and LoRaWAN [21] use the AES to encrypt and authenticate the Medium Access Control (MAC) layer frames. The 802.15.4 standard uses a variant of the AES-CCM [9,34], while LoRaWAN uses AES-CMAC [31]. The same CCM mode is used with the AES to encrypt the IPsec Encapsulating Security Payload (ESP) [14]. According to [29] the security architecture of IEEE 802.15.4 relies on four categories of security suites: none, AES-CTR, AES-CBC-MAC, and AES-CCM. A typical input for the AES-CTR and AES-CCM modes used in the IEEE 802.15.4 protocol is shown in Fig. 1. In this particular example, an attacker can manipulate up to 12 bytes of the input (`Source Address` and `Frame Counter`), while the other input bytes (`Flags`, `Key Counter` and `Block Counter`) are fixed. The attack on IEEE 802.15.4 wireless sensor nodes described in [24] assumes the control of only four input bytes (`Frame Counter`), while the remaining input bytes are constant. Thus the following question arises: *How* *many input bytes should an attacker change in the injected messages in order to fully recover the master key without triggering any network protection mechanism?*

While numerous network protocols use the AES to secure the communication between end nodes, major cryptographic libraries such as OpenSSL [25] and ARM mbed TLS [2] do not have a side-channel protected implementation of the AES for devices that do not support the AES-NI [13] instruction set as is the case with most IoT devices. Therefore, an elaborate analysis of the security of the unprotected implementations of the AES used in communication protocols is necessary. Only such a careful analysis can assess the impact of side-channel attacks on the security of real world systems using unprotected implementations of the AES.

In this work, we chose to focus on CPA attacks thanks to their efficiency and reliability. We opted for a non-invasive measurement setup and hence we selected the EM emissions of the target processor as source of side-channel leakage. The target is an ARM Cortex-M3 processor mounted on a STM32 Nucleo [32] board from STMicroelectronics. These processors are widely used for low-power applications and meet the requirements for use in the IoT.

The IoT will be a security nightmare if the whole ecosystem is not designed with security in mind. While many communication protocols for the IoT are in formative stages, the threat model of the IoT is less understood despite it is widely accepted that its attack surface is large. Although we focus on a particular side-channel attack (i.e. power/EM), other side-channel attacks such as timing, fault, cache or data remanence attacks might pose a similar or even a higher threat for the security of the IoT ecosystem. Attacks that do not exploit side-channel information, such as those used to compromise Internet-connected computers, should not be neglected since they have certain advantages over side-channel attacks. Thus, our work adds another piece to the security puzzle of the IoT by showing the need for side-channel countermeasures to prevent a somehow overlooked threat.

Research Contributions. This paper performs for the first time a thorough analysis of all possible attack scenarios against software implementations of the AES used to secure various communication protocols. Firstly, we present an algorithm for symbolic processing of a given input state of the AES. The algorithm outputs the number of rounds and the bytes that must be attacked to recover the secret key. Then, using this algorithm we perform a classification of all possible inputs depending on the number of rounds that must be attacked in order to recover the master key. The result is a set of 25 independent evaluation cases. Secondly, we describe an optimal algorithm that uses the above-mentioned symbolic representation to recover the master key of the AES using CPA attacks. The algorithm explores all possible combinations of input key bytes and discards the invalid key candidates, thus yielding only the correct master key if enough power traces with a good signal-to-noise ratio are provided. Afterwards, we evaluate the results of the attack algorithm in each of the 25 evaluation cases identified in the classification step using traces from an ARM Cortex-M3 processor.

Our results show that popular implementations of the AES found in well-known and widely used cryptographic libraries can be broken using CPA attacks. The only requirement is that a part of the AES input is known and variable, while the rest is constant, which is a common scenario in communication protocols. Knowledge of the AES implementation strategy improves the attack results, but it is not crucial. All software tools presented in this paper are in the public domain [1] to support reproducibility of results and to maximize reusability.

2 Preliminaries

2.1 Description of the AES

We give a brief description of the AES [23] to recall relevant aspects of the algorithm and to introduce the notation used in this paper. For more details on the AES algorithm, we refer the reader to the official specifications.

The AES standard uses the 128-bit block length version of the Rijndael cipher [8] with three different key lengths: 128, 192, and 256 bits. The round function is applied to the 4×4 byte state matrix 10, 12, or 14 times depending on the key length. It comprises four transformations: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. The final round function does not include the `MixColumns` transformation.

Let $s_{i,j}$ be the state byte located at row i and column j $(0 \leq i, j \leq 3)$, k_l the corresponding round key byte $(l = 16 \cdot r + i + 4 \cdot j)$ and r the round number. After application of the `AddRoundKey` transformation, each byte of the state becomes $s'_{i,j} = s_{i,j} \oplus k_l$, where the "$\oplus$" symbol denotes bitwise exclusive or of two 8-bit values. The non-linear `SubBytes` operation transforms each byte of the state using an 8-bit S-box S as follows: $s'_{i,j} = S[s_{i,j}]$. The `ShiftRows` transformation performs a rotation of row i by i bytes to the left. In the `MixColumns` transformation, a polynomial multiplication over $GF(2^8)$ is applied to each column of the state matrix. The symbol "$\bullet$" is used for multiplication of two numbers in $GF(2^8)$, while $\{01\}$, $\{02\}$, and $\{03\}$ are 8-bit vectors representing elements from $GF(2^8)$.

The key schedule expands the master key into the 16-byte round keys. The round constant array `Rcon` contains the powers of $\{02\}$ in $GF(2^8)$ as described in the specifications. The structure of the AES encryption and key schedule for a 16-byte master key are given in Appendix A.

2.2 Correlation Power Analysis

Correlation Power Analysis (CPA) [5] is a side-channel attack in which the attacker correlates the power model of a sensitive intermediate value of the target cryptographic algorithm with the measured power consumption or electromagnetic emission (EM) of the device running the target algorithm. Then, she

[1] `https://github.com/cryptolu/aes-cpa`

chooses the key hypothesis that gives the maximum correlation coefficient as the most likely key. Compared to classical Differential Power Analysis (DPA) [17] attacks, CPA attacks have several advantages in terms of efficiency, robustness and number of experiments, but are more resource demanding. Agrawal *et al.* [1] introduced the electromagnetic emissions of a target device as a source of leakage for side-channel attacks.

A CPA attack can be split into two phases: acquisition and attack. In the acquisition phase, the attacker observes and records the leakage of the target device (power consumption or electromagnetic emission) for different inputs. While the acquisition of power consumption traces requires insertion of a resistor into the circuitry of the target device to measure the voltage across it, the observation of electromagnetic emission is non-invasive; it only requires an electromagnetic probe placed in the vicinity of the leaking spot. In the attack phase, the attacker correlates these observations with the modeled power consumption of the selection function to recover the secret key. A selection function combines a known input with the secret material to be recovered.

In this work we focus on the electromagnetic emissions of an ARM Cortex-M3 processor clocked at 8 MHz running various software implementations of the AES. The acquisition was performed from a spot above the chip using a Langer RF-K 7-4 H-field probe. The signal was amplified by 30dB and fed into a Teledine LeCroy WaveRunner 8254M-MS oscilloscope sampling at 500 MS/s. For more details on the measurement setup we refer the reader to Appendix B.

