Recovering the CTR_DRBG state in 256 traces

Lauren De Meyer

KU Leuven, imec - COSIC
lauren.demeyer@esat.kuleuven.be

Abstract. The NIST CTR_DRBG specification prescribes a maximum size on each random number request, limiting the number of encryptions in CTR mode with the same key to 4 096. Jaffe's attack on AES in CTR mode without knowledge of the nonce from CHES 2007 requires 2^{16} traces, which is safely above this recommendation. In this work, we exhibit an attack that requires only 256 traces, which is well within the NIST limits. We use simulated traces to investigate the success probability as a function of the signal-to-noise ratio. We also demonstrate its success in practice by attacking an AES-CTR implementation on a Cortex-M4 among others and recovering both the key and nonce. Our traces and code are made openly available for reproducibility.

Keywords: DPA · SCA · CPA · AES · CTR · PRNG · NIST · DRBG · DDLA

1 Introduction

Cryptographic implementations in embedded devices are vulnerable to side-channel attacks (SCA) such as differential power analysis (DPA), which was first introduced by Kocher *et al.* in 1999 [KJJ99]. In the following years, many variations of this attack have been proposed, such as correlation power analysis (CPA) by Brier *et al.* [BCO04], mutual information analysis (MIA) by Gierlichs *et al.* [GBTP08] and very recently, differential deep learning analysis (DDLA) by Timon [Tim19].

The success of DPA and its variations lies in the ability to divide-and-conquer, because the power consumption at some instants depends on a (constant) small part of the secret combined with variable known data (*e.g.* plaintext bytes). In most cases, side-channel attacks are performed under the assumption that the adversary knows the plaintext and/or ciphertext, which allows him to hypothesize on and recover chunks of the secret key.

In some scenarios, this assumption does not hold. Consider for example a pseudo-random number generator (PRNG) that is used for key generation or for the supply of fresh randomness to masked implementations (to protect against SCA). In such cases, neither the plaintext (*i.e.* the state of the PRNG) nor the ciphertext (*i.e.* the output of the PRNG) are considered public. The adversary is then assumed to only have knowledge of the power consumption or electromagnetic radiation emanating from the device.

At CHES 2007, Jaffe [Jaf07] presented an attack of AES in Counter mode (AES-CTR) [Dwo01] in this adversary model. He showed that the sequential nature of the counter mode enables one to attack AES-CTR with only knowledge of the power traces and without knowledge of the initial counter (the nonce). Another line of works that consider the same adversary model is that of blind side-channel attacks, originally by Linge *et al.* [LDL14] and recently improved by Clavier *et al.* [CR17]. In these works, the joint distribution of leakage points is exploited to extract keys without knowledge of the plaintext or ciphertext.

In this work, we focus on the case of PRNGs. The NIST recommendations for random number generation include one type of PRNG which is based on a cipher in CTR mode,

IACR Transactions on Cryptographic Hardware and Embedded Systems ISSN 2569-2925, Vol. 2020, No. 1, pp. 37–65
DOI:10.13154/tches.v2020.i1.37-65

denoted CTR_DRBG [BK15]. AES being an important standardized cipher, many PRNGs naturally use AES-CTR at their core, which means they are vulnerable to Jaffe's attack. However, NIST recommends to limit the size of randomness requests to the CTR_DRBG to 2^{19} bits. Generating such a request thus takes at most 4 096 AES encryptions in CTR mode. The NIST CTR_DRBG also calls an Update function, which changes the PRNG state (nonce and key) between every request. Since Jaffe's attack requires 2^{16} encryption traces, it actually does not pose a threat to the NIST CTR_DRBG. In his conclusion [Jaf07], he does allude to the possibility of using only 2^8 traces.

1.1 Contribution

In this work, we demonstrate an adaptation of Jaffe's attack, which requires only 256 power measurements. We explain the methodology and investigate the success probability of the attack as a function of the signal-to-noise ratio. Interestingly, our attack's success depends on the nonce it is trying to recover and we show that in some cases, using *less* traces actually improves the success probability.

We demonstrate the feasibility of the attack on multiple real devices, essentially showing that the NIST recommendation for the CTR_DRBG allows for too large requests. We also explore blind SCA [CR17] as an alternative attack methodology and demonstrate the recently introduced DDLA [Tim19] in a variation of the attack for misaligned traces.

In the context of masked implementations against SCA, PRNGs are usually required to provide a constant stream of fresh randomness during the computation. Having that randomness compromised would nullify the protection offered by the masking countermeasure. To this day, very little research is publicly available on specific constructions for this PRNG. The question of whether this PRNG should be protected against side-channel analysis itself is largely avoided. We use our attack as a starting point for the discussion on how to protect PRNGs against adversaries who only have access to side-channel information and not the plaintexts/ciphertexts.

2 Preliminaries

In Section 2.1, we give a brief overview of AES and introduce our notation for the rest of the paper. Section 2.2 describes the NIST recommendations for the CTR_DRBG.

2.1 AES

The Advanced Encryption Standard (AES) is a 128-bit block cipher based on a substitution-permutation network. The master key can be 128, 192 or 256 bits long and the corresponding number of rounds is respectively 10, 12 or 14. Each round i (except the last round) consists of 4 transformations (AddRoundKey, SubBytes, ShiftRows and MixColumns), which we explain briefly below. The 128-bit state is considered as a matrix of 4 by 4 bytes (see Figure 1). Each round also receives a 128-bit round key K_i, which is derived from the master key using the key schedule. The details of the key schedule are not relevant here.

$$X_i = \begin{array}{|c|c|c|c|} \hline X_{i,0} & X_{i,4} & X_{i,8} & X_{i,12} \\ \hline X_{i,1} & X_{i,5} & X_{i,9} & X_{i,13} \\ \hline X_{i,2} & X_{i,6} & X_{i,10} & X_{i,14} \\ \hline X_{i,3} & X_{i,7} & X_{i,11} & X_{i,15} \\ \hline \end{array}$$

Figure 1: AES state

AddRoundKey is a linear transformation, which performs a 128-bit exclusive or ($\oplus$) between the state X_i and the round key K_i:

$$Y_i = X_i \oplus K_i$$

SubBytes is the only nonlinear transformation in the round function. It takes each of the 16 bytes of the state and substitutes it for another:

$$Z_{i,j} = S(Y_{i,j}) = S(X_{i,j} \oplus K_{i,j}) \qquad j = 0 \dots 15$$

A typical DPA attack targets the output of this function and exploits the fact that $X_{1,j}$ (a plaintext byte) is known and variable and $K_{1,j}$ (a master key byte) is unkown and fixed over the acquired traces.

ShiftRows is simply a permutation of the state bytes, obtained by rotating row j of the state matrix by j bytes to the left (see Figure 2).

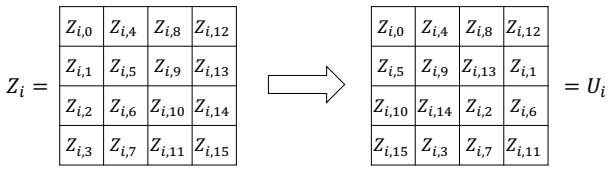


Figure 2: AES ShiftRows

MixColumns is a linear transformation of the AES state, by multiplying each column of the state with a matrix M in $\mathbb{F}_{2^8}$. This is the last transformation of each round, except the last round, where this step is skipped.

$$X_{i+1,[4j \dots 4j+3]} = M \times U_{i,[4j \dots 4j+3]} \qquad j = 0 \dots 3$$
$$\text{with } M = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

2.2 CTR-DRBG

The NIST publication SP 800-90A [BK15] describes recommendations for random number generation using Deterministic Random Bit Generators (DRBG). One of these is based on block ciphers in CTR mode and is therefore referred to as CTR_DRBG. For a detailed description of the operation of the CTR_DRBG, we refer to [BK15]. A simplified pseudocode of the functions relevant for this work is given in Algorithms 1 and 2.

Random Number Generation. In the context of this paper, it is important to know that the internal state of the CTR_DRBG contains a Key and a value V, as shown in Figure 3. The value of V at the beginning of a randomness request is what we refer to as the *nonce* N. The value V is incremented by a counter after every use of the block cipher AES, as in CTR mode. This is shown in Figure 3 on the right and in Algorithm 2. While the block cipher performs in CTR mode, the output blocks are concatenated until the requested output length is obtained.