2.3 Attacking Temporary Key Bytes

To attack the AES in counter mode, Jaffe introduced a technique that propagates a DPA attack to later rounds. It can be used when just few key bytes of the AES input are known and variable, while the others are fixed (constant) and unknown [16]. Next we briefly describe how the unknown fixed bytes can be incorporated into a round key byte to recover a temporary key byte. Then, using these temporary key bytes the attack can be carried into later rounds until enough round key bytes are recovered to reverse the key schedule.

Using a CPA attack an adversary can recover only those key bytes that are XORed with variable and known state bytes in the `AddRoundKey` transformation. The gist of Jaffe's technique is that an attacker can still recover a temporary key byte when an input byte of the `AddRoundKey` transformation is the result of the `MixColumns` transformation applied to at least one known and variable input byte while the other input bytes are unknown and constant.

To better illustrate how this technique works, let us consider the first state byte $s'_{0,0}$ after performing the first round function:

$$s'_{0,0} = (\{02\} \bullet s_{0,0}) \oplus (\{03\} \bullet s_{1,1}) \oplus (\{01\} \bullet s_{2,2}) \oplus (\{01\} \bullet s_{3,3}) \oplus k_{16}$$

Suppose now that the input bytes $s_{0,0}$ and $s_{1,1}$ are known and variable (key bytes k_0 and k_5 were successfully recovered using a side-channel attack on the `SubBytes` transformation of the first round), while the other input bytes ($s_{2,2}$

and $s_{3,3}$) are unknown, but fixed. Thus $s'_{0,0}$ can be written as $(\{02\} \bullet s_{0,0}) \oplus (\{03\} \bullet s_{1,1}) \oplus k'_{16}$, where the constant part is included in the temporary key k'_{16} that will be recovered by attacking the SubBytes transformation of the second round; $k'_{16} = (\{01\} \bullet s_{2,2}) \oplus (\{01\} \bullet s_{3,3}) \oplus k_{16}$. The temporary key k'_{16} enables the computation of four state bytes in the following round. In this way, the attack is carried into the next rounds until all state bytes are known; consequently, the real key bytes can be recovered.

The technique works similarly when three input bytes are known and variable. Though, when only one input byte is known and variable, the attacker will recover the same two equally likely key candidates for two bytes of the same column of the cipher state. For example, when only $s_{3,3}$ is known and variable while the other input bytes are unknown and fixed, then $s'_{0,0} = (\{01\} \bullet s_{3,3}) \oplus k'_{16}$ and $s'_{1,0} = (\{01\} \bullet s_{3,3}) \oplus k'_{17}$. Thus attacking either of the two, an attacker will get two equally likely key bytes (k'_{16} and k'_{17}). If the state bytes are not processed in order by the SubBytes transformation, the attacker will not know which key byte corresponds to $s'_{0,0}$ and which key byte corresponds to $s'_{1,0}$.

2.4 Software Implementations of the AES

There are various ways to implement the AES in software depending on the execution time, code size and RAM consumption requirements. Other factors that influence the implementation strategy are the cipher mode of operation and the number of plaintext blocks to be encrypted. Schwabe and Stoffelen [30] identified four different strategies to implement the AES in software: traditional, T-tables, vector permute, and bit slicing. In this paper, we consider the following two implementation approaches for the AES that are relevant for a secure communication protocol:

- The *straightforward implementation* (**S-box** strategy) performs the four round transformations as described above. The substitution layer is implemented using a 256-byte lookup table based on S-box S. This implementation approach is suitable for 8-bit architectures.
- The *table based implementation* (**T-table** strategy) uses four lookup tables (T_0, T_1, T_2, and T_3) of 1024 bytes each to perform the SubBytes, ShiftRows, and MixColumns operations at the cost of 16 table lookups, 16 masks and 16 XORs per round, except for the final round. A low memory alternative uses just one T-table, but performs 12 additional rotations per round. This strategy was initially described by the designers of Rijndael [8]. It leads to very fast implementations on 32-bit platforms.

We did not analyze bit-sliced or vector permute implementations because such implementations are uncommon in cryptographic libraries due to the following limitations. The bit-sliced implementations process at least two blocks in parallel and thus they can be applied only to non-feedback modes of operation. The vector permute implementations require support for vector permute instructions, but most of the resource constrained microcontrollers for the IoT do not support such instructions.

Table 1: A summary of the existing AES implementations used by open source cryptographic libraries written in C/C++. All the T-table implementations are vulnerable to the attack described in this paper.

Library	**Language**	**Version**	**Last update**	**AES-NI**	**T-table**
Botan [27]	C++	2.1.0	Apr 2017	✓	✓
cryptlib [6]	C	3.4.3	Feb 2017	✓	✓
Crypto++ [7]	C++	5.6.5	Oct 2016	✓	✓
Libgcrypt [18]	C	1.7.6	Jan 2017	✓	✓
libtomcrypt [10]	C	1.17	Apr 2017	✗	✓
libsodium [19]	C	1.0.12	Mar 2017	✓	✗
mbed TLS [2]	C	2.4.2	Mar 2017	✓	✓
Nettle [22]	C	3.3	Oct 2016	✓	✓
OpenSSL [25]	C	1.1.0e	Feb 2017	✓	✓
wolfCrypt [35]	C	3.10.2	Feb 2017	✓	✓

An analysis of the existing AES implementations used by different open source cryptographic libraries is given in Table 1. The default implementations of the AES for platforms that do not support the AES-NI [13] instructions in popular cryptographic libraries such as OpenSSL [12,25] or mbed TLS [2,11] use the T-table approach. Except for libsodium [19], all other cryptographic libraries analyzed have an implementation of the AES based on the T-table strategy. Moreover, these implementations are not protected against side-channel attacks such as DPA or cache attacks. It is well know that unprotected implementations of cryptographic algorithms are an easy target for DPA attacks. Recently, researchers from Rambus Cryptography Research Division have shown that even an unprotected software implementation based on AES-NI instructions can be attacked with DPA [28]. The T-table implementations of the AES are vulnerable to various cache attacks as shown in [20, 26]. Although the unprotected T-table implementations are vulnerable to side channel attacks, nine out of the ten libraries considered in Table 1 have such an implementation of the AES.

3 Quantifying the Leakage

Biryukov *et al.* [4] introduced the correlation coefficient difference metric to analyze the leakage of different selection functions in the context of CPA. The correlation coefficient difference δ gives the difference between the correlation coefficient of the correct key and the correlation coefficient of the most likely key guess, where the most likely key is different from the correct key.

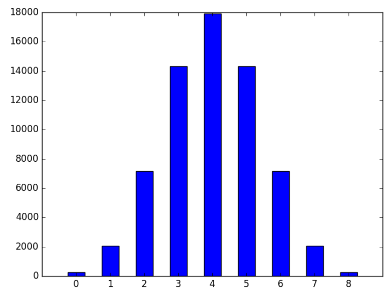
We use the correlation coefficient difference to quantify the leakages of two selection functions: φ_1 based on the AES S-box and φ_2 based on the AES T-table. The two selection functions are defined below:

$$\varphi_1 : \mathbb{F}_2^8 \mapsto \mathbb{F}_2^8, \quad \varphi_1(x \oplus k) = S(x \oplus k)$$

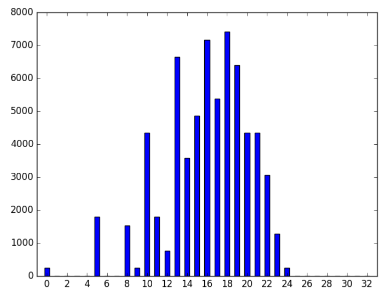
$$\varphi_2 : \mathbb{F}_2^8 \mapsto \mathbb{F}_2^{32}, \quad \varphi_2(x \oplus k) = T(x \oplus k)$$

Table 2: Correlation coefficient difference δ between the correlation of the correct key and the correlation of the most likely key [4], for different Hamming weights of the correct key; $\bar{\delta}$ and $\mathsf{SE}_{\bar{\delta}}$ are the mean and the standard error for a 95% confidence interval, respectively. The leakages are acquired from an ARM Cortex-M3 processor.

	Correct key									$\bar{\delta}$	$\mathsf{SE}_{\bar{\delta}}$
	`0x00`	`0x01`	`0x03`	`0x07`	`0x0F`	`0x1F`	`0x3F`	`0x7F`	`0xFF`		
φ_1	0.146	0.126	0.108	0.156	0.126	0.960	0.153	0.140	0.084	0.126	0.020
φ_2	0.104	0.072	0.143	0.074	0.070	0.126	0.078	0.044	0.028	0.082	0.028



(a) S-box



(b) T-table

Fig. 2: Distribution of the Hamming weight of the output of the AES (a) S-box and (b) T-table for all possible input combinations.

When using simulated leakages, the values of the correlation coefficient difference are 0.813 and 0.7 for φ_1 and φ_2, respectively. These values are the same regardless of the correct key used. In the simulated environment, the leakages of the two selection functions are very high and the difference between them is about 14% of the first one. On the other hand, the mean correlation coefficient difference $\bar{\delta}$ for different values of the correct key using leakages acquired from an ARM Cortex-M3 processor is given in Table 2. The measurements were performed at a sampling rate of 500 MS/s using assembly implementations of the analyzed selection functions. Increasing the sampling rate to 1 GS/s does not significantly improve the results. The mean correlation coefficient difference $\bar{\delta}$ is positive for both selection functions, which means they leak enough information about the secret key such that an attacker can recover the key byte using only one key guess. In practice, the selection function based on the AES S-box leaks about 50% more than the selection function based on the AES T-table. This can be explained by the distribution of the Hamming weight of the two selection functions for all possible input combinations (See Fig. 2).

The reader can easily observe in Fig. 2a that the distribution of values in the case of the AES S-box follows a binomial distribution. On the other hand, the distribution of values in the case of the AES T-table shown Fig. 2b does not resemble a binomial distribution. Moreover, there are 14 out of 32 possible output values that never occur (i.e. 1, 2, 3, 4, 6, 7, 25, 26, 27, 28, 29, 30, 31, and 32). These differences between the distribution of the Hamming weights of the output of the two selection functions φ_1 and φ_2 explain why the leakage of φ_1 is greater than the leakage of φ_2 as quantified using the correlation coefficient difference. This means that a CPA attack against an implementation based on the T-table strategy requires more effort (i.e. power traces) compared to a CPA attack against an implementation based on the S-box strategy.

4 Generating the Evaluation Cases

In this section we describe the algorithm for symbolic processing of a given initial state to determine the number of rounds required to recover the master key of the AES. We used this algorithm to explore all possible attack cases and to choose the relevant evaluation cases for our scenario. The algorithm relies on the following symbolic representation of a byte situated at row i and column j of the AES state at the start of round r:

$$s^r_{i,j} = \begin{cases} 0, & \text{the corresponding key byte can not be recovered} \\ 1, & \text{the corresponding key byte can be recovered} \\ -n, & n \text{ temporary key bytes can be recovered} \end{cases}$$

Thus, the byte $s^r_{i,j}$ is variable if its symbolic representation is different from 0 and fixed (constant) when its symbolic representation is 0. Due to the `MixColumns` transformation, each column of the state at round $r+1$ can be expressed as a function of four bytes of the state at round r. At the start of round $r+1$ each byte of the state is updated using the following rules:

- if the number of variable input bytes is 0, then the symbolic representation of the output byte is set to 0;
- if the number of variable input bytes is 1, then the symbolic representation of the output byte is updated as follows:
  - if the variable input byte is multiplied by $\{01\}$ in the `MixColumns` transformation, then the symbolic representation of the output byte is set to -2^{p+1}, where p is the number of independent input pairs. A new pair is added to the output byte;
  - else, the symbolic representation of the output byte is set to -2^p;

– if the number of variable input bytes is 2 or 3, then the symbolic representation of the output byte is set to -1;
– if the number of variable input bytes is 4, then the symbolic representation of the output byte is set to 1.

	Round 1				Round 2				Round 3				Round 4			
State	1	0	0	0	-1	0	0	0	-1	-2	-4	-2	1	1	1	1
	0	0	0	0	-2	0	0	0	-2	-2	-2	-2	1	1	1	1
	0	0	0	0	-2	0	0	0	-2	-1	-2	-4	1	1	1	1
	0	0	0	0	-1	0	0	0	-1	-1	-4	-4	1	1	1	1
Pairs	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_3	S_4	S_1	S_6	S_6	S_7	S_7
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_1	$\emptyset$	$\emptyset$	$\emptyset$	S_2	S_3	S_1	S_1	S_6	S_6	S_7	S_7
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_1	$\emptyset$	$\emptyset$	$\emptyset$	S_2	$\emptyset$	S_1	S_5	S_6	S_6	S_7	S_7
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	S_4	S_5	S_6	S_6	S_7	S_7

$$S_1 = \{1\}; \quad S_2 = \{2\}; \quad S_3 = \{3\}; \quad S_4 = S_1 \cup \{4\} = \{1,4\};$$
$$S_5 = S_1 \cup \{5\} = \{1,5\}; \quad S_6 = S_3 \cup S_5 = \{1,3,5\}; \quad S_7 = S_2 \cup S_4 = \{1,2,4\}$$

Fig. 3: Symbolic processing of an initial state.

Besides updating the symbolic representation of the state, the algorithm keeps a list of key pairs for each byte of the state and carries this list into the next round. The algorithm stops when the symbolic representation of all bytes in a round is 1. It outputs the symbolic representation of the state and the associated key pairs. These can be used to compute the number of rounds required to recover the master key and the number of possible master keys. The pseudocode for the algorithm is given in Appendix C.