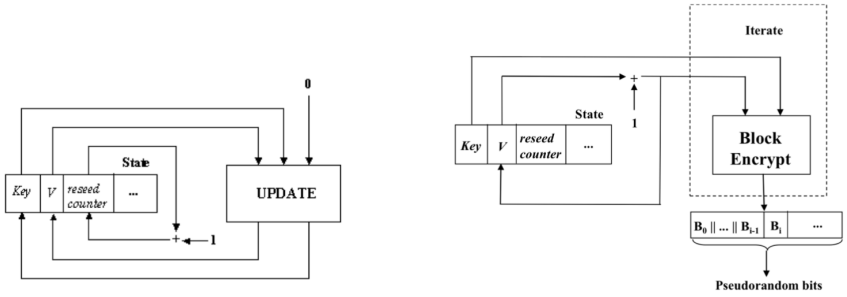


Figure 3: Operation of the NIST CTR_DRBG _Update function (left) and random bit stream generation (right) [BK15]

Algorithm 1 NIST CTR_DRBG _Update (simplified)

Input: (Key,V), seed length
Output: (Key,V)
1: Init $x = 0$
2: **while** length(x) < seed length **do**
3: $V = V + 1$
4: $x = x|AES_{\text{Key}}(V)$
5: **end while**
6: (Key,V) $\leftarrow x$

Updating the State. At the end the random bit generation in Algorithm 2, a new key and value V are generated by the CTR_DRBG _Update function, which is shown in Algorithm 1. This essentially means that performing a DPA attack across various requests is not possible, because the secret key changes. Any DPA attack would have to be performed during a single request to the DBRG (Algorithm 2 lines 2-5). However, the maximum number of bits per request is limited [BK15, Table 3]. If the counter field occupies at least 13 bits of the block, then the maximum number of bits per randomness request is 2^{19}. In the case of AES, which has a block length of 16 bytes, this is equivalent to $2^{12} = 4\,096$ encryptions. If the counter field length ("ctr_len") is smaller, the number of performed encryptions in CTR mode is $2^{\text{ctr\_len}} - 4$. Not specified here in these algorithms is the reseed counter, which makes sure that the PRNG is reseeded when the number of requests succeeds a threshold. According to the NIST specifications, this threshold must be at most 2^{49}.

Algorithm 2 NIST CTR_DRBG _Generate (simplified)

Input: (Key,V), requested # bits
Output: x
1: Init $x = 0$
2: **while** length(x) < requested # bits **do**
3: $V = V + 1$
4: $x = x|AES_{\text{Key}}(V)$
5: **end while**
6: Truncate x to requested # bits
7: (Key,V) $\leftarrow$ CTR_DRBG _Update(Key,V)

Forward/Backward Secrecy. The concepts of forward and backward secrecy evaluate the security of PRNGs when their state is compromised (*i.e.* known by an adversary). The CTR_DRBG provides backward secrecy because recovering the state (key and nonce) during one request does not allow an adversary to compute the previous states. The explanation for this is simply that the current state is the result of an AES-CTR computation with the previous (unkown) key (see Algorithm 1). On the other hand, as long as the DRBG is not reseeded with a fresh seed, it does not provide forward secrecy, since the knowledge of the current state allows one to perfectly predict the following states.

3 The Attack

In this attack, as in [Jaf07], we perform DPA on four rounds of AES-CTR. In the first rounds, we assume a large part of the state is constant and we recover information about a few variable bytes. By propagating them through the ShiftRows and MixColumns transformations, we obtain enough information to perform DPA in the next round, until finally, we can recover the entire round key in round four. In this work, we choose CPA as our attack methodology.

Simulated traces. For the remainder of this section, we apply the steps of the attack to simulated traces and explore the success rate as a function of the signal-to-noise ratio (SNR). We will apply the attack to traces from real devices in Section 4. To generate the simulated traces, we perform AES-CTR and after each round transformation, we collect the Hamming weights of the 16 bytes of the state and add them to the trace. Each time sample in a simulated trace thus corresponds to the Hamming weight of one state byte in one round. We then add Gaussian noise to the trace with some standard deviation σ. The variance σ^2 is calculated as the variance of the collected Hamming weights divided by the desired SNR. For example, the relationship between the actual Hamming weight and some simulated leakages is shown in Figure 4. For each experiment, we add new noise to the original Hamming weights and we measure the success as the proportion of correct bytes recovered. We repeat each experiment ten times for each SNR.

Setup. The input to the first round is constructed by the addition of a counter T with an unknown nonce N: $X_1 = N + T \mod 2^{128}$. We assume for simplicity that the counter starts at the least significant byte of the state. It is trivial to adapt the attack if this is not the case. We thus assume that $X_{1,15} = N_{15} + T \mod 256$, with N_{15} constant and unknown and T the counter starting from 0. Further, since we will only use 256 traces, we can consider the 14 most significant bytes completely constant: $X_{1,j} = N_j$ for $j < 14$.

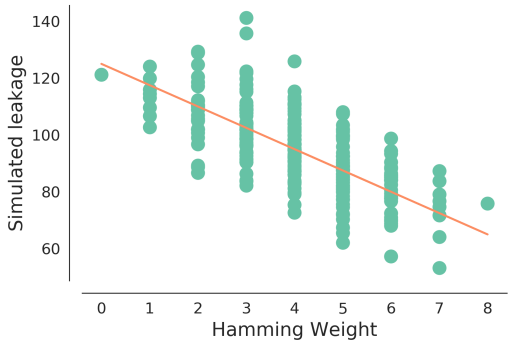


Figure 4: Simulated leakages vs. actual Hamming weights for SNR=1.0

Byte 14 is a special case, since it is not constant, but will only assume two values: N_{14} and $(N_{14}+1) \mod 256$. We visualize this in Figure 5, where white squares signify fixed values, black squares are varying continuously and the byte in the grey square toggles at most once in the set of traces.

$X_1 =$

$X_{1,0}$	$X_{1,4}$	$X_{1,8}$	$X_{1,12}$
$X_{1,1}$	$X_{1,5}$	$X_{1,9}$	$X_{1,13}$
$X_{1,2}$	$X_{1,6}$	$X_{1,10}$	$X_{1,14}$
$X_{1,3}$	$X_{1,7}$	$X_{1,11}$	$X_{1,15}$

Figure 5: Input to the first round of AES

3.1 Round 1: one byte

The attack on the least significant byte corresponds exactly to that described in [Jaf07]. This is the most complex step in our attack, as it requires hypothesizing on 15 unknown bits (*i.e.* complexity 2^{15}). We target the output of SubBytes:

$$Z_{1,15} = S(K_{1,15} \oplus X_{1,15}) = S\Big(K_{1,15} \oplus \big((N_{15}+T) \mod 256\big)\Big)$$

As in [Jaf07], let $N_{15} = N_{15,hi}|N_{15,lo}$ and $K_{1,15} = K_{1,15,hi}|K_{1,15,lo}$ where hi denotes the most significant bit and lo the other 7 bits and let $b = N_{15,hi} \oplus K_{1,15,hi}$. Then we can write $Z_{1,15}$ as

$$Z_{1,15} = S\Big((b \ll 7) \oplus K_{1,15,lo} \oplus \big((N_{15,lo}+T) \mod 256\big)\Big) \text{[Jaf07]}$$

We then perform CPA, where we hypothesize on the 15 bits ($b, K_{1,15,lo}$ and $N_{15,lo}$) and compute the correlation between our $Z_{1,15}$ and the traces. The winning hypothesis (with the largest absolute correlation) does not tell us the most significant bits of $K_{1,15}$ and N_{15}, but this is of no importance for the remainder of the attack. With these 15 bits, we know $Z_{1,15}$ completely.

Figure 6 shows the success rate of this step, which is 1.0 for reasonably low SNR levels. Below the threshold of SNR=0.2, the success rate decreases dramatically and becomes 0.0 as of SNR=0.01.

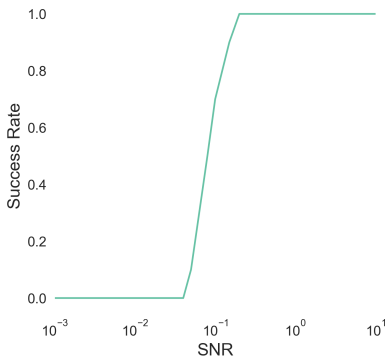


Figure 6: Success Rate of Step 1 with 256 traces as function of the SNR.

3.2 Round 2: four bytes

Figure 7 depicts the AES state after the ShiftRows and MixColumns operations in terms of variability. We will use the continuously changing byte $Z_{1,15}$ to recover the first column of the state after SubBytes.

$Z_{1,0}$	$Z_{1,4}$	$Z_{1,8}$	$Z_{1,12}$
$Z_{1,5}$	$Z_{1,9}$	$Z_{1,13}$	$Z_{1,1}$
$Z_{1,10}$	$Z_{1,14}$	$Z_{1,2}$	$Z_{1,6}$
$Z_{1,15}$	$Z_{1,3}$	$Z_{1,7}$	$Z_{1,11}$

$X_{2,0}$	$X_{2,4}$	$X_{2,8}$	$X_{2,12}$
$X_{2,1}$	$X_{2,5}$	$X_{2,9}$	$X_{2,13}$
$X_{2,2}$	$X_{2,6}$	$X_{2,10}$	$X_{2,14}$
$X_{2,3}$	$X_{2,7}$	$X_{2,11}$	$X_{2,15}$

Figure 7: AES state after the first Shiftrows (left) and MixColumns (right) transformations.

Consider for example the SubBytes output $Z_{2,0}$:

$$\begin{aligned} Z_{2,0} &= S(Y_{2,0}) = S(K_{2,0} \oplus X_{2,0}) \\ &= S(\underbrace{K_{2,0} \oplus 2Z_{1,0} \oplus 3Z_{1,5} \oplus 1Z_{1,10}}_{\text{constant \& unknown}} \oplus \underbrace{1Z_{1,15}}_{\text{variable \& known}}) \end{aligned} \tag{1}$$

By treating $K_{2,0} \oplus 2Z_{1,0} \oplus 3Z_{1,5} \oplus Z_{1,10}$ as one unkown 8-bit constant $C_{2,0}$, we can recover this constant using CPA with only 256 hypotheses and thus recover $Z_{2,0}$. The same is true for the other three bytes in the first column:

$$\begin{aligned} Z_{2,0} &= S(C_{2,0} \oplus 1Z_{1,15}) \\ Z_{2,1} &= S(C_{2,1} \oplus 1Z_{1,15}) \\ Z_{2,2} &= S(C_{2,2} \oplus 3Z_{1,15}) \\ Z_{2,3} &= S(C_{2,3} \oplus 2Z_{1,15}) \end{aligned} \tag{2}$$

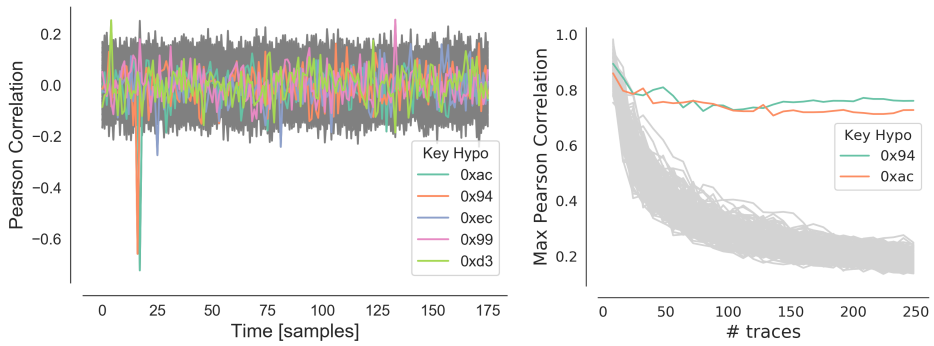


Figure 8: Pearson Correlation coefficients in Step 2 with SNR=1.0, with 256 traces as a function of the time samples (left) and their maximum as a function of the number of traces (right).

We note that performing CPA for $Z_{2,0}$ and $Z_{2,1}$ is identical, since in both cases the S-box input is the sum of $1Z_{1,15}$ with a constant. Indeed, the example in Figure 8 shows that there is not one but there are two prevailing hypotheses: `0xAC` and `0x94`. Since each byte corresponds to only one time sample in the simulated traces, the correlation peaks are very close to each other in Figure 8. The separation is more clear in real power traces. If the S-box evaluations are not randomly shuffled, it is trivial to decide which constant belongs to which state byte. In this case, $C_{2,0}$ = `0x95` and $C_{2,1}$ = `0xAC`.

Figure 9 shows the success rate of recovering all four bytes of the first column. Again, the threshold for reaching 100% success lies at SNR=0.2. The cutoff is still quite steep, with 0 success for SNR=0.001 and below.

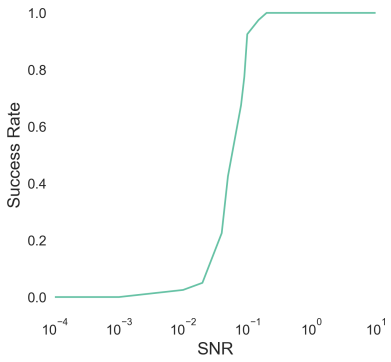


Figure 9: Success Rate of the second step of the attack with 256 traces.

3.3 Round 3: sixteen bytes

The known bytes $Z_{2,0}$ to $Z_{2,3}$ are spread to all columns of the state by the ShiftRows transformation as shown in Figure 10. The subsequent MixColumns operations will affect the entire state. As shown by the grey squares in Figure 10, each column now has an additional byte that is non-constant. Because we only have 256 traces and didn't follow the approach from [Jaf07], the grey bytes are also unknown. However, keep in mind that the grey bytes only assume two distinct values throughout all the traces. The number of traces for each depends on the carry of the addition $X_{1,15} = (N_{15} + T) \mod 256$, which makes $X_{1,14}$ toggle from N_{14} to $(N_{14} + 1) \mod 256$.

$Z_{2,0}$	$Z_{2,4}$	$Z_{2,8}$	$Z_{2,12}$
$Z_{2,5}$	$Z_{2,9}$	$Z_{2,13}$	$Z_{2,1}$
$Z_{2,10}$	$Z_{2,14}$	$Z_{2,2}$	$Z_{2,6}$
$Z_{2,15}$	$Z_{2,3}$	$Z_{2,7}$	$Z_{2,11}$

Figure 10: AES state after the second ShiftRows transformation.

Best Case. Assume for simplicity that the least significant byte of the nonce N_{15} is `0x00`. In that case, $X_{1,14} = N_{14}$ never toggles and all grey squares in Figure 10 are constant, just like the white squares. This means that in each column, we can apply the same method as we did in round 2. Each byte of the SubBytes output can be written as (2):

$$Z_{3,j} = S(C_{3,j} \oplus f_j Z_{2,k_j}) \tag{3}$$

where the factors f_j are easily derived from MixColumns matrix M and k_j refers to the known byte (the black squares) in each column (see Appendix A). As in round 2, each column again has two bytes for which the hypotheses are identical (when $f_j = 1$) and the correct constants can be derived by comparing the time samples where the maximum correlation occurs.

Now, assume that the nonce is `0xFF` and $X_{1,14}$ toggles immediately, leading to the grey squares in Figure 10 being identical in all but one of the traces. When performing the same CPA, we now recover different constants $C'_{3,j}$, corresponding to when $X_{1,14} = N_{14} + 1 \mod 256$.

Average Case. In all other cases, the constants in the computation will be C_3 for the first portion of traces and C'_3 for the second portion, after $X_{1,14}$ has toggled. Interestingly, the same approach as before, with 256 traces, still works. The winning hypotheses are those constants that occur most often in the set of traces. The traces that correspond to the other (not-winning) constants act as noise. The attack is successfull if the 16 recovered bytes are either C_3 or C'_3, but not a mix of both. Clearly, this depends on the least significant byte of the nonce (N_{15}), since this byte decides when $X_{1,14}$ toggles from N_{14} to $N_{14} + 1$ and the constants from C_3 to C'_3. This is demonstrated in Figure 13, where we show the success rate for various values of N_{15}.