Fig. 3 gives a graphical representation of how the algorithm works when only the first byte of the initial state is variable and known, while the other bytes are fixed and unknown. By attacking the result of the `SubByte` transformation applied to the first byte of the state in the first round, the key byte k_0 is recovered. This recovered key byte allows a carry of the attack into the second round where four key bytes $(k'_{16}, k'_{17}, k'_{18}, k'_{19})$ can be recovered by attacking the result of the `SubBytes` transformation. Because the attacker cannot distinguish between k'_{17} and k'_{18}, a new pair $S_1 = \{1\}$ is added to the corresponding state bytes. Then, the attacker targets the third round, where she can recover temporary key bytes for all state bytes. The pair S_1 from previous round affects all bytes of the third and fourth column of the state and thus the corresponding pairs are updated accordingly. In addition, new pairs are added when the attacker can not distinguish between key candidates as shown in Fig. 3. In the fourth round, the attacker is able to recover all round key bytes. Then, having all the round key bytes of the fourth round, she can reverse the AES key schedule to get the master key.

Table 3: Possible attack outcomes for different number of bytes (**Bytes**) controlled by attacker. **Rnds** is the number of rounds that have to be attacked in order to recover the master key. **Prop. (%)** is the proportion of a given evaluation case with respect to all possible input configurations for a fixed number of bytes controlled by attacker.

Bytes	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
min(Rnds)	4	4	4	3	3	3	3	3	3	3	3	3	3	3	3	1
Prop. (%)	100	100	100	14.1	35.2	55.9	72.7	84.7	92.3	96.7	98.9	99.8	100	100	100	100
max(Rnds)	4	4	4	4	4	4	4	4	4	4	4	4	3	3	3	1
Prop. (%)	100	100	100	85.9	64.8	44.1	27.3	15.3	7.7	3.3	1.1	0.2	100	100	100	100
Trade-off	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗

The attacker has to build 2^p possible round keys, where p is the number of independent pairs associated with the state bytes of the last attacked round. For the example in Fig. 3, the number of possible keys is 2^5 because $card(S) = card(S_6 \cup S_7) = card(\{1, 2, 3, 4, 5\}) = 5$. Thus, in addition to the number of rounds to attack, the algorithm for symbolic processing of an initial state gives the number of possible master keys to be recovered by an attacker. Though, the attacker does not have to check all 2^p candidates to see which one is the correct one since she can discard the wrong candidates based on the difference between the correlation coefficients of the first two key candidates as we will show in Section 5.

Using the algorithm for symbolic processing of an initial state we evaluated all possible input combinations. More precisely, we considered all configurations of the initial state when the attacker controls i bytes of the input for $i \in [1, 16]$. When the attacker controls i bytes, there are $\binom{16}{i}$ possible input configurations. This results in $2^{16} - 1$ possible configurations of the initial state in total. Then, we classified these inputs into equivalence classes (evaluation cases) depending on the number of rounds that must be attacked in order to recover the master key. The results are summarized in Table 3. When the attacker controls between four and eleven bytes of the input, a trade-off between the input configuration and the number of rounds to be attacked is possible. When this is the case, the proportion of possible input configurations shows which evaluation case is more likely to appear if the initial state is chosen at random. Thus, when the attacker controls only four or five bytes of the input, it is crucial to carefully choose an input configuration from the limited set of possible input configurations that minimize the number of rounds to be attacked.

Table 4: All evaluation cases with an example of a possible initial state for each evaluation case. **Bytes** gives the number of bytes controlled by attacker; **Rounds** gives the number of rounds that have to be attacked to recover the master key.

Case	Bytes	Rounds	Possible initial state
0	1	4	[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1	2	4	[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
2	3	4	[1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
3	4	3	[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
4	4	4	[1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
5	5	3	[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
6	5	4	[1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
7	6	3	[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
8	6	4	[1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
9	7	3	[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10	7	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
11	8	3	[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
12	8	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0]
13	9	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
14	9	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0]
15	10	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
16	10	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
17	11	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
18	11	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0]
19	12	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
20	12	4	[1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1]
21	13	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]
22	14	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0]
23	15	3	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
24	16	1	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

We give an example of a possible initial state for each of the 25 distinct evaluation cases identified after processing all possible input combinations in Table 4. Any possible input configuration for the AES encryption falls into one of these evaluation cases depending on the number of bytes controlled by attacker and the number of rounds that must be attacked in order to recover the master key.

5 The Attack

The attack we present in this section uses the symbolic representation of the AES state (described in Section 4) in conjunction with CPA attacks to recover individual bytes of the AES round keys. After executing Algorithm 1, the attacker has all round key bytes of round R. Thus, she is able to recover the master key of the cipher by reversing the key schedule.

The algorithm follows the symbolic representation of the state to infer which key bytes must be attacked and how many key candidates it should yield for each attacked key byte. By tracking the pairs associated with the recovered key bytes, the algorithm is able to discard all impossible round keys, thus saving computational resources. Indeed, the algorithm uses an optimal number of CPA attacks to recover the master key.

Initially, the set of known pairs is empty and all possible keys are considered valid. The algorithm keeps track of 2^p possible keys, where p is the total number of independent pairs in the symbolic representation of the state at round R.

The main loop of the algorithm runs through all rounds that must be attacked. At each round, the key bytes corresponding to variable state bytes are attacked to recover one or more temporary key bytes or a round key byte. Depending on the pairs associated with the byte to be attacked, there are three possible cases:

- **No pair**: If the symbolic representation does not have a pair associated with the byte of the state to be used for the attack, then the algorithm will recover a single key byte which is distributed to all possible keys.
- **New pair**: If one of the pairs associated with the byte under attack is not present in the set of known pairs, then the algorithm will recover 2^u possible values for the corresponding key byte, where u is the number of known independent pairs associated with the byte under attack. The number of known pairs determines the number of CPA attacks to be performed. Using a mask based on the existing pairs and a mask for the new pair, the algorithm correctly distributes the recovered key byte values to all possible keys. The new pair is added to the set of known pairs and the two indexes of the state affected by the recovered temporary keys are mapped to this new pair. This mapping prevents the computation of the same temporary keys twice.
- **Known pairs(s):** In the case where the t independent pairs associated with the key byte to be attacked are known but not mapped to the current state byte, the algorithm performs 2^t CPA attacks. Then, it distributes the attack results (the recovered key and the difference between the correlation coefficients of the first two most likely key candidates α) to the corresponding bytes of all possible keys. Afterwards, the possible keys for which the value of α is less than the maximum observed value α_{max} minus a threshold β

Algorithm 1 The attack

```
Require: $state$   ▷ Initial state: 0 – fixed byte, 1 – variable byte
Require: $\lambda = (plaintexts, traces)$   ▷ Recorded in the acquisition phase
 1: $state, pairs$ = PROCESS($state$)   ▷ Symbolic processing (Appendix C, Algorithm 4)
 2: $known\_pairs = \emptyset, \quad mapped\_pairs = \emptyset$
 3: $keys[2^p] = \emptyset, \quad valid\_keys[2^p] = True$   ▷ $p$ is the number of independent pairs
 4: for $r = 1$ to $R$ do   ▷ $R$ is the number of rounds to be attacked
 5:     for $i = 0$ to $15$ do
 6:         if $state[r][i] \neq 0$ then
 7:             if $pairs[r][i] == \emptyset$ then   ▷ No pair
 8:                 $keys[0, \cdots, 2^p - 1][\mathrm{r}][i] = \mathsf{CPA}(\lambda, keys[0], r, i)$
 9:             else if $pairs[r][i] \subseteq known\_pairs$ then   ▷ Known pair(s)
10:                 if $i \notin mapped\_pairs[pairs[r][i]]$ then
11:                     $mask = 0, \quad temp\_keys = \emptyset, \quad \alpha_{max} = -1$
12:                     for $pair \in pairs[r][i]$ do
13:                         $mask = mask \vee 2^{pair-1}$
14:                     end for
15:                     for $j \in [0, 2^p - 1]$ do
16:                         if $valid\_keys[j]$ and $temp\_keys[j \wedge mask] == \emptyset$ then
17:                             $temp\_keys[j \wedge mask], \alpha = \mathsf{CPA}(\lambda, keys[j], r, i)$
18:                             if $\alpha > \alpha_{max}$ then
19:                                 $\alpha_{max} = \alpha$
20:                             end if
21:                         end if
22:                         $valid\_keys[j][r][i] = temp\_keys[j \wedge mask]$
23:                     end for
24:                     for $j \in [0, 2^p - 1]$ do
25:                         if $abs(state[r][i]) == 1$ and $\alpha + \beta < \alpha_{max}$ then
26:                             $valid\_keys[j] = False$
27:                         end if
28:                     end for
29:                 end if
30:             else   ▷ New pair
31:                 $mask = 2^{pairs[r][i] - new\_pair}, \quad k_1 = k_2 = \emptyset$
32:                 for $j \in [0, 2^p - 1]$ do
33:                     if $k_1[j \wedge mask] == \emptyset$ then
34:                         $k_1[j \wedge mask], k_2[j \wedge mask] = \mathsf{CPA}(\lambda, keys[j], r, i)$
35:                     end if
36:                     if $j \wedge 2^{new\_pair-1}$ then
37:                         $keys[j][r][i] = k_1[j \wedge mask], \quad keys[j][r][i'] = k_2[j \wedge mask]$
38:                     else
39:                         $keys[j][r][i'] = k_2[j \wedge mask], \quad keys[j][r][i'] = k_1[j \wedge mask]$
40:                     end if
41:                 end for
42:                 $known\_pairs = known\_pairs \cup new\_pair$
43:                 Add $(i, i')$ to $mapped\_pairs[new\_pair]$
44:             end if
45:         end if
46:     end for
47: end for
48: return $keys[i]$, where $valid\_keys[i] == True$ for $i \in [0, 2^p - 1]$
```

are marked as invalid. In this way, only the combination of keys yielding the highest correlation peak is selected. At this moment, the input pairs are solved in the sense that the algorithm can uniquely assign each of the two temporary keys of a pair to the corresponding state bytes. As a consequence, the algorithm will not further process the possible keys marked as invalid. Thus, this optimization improves the algorithm efficiency by reducing the number of performed CPA attacks.

Finally, the algorithm returns all possible keys which are marked as valid. If the threshold β tends to zero, the algorithm will return only one possible key. When the quality of the side-channel acquisition is good (i.e. high signal-to-noise ratio) and there are enough power traces, the algorithm yields the correct key.

5.1 Optimality

We prove that our algorithm uses the minimum number of CPA attacks possible to recover the master key and thus is optimal. Hence, the lower bounds provided in Table 5 are optimal.

Theorem 1. *Algorithm 1 performs an optimal number of CPA attacks to recover the 16-byte master key of the AES.*

Proof. The only way an attacker can recover the 16-byte master key of the AES is to recover all key bytes of a round r and then to reverse the key schedule. Since the function that derives the round keys of round i from the round keys of round $i - 1$ is bijective, knowledge of all round key bytes of a round r leads to the knowledge of the master key.

Let us assume that Algorithm 1 uses n individual CPA attacks for a given initial state and it is not optimal. Thus, there exists at least an algorithm that is able to recover the master key using only m CPA attacks, with $m < n$. We show next that such an algorithm does not exist. If there exists an algorithm that uses less CPA attacks than Algorithm 1, then this algorithm attacks at least one key byte less. But if it does so, then the attack can not be carried into later rounds any more because the state byte corresponding to the unrecovered key yields unknown and variable state bytes after `MixColumns` transformation. These bytes can not be recovered using a CPA attack and thus the attack fails. As a consequence, there is no algorithm that uses less CPA attacks than Algorithm 1. □

5.2 Choosing the Best Attack Strategy

For up to seven bytes controlled by attacker, our attack algorithm (Algorithm 1) is more efficient than the classic attack algorithm where all possible key bytes are attacked to recover the master key. The gain varies between 15% and 68% of the number of CPA attacks required by the classic attack. When an attacker has control of more than seven input bytes, our algorithm performs the same number of CPA attacks as the classic attack. At the same time, our algorithm gives a unique master key, provided that there are available enough traces with a high signal-to-noise ratio. This is not the case for a classic attack unless an additional mechanism to discard invalid keys, as the one in Algorithm 1, is employed.

Table 5: The number of individual CPA attacks required to recover the master key for different number of bytes (**Bytes**) controlled by attacker; **min(Rnds)/max(Rnds)** and **Bytes** precisely identify the evaluation case. **Classic attack** does not use the optimizations introduced in **Algorithm 1** to discard the invalid keys. **Gain** gives the number of CPA attacks saved by an attacker using **Algorithm 1** over an attacker using **Classic attack**.

	Bytes	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
min(Rnds)	**Classic attack**	150	104	188	80	66	52	46	40	41	42	43	44	45	46	47	16
	Algorithm 1	48	42	48	38	38	38	39	40	41	42	43	44	45	46	47	16
	Gain	102	62	140	42	28	14	7	0	0	0	0	0	0	0	0	0
max(Rnds)	**Classic attack**	150	104	188	110	72	50	51	52	53	54	55	56	45	46	47	16
	Algorithm 1	48	42	48	48	49	50	51	52	53	54	55	56	45	46	47	16
	Gain	102	62	140	62	23	0	0	0	0	0	0	0	0	0	0	0

An attacker willing to reduce the duration of the offline phase of the attack (without increasing the number of rounds that must be attacked) can use the results in Table 5 in corroboration with the data in Table 3 to adjust the attack accordingly. More precisely, if an attacker is able to control up to n bytes of the AES input, she can choose to control m ($m \leq n$) bytes of the input because m variable bytes minimize the number of CPA attacks required to recover the master key. This decision has to be made before performing the side-channel acquisition since it influences the chosen inputs. Another argument in favor of using less variable input bytes is that the attack is much more difficult to detect if the injected packets have fewer variable bytes and mimic the appearance of a normal network traffic. For example, when $n = 12$, an attacker can choose $m = 4, 5,$ or 6 to reduce the complexity of the offline attack from 44 to 38 individual CPA attacks, while still attacking just three rounds. The result is an overall improvement of the attack efficiency by 14% over the classic attack.

An even better decision can be made with the help of experimental results for different configurations of the input from a similar target to the one to be attacked in addition to the results presented so far. For this reason, in the next section we determine experimentally the number of traces required to recover the master key for each evaluation case using EM leakages from an ARM Cortex-M3 processor.