Worst Case. The worst case scenario is when the toggle occurs approximately halfway, *i.e.* when $N_{15} \approx$ `0x80`. In that case, the constants $C_{3,j}$ and $C'_{3,j}$ are in a close race (see Figure 11, left). This results in recovering some bytes from C_3 and some from C'_3, which is a problem for the next and last stage of the attack. This is clearly reflected in the results in Figure 13, since the success rate only converges to approximately 0.6 for nonce `0x80`. Figure 13 also shows the success rate of the attack with $N_{15} =$ `0x80` when we use only half of the traces, indicated by `0x80`*. This is a rare and interesting case, where using *less* traces actually improves the performance of the attack, though, not surprising since we know that the traces we are removing act as noise.

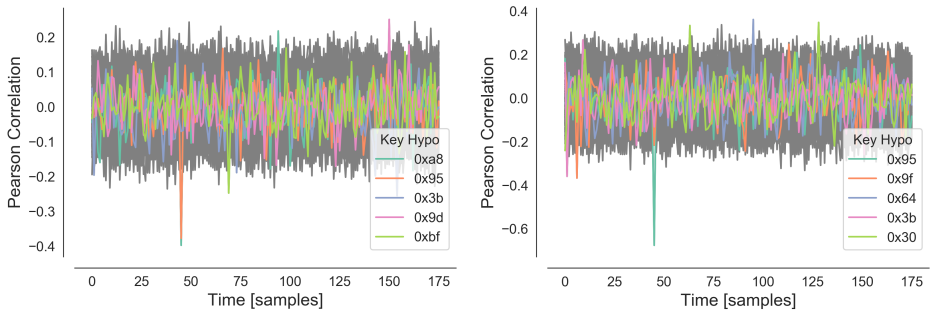


Figure 11: Pearson Correlation coefficients in Step 3 with $N_{15} =$ `0x80` and SNR=1.0 with 256 traces (left) and 128 traces (right).

In Figure 12, we depict the maximum correlation coefficient for each hypothesis as a function of the amount of traces used for the best and worst case. It demonstrates again very clearly that with nonce $N_{15} =$ `0x80`, using more than 128 traces only deteriorates the success of key recovery.

The attacker only knows the least significant 7 bits of the nonce, so is unable to distinguish `0x80` from `0x00`. However, seeing a close race as in Figure 11, left is a good clue, especially if performing the CPA again with only half the traces results in a clear winner (Figure 11, right).

Also in other cases, the knowledge of the 7 least significant nonce bits can be used to calculate exactly how many traces to remove (either at the beginning or the end of the acquired set) to have a pure subset of traces using only one constant. There are two possible sets of traces, depending on whether the most significant bit of N_{15} is 0 or 1. We can try out both possibilities and detect as in Figure 11, which option gives the best results. We will demonstrate this in in the examples in Section 4.

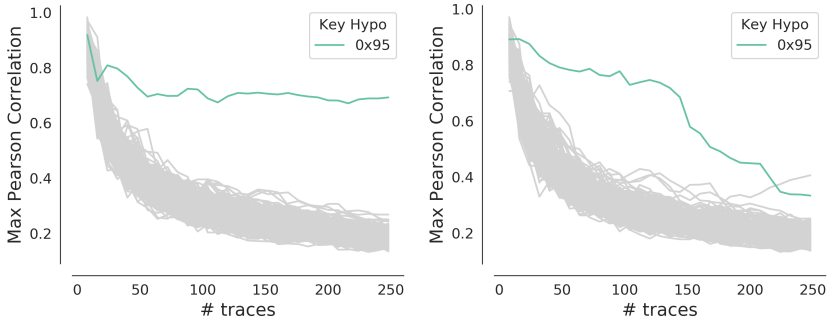


Figure 12: Maximum Correlation coefficients as a function of the number of traces in Step 3 with SNR = 1.0 and N_{15} = `0x00` (left) or N_{15} = `0x80` (right).

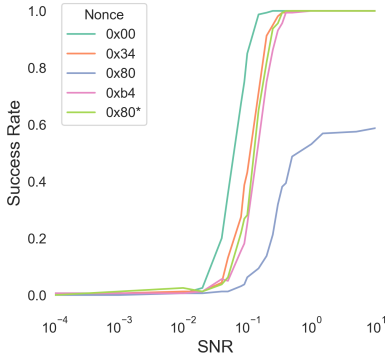


Figure 13: Success Rates of the third step of the attack for various nonces with 256 traces (except `0x80`* with 128 traces).

3.4 Round 4: recovering the round key

Whether the previous step recovered constants C_3 or C_3', we now know exactly the state Z_3 in *most* of the traces, which after propagation through ShiftRows and MixColumns allows us to do a classic CPA in the next round and recover round key K_4. As in § 3.3, the success of this stage depends on the least significant byte of the nonce N_{15}. This is shown in Figure 15, although now, even the worst case can lead to a successful attack if the SNR is sufficiently high (SNR ≥ 1). Removing part of the traces can still help to improve the success probability. This is again indicated in Figure 14, right and in Figure 15 by `0x80`*. From now on, we always perform the fourth step of the attack with the same selection of traces as step 3. With the recovery of the round key K_4, it is trivial to reverse the key schedule and calculate the master key K_1. Next, we can calculate the nonce N by performing the AES rounds backward from the state Z_3.

The success probabilities in Figures 9 to 15 were each obtained in experiments using the correct information from the previous steps. They are thus actually conditional probabilities, conditioned on the success of the previous step of the attack. Hence, by multiplying these success rates, we obtain the success rate of the entire attack. This is shown in Figure 16.

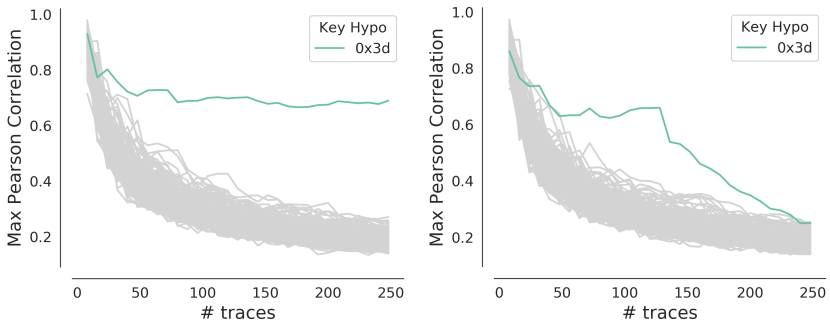


Figure 14: Maximum Correlation coefficients as a function of the number of traces in Step 4 with SNR=1.0 and N_{15} = `0x00` (left) or N_{15} = `0x80` (right).

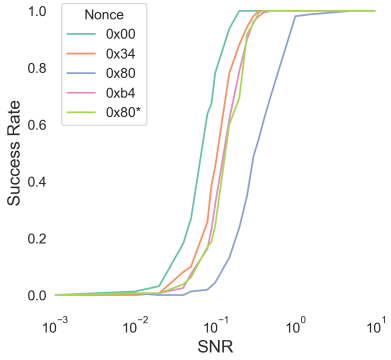


Figure 15: Success Rates of the fourth step of the attack for various nonces with 256 traces (except `0x80`* with 128 traces).

3.5 Discussion

Jaffe's Original Attack. In the original attack by Jaffe [Jaf07], the first step that recovers $Z_{1,15}$ by hypothesizing on 15 bits is identical. The difference with this paper is that Jaffe uses 2^{16} power traces and can therefore also recover $Z_{1,14}$ with this approach. This requires hypothesizing on 16 bits and is thus more complex. In our case, with only 256 traces, byte $Z_{1,14}$ is almost constant, hence we must follow a different approach. This way, we also avoid the hypothesis on 16 bits. After retrieving $Z_{1,14}$ and $Z_{1,15}$, Jaffe selects a subset of traces in which the remaining bytes $Z_{1,0}, \ldots, Z_{1,13}$ are constant. In the second round of encryption, the attack follows the same approach as described in § 3.2. With both $Z_{1,15}$ and $Z_{1,14}$ known, it is possible to recover the first two columns: $Z_{2,0}, \ldots, Z_{2,7}$. Finally, in round three, two bytes per column are known and variable, so the same approach as in round two allows retrieval of the entire state Z_3. The last step of the attack is again analogous to ours.