6 Results

For the experimental evaluation, we considered two unprotected implementations of the AES written in ANSI C. The first implementation uses the S-box implementation strategy, while the second one uses the T-table implementation strategy. For each of the 25 evaluation cases we measured up to 2000 EM traces. The acquisition took about 90 minutes for an evaluation case. The samples were split into files corresponding to the AES round number. Then, we mounted the attack presented in Algorithm 1 using an increasing number of traces in the interval $[100, 2000]$ with a step of 100 traces until the guessing entropy converged to zero.

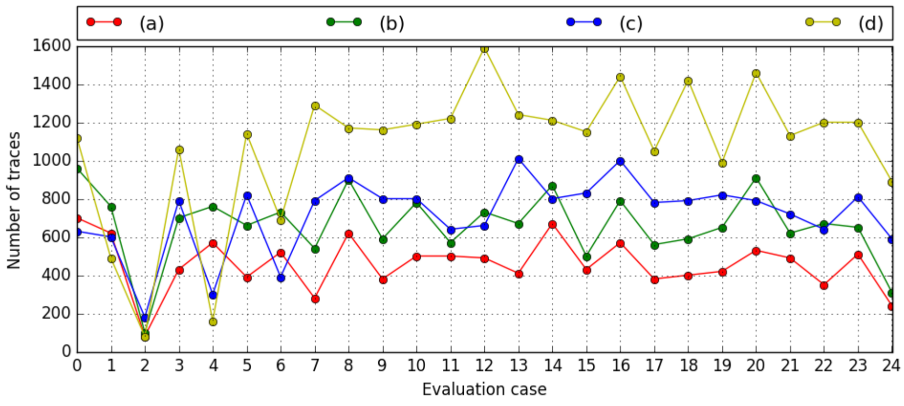


Fig. 4: The number of EM traces required to fully recover the master key. Scenarios: **(a)** S-box implementation, S-box selection function; **(b)** S-box implementation, T-table selection function; **(c)** T-table implementation, T-table selection function; **(d)** T-table implementation, S-box selection function.

For each implementation we considered two selection functions based on the AES S-box and T-table, respectively. The minimum number of traces for which the guessing entropy becomes zero and remains stable is pictorially shown in Fig. 4 for each evaluation case. All attacks recovered the full 16-byte master key with less than 1600 EM traces. In general, the master key was recovered with fewer traces when the selection function perfectly matched the implementation strategy. Though, our results show that full key recovery is possible even when the selection function does not perfectly match the attacked implementation. The attacks on the S-box implementation using the T-table selection function needed 204 more traces on average to recover the master key compared to the attacks on the same implementation using the S-box selection function. Similarly, using the S-box selection function instead of the T-table selection function to attack the implementation based on the T-table strategy required 354 more traces on average. For details on the exact number of traces required to recover the master key for each evaluation case and attack scenario we refer the reader to Appendix E.

Countermeasures. Our experimental results show that side-channel countermeasures such as masking must be employed in order to protect the AES implementations based on lookup tables (S-box and T-table implementation strategies) even in a communication protocol scenario, when the adversary has a limited control of the input. Masking non-linear lookup tables is a challenging task since it adds a considerable penalty on execution time and memory usage [33].

Although not present in many cryptographic libraries due to their limitations (i.e. can not be used in a feedback mode of operation such as CCM), the bitsliced implementations have a lower CPA leakage than implementations using lookup tables [4], but they are still vulnerable to DPA attacks [3].

A lightweight primitive (block cipher or authenticated encryption), particularly one designed for efficient masking, is a good replacement for the AES-CCM when considering side-channel protection.

Other countermeasures, such as a key refreshing mechanism, can support a defense in depth approach. However, any additional countermeasure affects the overall efficiency of an IoT protocol and consequently the most effective ones (i.e. masking) must have priority given the resource constraints.

7 Conclusions

In this paper, we presented an extensive security analysis of AES software implementations against CPA attacks in the context of network protocols. In this scenario the attacker has control of several input bytes, while the remaining input bytes are fixed. To asses the resilience of AES implementations to all possible input combinations, we presented an algorithm for symbolic processing of the cipher state. Then, we classified all possible inputs into 25 independent evaluation cases depending on the number of input bytes controlled by attacker and the number of rounds that must be attacked to recover the master key. Finally, we described an optimal algorithm that recovers the master key by mounting the minimum number of CPA attacks possible. It makes clever decisions based on the set of key pairs that affects the key byte under attack and the correlation coefficient of possible key candidates to discard impossible keys.

We showed that unprotected implementations of the AES based on the S-box and T-table strategies can be broken even when the attacker controls only one input byte of the cipher with less than 1600 electromagnetic traces acquired from a 32-bit ARM Cortex-M3 processor in about one hour. Knowledge of the implementation strategy does not significantly improve the attack outcome, nor does it reduce the attack complexity. Thus, unprotected implementations of the AES should not be used to secure the communication between end devices in network protocols. Care must be taken when using implementations of the AES from popular open-source cryptographic libraries since most of them are not protected against side-channel attacks.

Acknowledgements. We would like to thank the ACNS 2017 reviewers for their valuable feedback and Johann Großschädl for proofreading the final version of this paper. The work of Daniel Dinu is supported by the CORE project ACRYPT (ID C12-15-4009992) funded by the Fonds National de la Recherche (FNR) Luxembourg.

References

1. D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In B. S. K. Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2002.
2. ARM. mbed TLS. `https://tls.mbed.org/`. Accessed: Apr 2017.
3. J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede. DPA, Bitslicing and Masking at 1 GHz. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 599–619. Springer, 2015.
4. A. Biryukov, D. Dinu, and J. Großschädl. Correlation power analysis of lightweight block ciphers: From theory to practice. In M. Manulis, A. Sadeghi, and S. Schneider, editors, *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, volume 9696 of *Lecture Notes in Computer Science*, pages 537–557. Springer, 2016.
5. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.
6. cryptlib. The cryptlib Security Software Development Toolkit. `http://www.cryptlib.com/`. Accessed: Apr 2017.
7. Crypto++. Crypto++: a free C++ class library of cryptographic schemes. `https://www.cryptopp.com/`. Accessed: Apr 2017.
8. J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.
9. M. J. Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. NIST Special Publication 800-38C, 2007.
10. GitHub. libtomcrypt: a fairly comprehensive, modular and portable cryptographic toolkit. `https://github.com/libtom/libtomcrypt`. Accessed: Apr 2017.
11. GitHub. mbed TLS – An open source, portable, easy to use, readable and flexible SSL library. `https://github.com/ARMmbed/mbedtls/blob/development/library/aes.c`. Accessed: Apr 2017.
12. GitHub. OpenSSL – TLS/SSL and crypto library. `https://github.com/openssl/openssl/blob/master/crypto/aes/aes_core.c`. Accessed: Apr 2017.

13. G. Hofemeier and R. Chesebrough. Introduction to Intel AES-NI and Intel Secure Key Instructions. Technical report available at `https://software.intel.com/sites/default/files/m/d/4/1/d/8/Introduction_to_Intel_Secure_Key_Instructions.pdf`. Accessed: Apr 2017.
14. R. Housley. Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP). RFC 4309, December 2005. `https://tools.ietf.org/html/rfc4309`.
15. IEEE. IEEE Standard for Low-Rate Wireless Networks. `https://standards.ieee.org/about/get/802/802.15.html`.
16. J. Jaffe. A first-order DPA attack against AES in counter mode with unknown initial counter. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2007.
17. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
18. libgcrypt. Libgcrypt: a general purpose cryptographic library based on the code from GnuPG. `https://www.gnu.org/software/libgcrypt/`. Accessed: Apr 2017.
19. libsodium. The Sodium crypto library (libsodium). `https://download.libsodium.org/doc/`. Accessed: Apr 2017.
20. M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 549–564. USENIX Association, 2016.
21. LoRa Alliance. Wide Area Networks for IoT. `https://www.lora-alliance.org/`. Accessed: Apr 2017.
22. Nettle. Nettle - a low-level cryptographic library. `http://www.lysator.liu.se/~nisse/nettle/`. Accessed: Apr 2017.
23. NIST. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.
24. C. O'Flynn and Z. Chen. Power analysis attacks against IEEE 802.15.4 nodes. In F. Standaert and E. Oswald, editors, *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*, volume 9689 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 2016.
25. OpenSSL. Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`. Accessed: Apr 2017.
26. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006.
27. Randombit. mbed TLS. `https://botan.randombit.net/`. Accessed: Apr 2017.
28. S. Saab, P. Rohatgi, and C. Hampel. Side-channel protections for cryptographic instruction set extensions. Cryptology ePrint Archive, Report 2016/700, 2016. `http://eprint.iacr.org/2016/700`.
29. N. Sastry and D. Wagner. Security considerations for IEEE 802.15.4 networks. In M. Jakobsson and A. Perrig, editors, *Proceedings of the 2004 ACM Workshop on*

Wireless Security, Philadelphia, PA, USA, October 1, 2004, pages 32–42. ACM, 2004.

30. P. Schwabe and K. Stoffelen. All the AES you need on Cortex-M3 and M4. In *Selected Areas in Cryptography–SAC*, 2016.
31. J. Song, R. Poovendran, J. Lee, and T. Iwata. The AES-CMAC Algorithm. RFC 4493, June 2006. `https://tools.ietf.org/html/rfc4493`.
32. STMicroelectronics. STM32 MCU Nucleo. `http://www.st.com/en/evaluation-tools/stm32-mcu-nucleo.html`. Accessed: Apr 2017.
33. P. K. Vadnala. Time-memory trade-offs for side-channel resistant implementations of block ciphers. In H. Handschuh, editor, *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*, volume 10159 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2017.
34. D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610, September 2003. `https://tools.ietf.org/html/rfc3610`.
35. wolfSSL. wolfCrypt Embedded Crypto Engine. `https://www.wolfssl.com/wolfSSL/Products-wolfcrypt.html`. Accessed: Apr 2017.

A Structure of the AES Encryption and Key Schedule

The structure of the AES encryption is given in Algorithm 2.

Algorithm 2 AES encryption

```
Require: $state$, $round\_keys$
 1: AddRoundKey($state$, $round\_keys[0]$)
 2: for $r = 1$ to $R - 1$ do                    ▷ $R$ is the total number of rounds
 3:     SubBytes($state$)
 4:     ShiftRows($state$)
 5:     MixColumns($state$)
 6:     AddRoundKey($state$, $round\_keys[r]$)
 7: end for
 8: SubBytes($state$)
 9: ShiftRows($state$)
10: AddRoundKey($state$, $round\_keys[R]$)
11: return $state$
```

Algorithm 3 describes the AES key schedule at the byte level when using a 16-byte master key.

Algorithm 3 AES key schedule for a 16-byte master key

```
Require: $key$
 1: $rk[0] = key$
 2: for $i = 1$ to $R$ do                    ▷ $R$ is the total number of rounds
 3:     $rk[i][0] = rk[i-1][0] \oplus$ SubBytes($rk[i-1][13]$) $\oplus$ Rcon[$i-1$]
 4:     $rk[i][1] = rk[i-1][1] \oplus$ SubBytes($rk[i-1][14]$)
 5:     $rk[i][2] = rk[i-1][2] \oplus$ SubBytes($rk[i-1][15]$)
 6:     $rk[i][3] = rk[i-1][3] \oplus$ SubBytes($rk[i-1][12]$)
 7:     for $j = 4$ to 15 do
 8:         $rk[i][j] = rk[i-1][j] \oplus rk[i][j-4]$
 9:     end for
10: end for
11: return $rk$
```

B Measurement Setup

For all experimental results reported in this paper we used a STM32 Nucleo [32] board from STMicroelectronics. It has a 32-bit ARM Cortex-M3 processor clocked at 8 MHz, 512 KB of Flash, 80 KB of RAM and 16 KB of EEPROM. The measurement of the electromagnetic emissions was performed from a spot above the chip using a Langer RF-K 7-4 H-field probe as shown in Fig. 5. The signal was amplified by 30dB and then sampled at 500 MS/s using a Teledine LeCroy WaveRunner 8254M-MS oscilloscope. We did not use any noise reduction technique. The board was powered through an USB cable, which was also used to control the device under test (DUT).

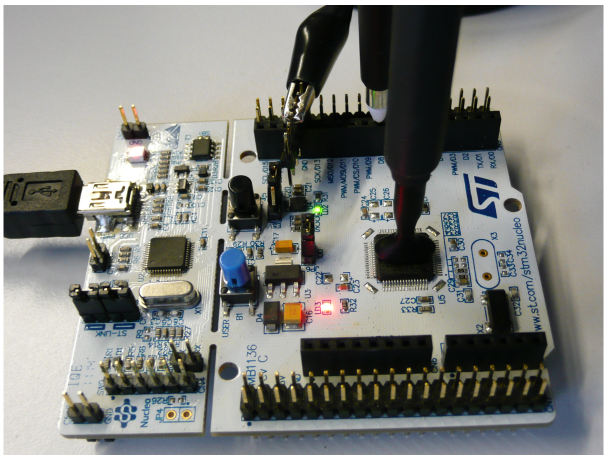


Fig. 5: The device under test (DUT).

C Symbolic Processing of an Initial State

Algorithm 4 describes the symbolic processing of an initial state. It returns the processed state and the associated set of pairs. Using this output, an attacker knows what key bytes have to be attacked in each round of the AES, the number of rounds to be attacked, and the maximum number of possible master keys.