More traces available? The NIST recommendations currently allow an adversary to obtain up to 4 096 traces of AES-CTR, which is well above 256. What happens to the attack success probability when we can actually use this full number of traces? A general understanding in side-channel analysis is that increasing the number of traces always increases the success probability of an attack. This is certainly also true for the first step of the attack, since the least significant byte of the counter is not affected by a carry from

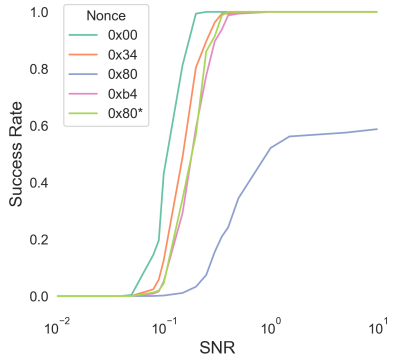


Figure 16: Success Rates of the attack for various nonces with 256 traces (except `0x80`* with 128 traces).

a previous byte. With up to $4\,096 = 2^{12}$ encryptions in CTR-mode, the same can be said for the second step of the attack, since a counter to 2^{12} is not enough to invalidate the assumption that three bytes in the first column are constant. In the third and fourth step however, the success very much relies on the assumption that three bytes in each column are (quasi-)constant, which means increasing the number of traces would only increase the "noise". However, having more than 256 traces available can certainly help, since one can select from them the perfect subset of traces. For example, if the attacker suspects from the first 256 traces that the least significant byte of the nonce is near `0x80`, (s)he only has to throw away the first 128 traces and use the next 256 traces to turn a worst case scenario into a best case scenario (nonce `0x00`). Similarly, with any other nonce N_{15} the adversary can compute exactly how many traces to throw away $(256 - N_{15})$ to obtain the subset with nonce `0x00`. It is important that step 3 and 4 of the attack are still performed with only 256 traces, in order for the assumptions on the white squares to hold.

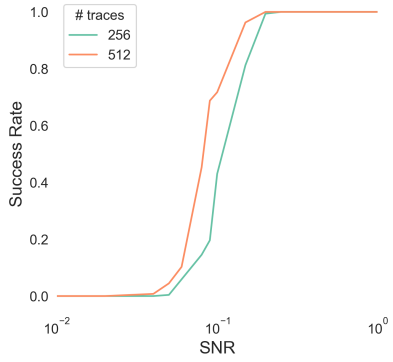


Figure 17: Success Rate of the attack with 256 traces (best case) or 512 traces (any case)

Hence, if an adversary has 512 traces at his disposal, the success rate of the attack will always follow the best case in Figure 16, or even a bit better, since the first two steps can use the full amount of 512 traces (see Figure 17). We will illustrate this method in an application in Appendix C.

The Rippling Carry. There is one more case we did not consider in the above description of the attack. We mention it here, since it does not significantly affect the attack. In Figure 5, we assume that the white squares are completely constant throughout a set of 256 traces and that only the grey square can toggle once. However, if $X_{1,14} = \texttt{0xFF}$, its toggling to `0x00`, will actually create a non-zero carry which affects $X_{1,13}$ and makes it increment as well. If that byte is `0xFF` as well, the carry propagates to the next byte, and so on.

While this is something to keep an eye on when recovering the nonce N from X_1, it should not affect the first four steps of the attack. The toggling of any other byte from one value to another will happen at the same time as the toggling of $X_{1,14}$. Hence, the situation in round 3 and 4 of the attack remains the same: a part of the traces corresponds to one constant (C_3) and another part uses constant C_3'.

Step two of the attack is affected if the carry ripples all the way to byte $X_{1,10}$, which affects the first column of the state in round 2. This would mean that $N_{11} = N_{12} = N_{13} = N_{14} = \texttt{0xFF}$ and is thus a very special case.

4 Experimental Validation

To test our attack on a real device, we program a Cortex-M4 CPU with an AES-CTR implementation. For this, we use the ChipWhisperer CW308T-STM32F3 target mounted on the CW308 UFO board. The UFO board is connected to the ChipWhisperer-Lite board. We use the ChipWhisperer Capture software for programming the device, communicating with the device and for collecting power measurements. The clock frequency of the target and sample rate of the scope are set to the ChipWhisperer defaults.

We collect exactly 256 traces of 12 000 samples each, consisting of approximately the first four rounds of AES. The nonce and key are chosen randomly by the ChipWhisperer Capture software. Thanks to the ChipWhisperer measurement setup, the traces are well aligned. An example trace is shown in Figure 18. For efficiency, we will use only the SubBytes region of each round in the corresponding steps of the attack.

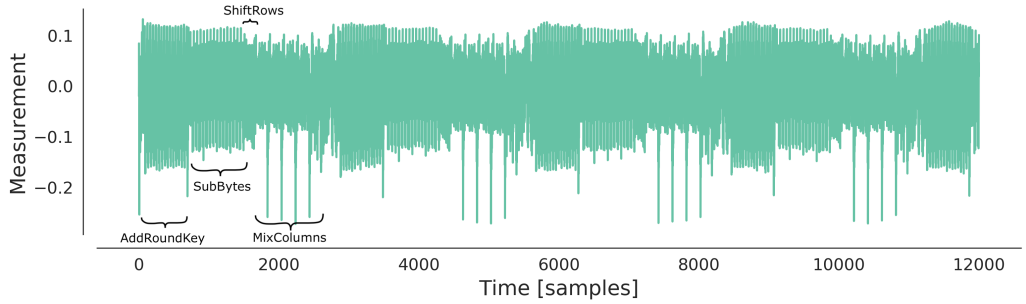


Figure 18: Example trace of the first four rounds of AES-CTR on a Cortex-M4.

Round 1. In the first step of the attack, the winning hypothesis achieves almost double the correlation of the others. We learn that $(b, K_{1,15,lo}, N_{15,lo}) = (0, \texttt{0x57}, \texttt{0x0D})$ (see Figure 19). This means that $(K_{1,15}, N_{15})$ is either $(\texttt{0x57}, \texttt{0x0D})$ or $(\texttt{0xD7}, \texttt{0x8D})$. We already have here an example of a possible worst-case scenario.

Round 2. In Round 2, we recover the constants $C_2 = [\texttt{0x65}, \texttt{0x22}, \texttt{0x52}, \texttt{0x52}]$ (see Figure 20).

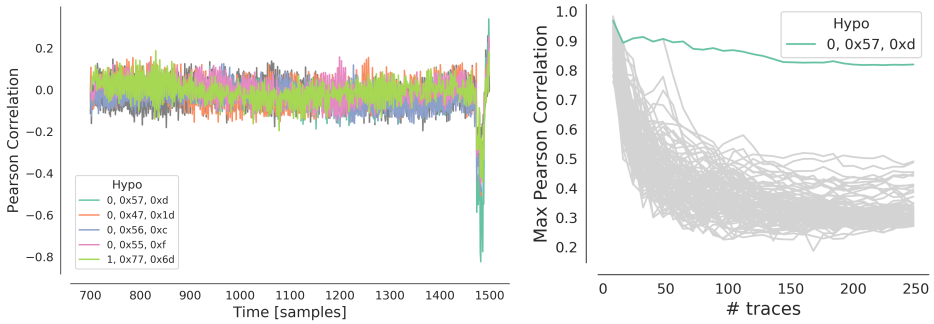


Figure 19: Pearson Correlation coefficients in Step 1, with 256 traces as a function of the time samples (left) and their maximum as a function of the number of traces (right).

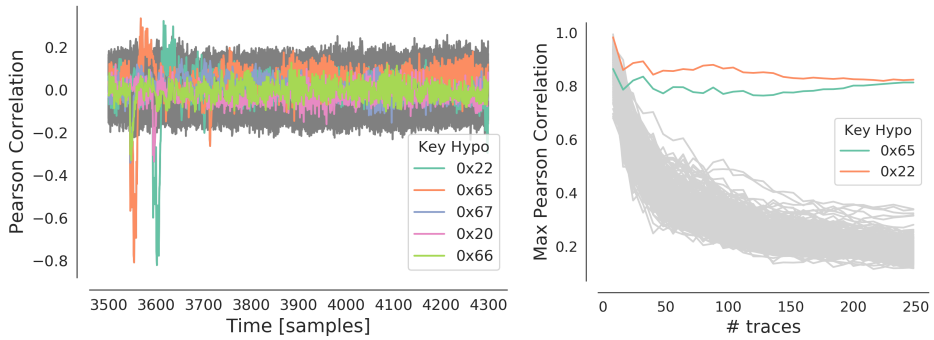


Figure 20: Pearson Correlation coefficients in Step 2 (bytes 0 and 1), with 256 traces as a function of the time samples (left) and their maximum as a function of the number of traces (right).