Algorithm 4 Symbolic processing of an initial state

```
Require: $state$                               ▷ Initial state: 0 – fixed byte, 1 – variable byte
1: function ProceessColumn($r, pairs, i_0, i_1, i_2, i_3, o_0, o_1, o_2, o_3$)
2:     Compute the number of variable inputs for $i_0, i_1, i_2, i_3$: $var\_in$
3:     Update $pairs$
4:     if $var\_in == 0$ then                                          ▷ No key bytes recovered
5:         $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = 0$
6:     else if $var\_in == 1$ then                  ▷ 4 temporary key bytes recovered; new pair
7:         Compute the number of independent pairs: $p$
8:         $pairs = pairs \cup \{new\_pair\}$
9:         $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = -2^p$
10:        if $state[r][o_i] == state[r][o_j] == (\{01\} \bullet state[r-1][i_t]) \oplus k'$ then
11:            $state[r][o_i] = state[r][o_j] = -2^{p+1}$
12:        end if
13:    else if $var\_in \in \{2, 3\}$ then                       ▷ 4 temporary key bytes recovered
14:        $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = -1$
15:    else if $var\_in == 4$ then                                   ▷ All 4 key bytes recovered
16:        $state[r][o_0] = state[r][o_1] = state[r][o_2] = state[r][o_3] = 1$
17:    end if
18: end function

19: function ProcessRound($r, pairs$)
20:     ProcessColumn($r, pairs, 0, 5, 10, 15, 0, 1, 2, 3$)
21:     ProcessColumn($r, pairs, 4, 9, 14, 3, 4, 5, 6, 7$)
22:     ProcessColumn($r, pairs, 8, 13, 2, 7, 8, 9, 10, 11$)
23:     ProcessColumn($r, pairs, 12, 1, 6, 11, 12, 13, 14, 15$)
24: end function

25: function RoundKeyRecovered($r$)
26:     for $i = 0$ to 15 do
27:         if $0 \geq state[r][i]$ then return False
28:         end if
29:     end for
30:     return True
31: end function

32: function Process($state$)
33:     $pairs = \emptyset$
34:     for $r = 1$ to $R$ do                              ▷ $R$ is the total number of rounds
35:         if RoundKeyRecovered($r - 1$) then return $state, pairs$
36:         end if
37:         ProcessRound($r, pairs$)
38:     end for
39: end function

40: return Process($state$)
```

D Results for Simulated Traces

We averaged the guessing entropy of 100 experiments on simulated traces for each of the 25 evaluation cases. Then we selected the minimum number of traces for which the guessing entropy was zero and remained stable. The results are shown in Fig. 6.

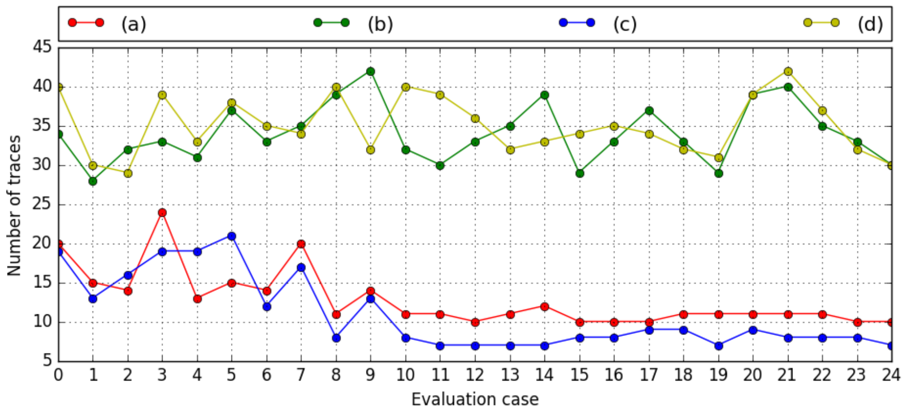


Fig. 6: The number of simulated traces required to fully recover the master key. Scenarios: **(a)** S-box implementation, S-box selection function; **(b)** S-box implementation, T-table selection function; **(c)** T-table implementation, T-table selection function; **(d)** T-table implementation, S-box selection function.

Comparing the results for the simulated traces (Fig. 6) with the results for the EM traces acquired from the Cortex-M3 processor (Fig. 4), we can notice the following differences:

– In general, for simulated traces the attacks against the T-table implementation using the T-table selection function (c) required a similar but slightly smaller number of traces than the attacks on the S-box implementation using the S-box selection function (a). In contrast, the leakage estimation of the two selection functions indicates that the S-box selection function leaks a little bit more than the T-table selection function. But when the leakages of the two selection functions were quantified, they were clearly isolated from the leakages of other operations. As a consequence, the intermediate results of similar neighboring operations have a greater influence on the correlation of the S-box leakage than on the correlation of the T-table leakage. This can be explained by the fact that there are 19 possible Hamming weight values for the T-table output and only 9 possible Hamming weight values for the S-box output. Thus, it is easier to distinguish a T-table output from the neighboring T-table outputs than an S-box output from the neighboring S-box outputs. On the other hand, for the EM traces, the attacks on the S-box implementation using the S-box selection function (a) required less traces than the attacks on the T-table implementation using the T-table selection function (c). For the EM traces, the attack results are consistent with the leakages of the selection functions as quantified by the correlation coefficient difference.

- The attacks that used the non-matching selection functions (b, d) required a similar number of simulated traces. Contrariwise, the attacks on the T-table implementation using the S-box selection function (d) required more EM traces than the attacks on the S-box implementation using the T-table selection function (b).
- In the case of simulated traces attacked with the matching selection functions (a, c), the number of traces necessary to fully recover the master key when the attacker controlled less than six input bytes was greater than when the attacker controlled more than six input bytes. On the contrary, the number of traces necessary to fully recover the master key for the EM leakage was minimal when the attacker controlled exactly three input bytes.

For details on the exact number of traces required to recover the master key for each evaluation case and attack scenario we refer the reader to Appendix E.

E Detailed Results

We give the number of traces required to fully recover the AES master key using simulated and EM traces for all evaluation cases in all considered attack scenarios in Table 6.

Table 6: The number of traces required to fully recover the master key for each evaluation case (**Case**). **Bytes** gives the number of bytes controlled by attacker; **Rounds** gives the number of rounds that have to be attacked to recover the master key. Scenarios: **(a)** S-box implementation, S-box selection function; **(b)** S-box implementation, T-table selection function; **(c)** T-table implementation, T-table selection function; **(d)** T-table implementation, S-box selection function.

Case	**Bytes**	**Rounds**	**EM leakage**				**Simulated leakage**			
			(a)	**(b)**	**(c)**	**(d)**	**(a)**	**(b)**	**(c)**	**(d)**
0	1	4	700	960	630	1120	20	34	19	40
1	2	4	620	760	600	490	15	28	13	30
2	3	4	80	100	180	80	14	32	16	29
3	4	3	430	700	790	1060	24	33	19	39
4	4	4	570	760	300	160	13	31	19	33
5	5	3	390	660	820	1140	15	37	21	38
6	5	4	520	730	390	690	14	33	12	35
7	6	3	280	540	790	1290	20	35	17	34
8	6	4	620	900	910	1170	11	39	8	40
9	7	3	280	590	800	1160	14	42	13	32
10	7	4	500	780	800	1190	11	32	8	40
11	8	3	500	570	640	1220	11	30	7	39
12	8	4	490	730	660	1590	10	33	7	36
13	9	3	410	670	1010	1240	11	35	7	32
14	9	4	670	870	800	1210	12	39	7	33
15	10	3	430	500	830	1150	10	29	8	34
16	10	4	570	790	1000	1440	10	33	8	35
17	11	3	380	560	780	1050	10	37	9	34
18	11	4	400	590	790	1420	11	33	9	32
19	12	3	420	650	820	990	11	29	7	31
20	12	4	530	910	790	1460	11	39	9	39
21	13	3	490	620	720	1130	11	40	8	42
22	14	3	350	670	640	1200	11	35	8	37
23	15	3	510	650	810	1200	10	33	8	32
24	16	1	240	310	590	890	10	30	7	30
	Average		459	663	716	1070	13	34	11	35