Round 3. In round 3, we start with the attack to recover constant $C_{3,0}$. The result is shown in Figure 21, left and gives a strong suspicion that the least significant byte of the nonce is actually `0x8D`, since we see a close race between two constants. This means that the least significant key byte should be `0xD7`.

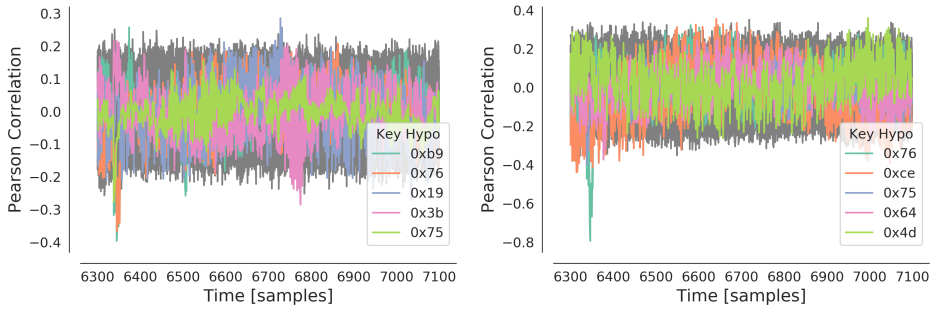


Figure 21: Pearson Correlation coefficients in Step 3 with 256 traces (left) and 128 traces (right) (byte 0).

If we perform the same attack with only half the traces (see Figure 21, right), we obtain a clear winner. In Figure 22, we show the maximum correlation coefficients as a function of the number of traces used. We thus suspect that $N_{15} =$ `0x8D` and continue the attack

with only half the traces. We recover the following constants in round 3:

$C_3 = [\texttt{0x76}, \texttt{0x23}, \texttt{0x3D}, \texttt{0xCE}, \texttt{0x70}, \texttt{0xB9}, \texttt{0xCB}, \texttt{0xA4}, \texttt{0x46}, \texttt{0x32}, \texttt{0x6E}, \texttt{0x84}, \texttt{0xA0}, \texttt{0x64}, \texttt{0x68}, \texttt{0x09}]$

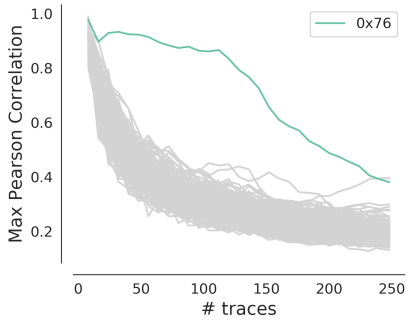


Figure 22: Maximum Correlation coefficients as a function of the number of traces in Step 3 (byte 0).

Round 4. Finally, still using half the traces (see Figure 23), we recover the following round key in Round 4:

$K_4 = [\texttt{0x7B}, \texttt{0xFF}, \texttt{0x7A}, \texttt{0xD7}, \texttt{0x0D}, \texttt{0x28}, \texttt{0x2E}, \texttt{0xE3}, \texttt{0x00}, \texttt{0x3E}, \texttt{0xD1}, \texttt{0x58}, \texttt{0xCB}, \texttt{0x87}, \texttt{0x0B}, \texttt{0xBB}]$

If we perform the Key Schedule backward, we find that the Master Key is

$K_1 = [\texttt{0xCC}, \texttt{0x8E}, \texttt{0x0F}, \texttt{0x06}, \texttt{0x0D}, \texttt{0xE8}, \texttt{0x3E}, \texttt{0x80}, \texttt{0x24}, \texttt{0xBE}, \texttt{0x94}, \texttt{0x73}, \texttt{0xBD}, \texttt{0x6E}, \texttt{0x8E}, \texttt{0xD7}]$

Looking at the least significant byte, we can now confirm that $(K_{1,15}, N_{15}) = (\texttt{0xD7}, \texttt{0x8D})$ and that we probably performed the attack correctly. Indeed, when we perform AES backward from Z_3, we obtain

$X_1 = [\texttt{0xE6}, \texttt{0x10}, \texttt{0x3B}, \texttt{0x22}, \texttt{0x55}, \texttt{0x62}, \texttt{0x7E}, \texttt{0xE6}, \texttt{0xBE}, \texttt{0x93}, \texttt{0x18}, \texttt{0xBD}, \texttt{0x71}, \texttt{0xB7}, \texttt{0xBA}, \texttt{0x8D}]$

which is equal to the nonce N, since we used the first half of the traces. Pay attention when using the second half or when the majority of the traces use the constant C_3'. In that case, we recover $X_{1,14} = N_{14} + 1 \mod 256$.

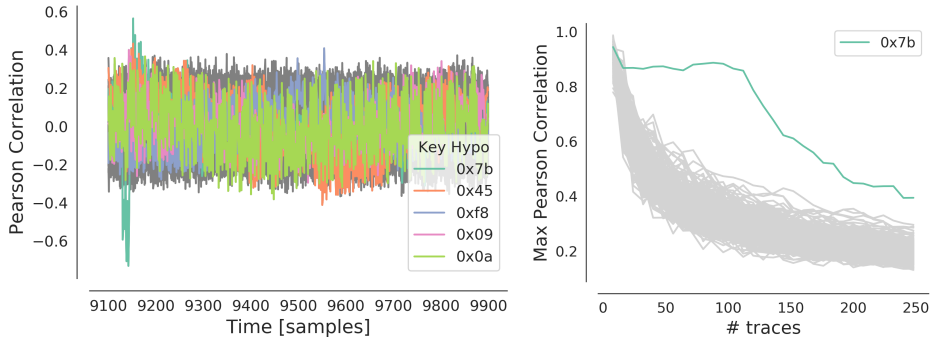


Figure 23: Pearson Correlation coefficients in Step 4 (byte 0), with 128 traces as a function of the time samples (left) and their maximum as a function of the number of traces (right)

Device Behaviour. Now that we know the key and nonce, we can investigate the relation between Hamming weights and their leakage on the Cortex-M4. In Figure 24, we plot the Hamming weight of one point of interest (byte 15 in the first SubBytes) across 256 encryptions on the x-axis and the measurements of the corresponding sample in the power traces on the y-axis. The corresponding trace sample is chosen as the time sample where the power measurements have the largest Pearson correlation with this array of Hamming weights. We also estimate the SNR at this time sample as

$$SNR = \frac{Var(signal)}{Var(noise)}$$

The signal is constructed by replacing each measurement with the average of all measurements for that Hamming weight, as is done in [MOP07]. The noise is approximated by subtracting these averages from the actual measurements. This way, we obtain $SNR \approx 2.18$ at the time sample corresponding to state byte 15 after the S-box in the the first round.

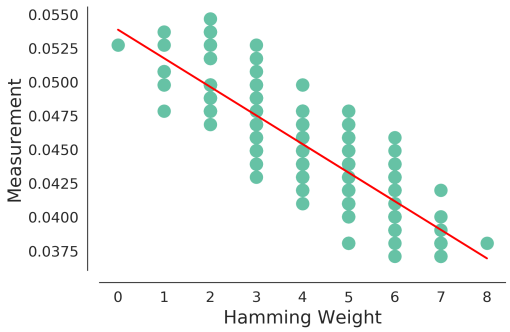


Figure 24: Measured leakages vs. actual Hamming weights on the Cortex-M4.

Other devices. We additionally successfully performed the attack on an Arduino Uno and the ChipWhisperer-lite XMEGA target. For the results, we refer to Appendices B and C. The trace files and a JuPyter Notebook performing the above attack can be found online[1].

5 Discussion

5.1 The Worst Case Nonce

Figure 16 gives a bleak impression of the attack's sensitivity to the least significant byte of the nonce. However, the worst case scenario is not as bad as it seems.

Firstly, it does not imply the existence of a protection mechanism since biasing the nonces towards `0x80` would only reduce the search space of the attacker.

Secondly, we have shown that removing part of the traces may improve the chance of success. This trick is not limited to the worst case, as the attacker has knowledge of $N_{15,lo}$ and can thus always compute the right number of traces to throw away. Without knowing $N_{15,hi}$, there are two possible ways to do it, but one will clearly improve results, while the other will make them worse. However, depending on the amount of noise in the traces, extra traces may still improve the performance. As stated in § 3, in the worst case, we can say the attack requires 512 traces, which is still far less than 4 096. With 512 traces

[1] https://github.com/LaurenDM/AttackAESCTR

available, step 3 and 4 always achieve the best success rate and the dependency on the nonce thus disappears.

Finally, PRNGs tend to be used for applications that need a continuous supply of randomness. If the attacker really has access to at most 256 traces per PRNG request and the worst case scenario occurs, the next PRNG request will have a different nonce. NIST prescribes the PRNG to be reseeded after at most 2^{48} requests. As soon as the adversary manages to recover the key and nonce for one request, the internal state of the PRNG is known and the future random outputs can be calculated as long as the PRNG is not reseeded.

5.2 Variations on a theme: DDLA

In the original work of Jaffe [Jaf07], the attack was not performed using CPA, but rather DPA as introduced in the original work of Kocher [KJJ99]. In this work, we opted for CPA, but any similar SCA methodology can replace this. For example, recently, a non-profiled SCA using deep learning (DDLA) was introduced by Timon [Tim19], which was shown to be more resilient in case of misaligned traces.

The main idea of DDLA is to train a neural network for various key guesses. With each training, the inputs to the network are the traces and the outputs are the corresponding leakage hypotheses for a particular key guess. For the correct key guess, the network accuracy during training is supposed to grow a lot faster than for wrong key guesses. The use of a convolutional neural network in this method is more robust in the case of misaligned traces. For more details on DDLA, we refer to [Tim19].

Application to AES-CTR. In the original paper, this methodology uses around 3 000 traces. With only 256 traces available, it is more likely that the network "memorizes" the data and starts to overfit. Choosing a suitable network architecture is therefore a bit more challenging in our case. We used the same traces as in Section 4, but as in [Tim19], created a misalignment by shifting each trace by a random offset between -25 and 25. This is not a large offset, but it is sufficient to make regular CPA fail. Our neural network starts from the CNN_{exp} from Timon [Tim19], but we use 8 filters of size 100 in the first convolutional layer and we replace the second convolutional layer with a 10-neuron dense layer. We also use the most significant bit of the S-box output in our hypotheses.

It is standard to randomly initialize a neural network's weights. The initial weights have some influence on the accuracy of the training, which is why training the network for the same hypothesis twice can give different results in accuracy. Therefore, we noticed that, when training the same network for different hypotheses and comparing their accuracies, it is better to always use the same initial weights in the neural network.

Figure 25 shows the resulting accuracies for the first step of the attack. Even with only 256 traces, this methodology works. It takes a lot of computation time, since we need to train the network 2^{15} times, but it succeeds where regular CPA does not.

Distinguishing time samples. In the second step of the attack, it is important to know the most defining time samples in the trace in order to distinguish the winning hypotheses of two bytes in one column. For this purpose, we use the sensitivity analysis as described in [Tim19, §3.2.2]. The results are shown in Figure 26 with the accuracies on the left and the sensitivities on the right. They show clearly which of the two constants appears first in the trace. The results correspond to those of Section 4.

We can thus conclude that other variants of DPA methods can be applied in the attack. DDLA is a good choice if the traces are misaligned, but does take quite some computation time.

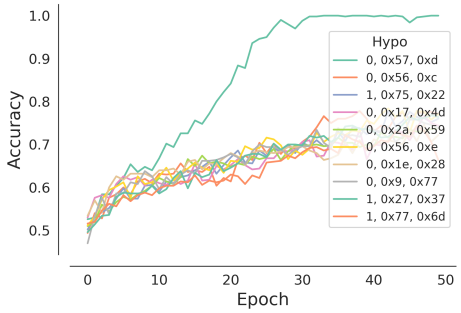


Figure 25: Performing the first step of the attack with DDLA, using 256 traces.

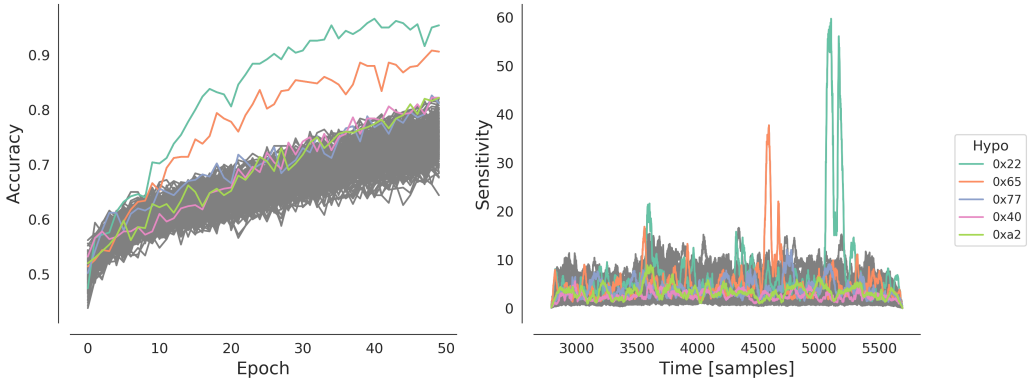


Figure 26: Performing the second step of the attack (byte 0 and 1) with DDLA, using 256 traces. Accuracies (left) and sensitivity analysis (right).

5.3 Blind SCA

An alternative approach to attack a CTR mode with unknown nonce is to use a blind SCA as described at CHES 2017 by Clavier *et al.* [CR17]. This methodology stems from the observation that the joint probability distribution of $(HW(X_{i,j}), HW(Z_{i,j}))$ with $Z_{i,j} = S(X_{i,j} \oplus K_{i,j})$ depends on the secret key $K_{i,j}$. In [CR17], this is exploited by computing the maximum likelihood that the leakages observed for $X_{i,j}$ and $Z_{i,j}$ occur in the case of a specific key guess.

This method has as advantage that it does not even require the CTR mode as it does not require specific knowledge on the plaintext X_1. It can thus be applied to PRNGs based on different modes of operation, but only to recover the key. However, we found that the methodology is very sensitive to noise and less effective in this case than the ones described in this paper and Jaffe's [Jaf07]. We were not able to recover the secret key from our devices running AES in CTR mode, using 4 096 traces.

Application to AES-CTR. Since not all plaintext bytes vary in CTR mode, it makes more sense to apply the blind attack to the last round, where the ciphertext bytes are constantly changing. We noticed a number of drawbacks to blind SCA compared to regular CPA in this application. For example, the blind attack requires a precise estimation of the location of the two points of interest: $X_{i,j}, Z_{i,j}$ corresponding to a byte $X_{i,j}$ at the input of AddRoundKey and $Z_{i,j} = S(K_{i,j} \oplus X_{i,j})$ the byte at the output of SubBytes. Even if one manages to pinpoint the correct samples in the traces, the attack also requires the leakages at these points to be converted to Hamming weight estimations. In the work of [CR17],

this is done by estimating coefficients α and β such that the leakage is approximately $\alpha HW + \beta$. However, when we compare the measured leakages on a Cortex-M4 device with the actual Hamming weights in Figure 24, we see that one easily estimates the wrong Hamming weights from these.

In contrast, for a regular CPA attack, it suffices to identify only an approximate region of interest, since the Pearson correlation coefficient can be computed for many time samples. Moreover, it is not required to estimate the Hamming weights, since CPA can be applied directly to the measurements obtained from the oscilloscope (no matter the leakage unit). The same can thus be said for the CPA-based attack of this work.

Experiments. We tried a simplified attack, where the points of interest and α, β are given to the adversary: We collected power measurements both from an Arduino Uno and from a Cortex-M4. We computed the actual Hamming weight values using a simulation of AES-CTR with the same nonce and key as was sent to the device. We determined the points of interest in the real power traces by computing the correlation of the trace points with the real Hamming weights. We then used the least squares method to determine the coefficients α, β in the relationship between the real Hamming weights and the leakage units of the trace. We used the maximum number of traces available according to the NIST recommendations: 4 096. Even then, the blind SCA was only able to recover 11 of the 16 key bytes on the Arduino Uno device and 8 bytes on the Cortex-M4. We show figures for each key byte in Appendix D.

5.4 How (not) to use CTR mode

It is clear that the presence of the counter in CTR mode gives more information to the adversary than in the case of for example CBC mode. However, the possibility of the demonstrated attack does not imply that using a CTR mode-based PRNG is always a bad idea. By following a few simple guidelines when using the CTR_DRBG, the attack can be avoided. In this section we discuss some observations and recommendations for the use of AES-CTR in a PRNG. As an example, we consider the context of masked implementations against side-channel attacks, where online PRNGs are usually required to provide a continuous stream of randomness. This is a very interesting use case for the attack, since recovering the state of the CTR_DRBG once implies that the attacker can derive any future PRNG output. The attacker can then compute all the masks used in the masked implementation and perform a classic first-order DPA attack to recover the secret key. It does not matter then whether the masked implementation is first-, second- or even fifth-order secure. Recall that the CTR_DRBG does not provide forward secrecy as long as it is not reseeded and the recommended maximum number of request between reseeds is 2^{48} [BK15], which allows for more than enough traces for a first-order attack.

While we keep this application in mind, the discussion is of course also relevant for any other use of the CTR_DRBG, such as key generation or IV generation for protocols.

5.4.1 Observations

Hiding. A common hiding technique against side-channel attacks is to randomize the order of the 16 S-box calculations during SubBytes. We saw in § 3.2 and § 3.3 that the order of execution is important to distinguish two of the constants in each state column. The hiding countermeasure therefore does not increase the number of traces required but can increase the complexity of the attack by increasing the number of possibilities to try in step 2 and step 3. However, it does not completely prevent our attack.

Hardware. Related to the shuffling of S-box calculations, a hardware PRNG implementation that performs all 16 S-boxes in parallel, does not allow to distinguish the two

equal-hypothesis constants in each column. More importantly, when the 128-bit state is being operated on in parallel, the signal-to-noise ratio is a lot smaller, since the leakage of one byte (the signal) only corresponds to approximately one sixteenth of the measurement (not including the noise) [MOP07]. Furthermore, in an unrolled implementation, it is difficult to separate the power measurements of the 128-bit states of different rounds, even if the device is sampled at a very high rate. In other words, the SNR of such a hardware implementation would be much worse than for our software implementations and it is unlikely that even a regular CPA attack (with knowledge of the plaintext) would succeed with only 256 traces.

Real-world Crypto. Despite the NIST recommendations, we found two commercial CTR_DRBG implementations which do put a proper limit on the request size. In the open-source mbed TLS library [arm], we can see that the maximum number of requested bytes per CTR_DRBG call is 1024, which is equivalent to only 64 encryptions with AES-128 in CTR mode. Even more secure is a CTR_DRBG implementation by Texas Instruments [Ins], which puts the limit at 2^{11} bits, or equivalently only 16 AES-CTR encryptions.

5.4.2 Recommendations

Types of counters. As previously mentioned, the attack is not prevented if the counter field starts in a byte X_{1,j^*} other than the least significant byte $X_{1,15}$. The first round of the attack then simply recovers Z_{1,j^*}, which propagates to a different column in step 2, but does not change the overall approach or complexity. On the other hand, the NIST document on modes of operation also suggests the possibility to use an LFSR as incrementing function in AES-CTR, as long as its period is sufficiently long [Dwo01]. A good choice of LFSR would update bits which are spread over the entire 128-bit AES state, rather than just one byte, thereby preventing the divide-and-conquer approach that enables targeting key bytes in side-channel attacks. In contrast with a normal incrementing counter, it is thus possible for a CTR_DRBG based on an LFSR to resist our attack, even if 4 096 traces are available.

Size of Requests. Each new request to the NIST CTR_DRBG results in the computation of AES-CTR with a different nonce and key, because of the CTR_DRBG _Update function, which is performed with every randomness request. Hence, when using such a PRNG for a masked implementation, it would clearly be less secure to perform a single (large) request for all the random bits needed in a masked AES than to perform multiple (small) requests, such as one for each masked S-box evaluation separately. Even worse would of course be to not follow the NIST recommendations and to keep using the same key across PRNG requests.

Conclusion for masked implementations. Through this section, we also want to start the discussion on whether PRNGs for masked implementations need their own side-channel protection, a question which is often sidestepped due to its "chicken-and-egg" character. Indeed, protecting a PRNG used for masked implementations against side-channel attacks, would require its own fresh randomness source in return. However, by keeping these observations and recommendations in mind and ensuring that the request sizes are sufficiently limited, this attack against AES-CTR can be avoided. For other modes of operations, it looks like also the blind SCA can be avoided on real devices if the number of available traces is limited. Under the assumption that the state (nonce and key) of the CTR_DRBG is not known to the side-channel adversary, it does not seem like masking the PRNG is necessary. An investigation of other PRNG constructions and attacks against them is an interesting direction for future research.

6 Conclusion

In this work, we demonstrated an attack on AES-CTR mode with unknown key and nonce in only 256 traces, a significant improvement over the previous attack by Jaffe [Jaf07]. Most importantly, this number of traces shows that a CTR_DRBG following the NIST specification can be vulnerable to this attack, as it currently allows adversaries to obtain as much as 4 096 traces of a CTR_DRBG performing AES-CTR. We demonstrated the feasibility of our attack on several real devices such as a Cortex-M4 and make our implementations openly available for reproducability.

We explored alternative methods such as DDLA for misaligned traces and blind SCA, which does not require the CTR-mode assumption.

We start the discussion on PRNGs for masked implementations (*i.e.* a PRNG for which an adversary can only observe the power consumption), a topic for which very little research is available. Using the observations from this attack, we can conclude that masking should not be necessary for such a PRNG, provided its correct use such as limiting the request size and updating the state between requests. The question remains on whether a construction as large as AES-CTR is necessary in this adversary model. An investigation of various PRNG constructions and their security in this context is an interesting direction for future work.

Acknowledgements

The author would like to thank Josep Balash, Arthur Beckers, Begül Bilgin, Vincent Rijmen and Lennert Wouters. The author is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO).

References

[arm] arm. Mbed tls. https://tls.mbed.org/api/ctr__drbg_8h.html#a5b787e6157d91055d7c07d40f519cf52.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques 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.

[BK15] Elaine Barker and John Kelsey. Recommendations for random number generation using deterministic random bit generators. NIST SP 800-90A Rev. 1, June 2015.

[CR17] Christophe Clavier and Léo Reynaud. Improved blind side-channel analysis by exploitation of joint distributions of leakages. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 24–44. Springer, 2017.

[Dwo01] Morris Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. NIST SP 800-38A, December 2001.

[GBTP08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic*

Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings, volume 5154 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2008.

[Ins] Texas Instruments. Random number generation using msp430fr59xx and msp430fr69xx microcontrollers. http://www.ti.com/lit/an/slaa725/slaa725.pdf.

[Jaf07] Joshua Jaffe. A first-order DPA attack against AES in counter mode with unknown initial counter. In Pascal Paillier and Ingrid 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.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael 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.

[LDL14] Yanis Linge, Cécile Dumas, and Sophie Lambert-Lacroix. Using the joint distributions of a cryptographic function in side channel analysis. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, volume 8622 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2014.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[Tim19] Benjamin Timon. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):107–131, 2019.

A Constants

Table 1: MixColumns constants f_j and k_j to use in step 3 of the attack (Eqn (3))

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
f_j	2	1	1	3	1	1	3	2	1	3	2	1	3	2	1	1
k_j	0	0	0	0	3	3	3	3	2	2	2	2	1	1	1	1

B Application to Arduino Uno

Rounds 1 & 2. In the first step, we obtain $(b, K_{1,15,lo}, N_{15,lo}) = (0, \texttt{0x3c}, \texttt{0x34})$ (see Figure 27).

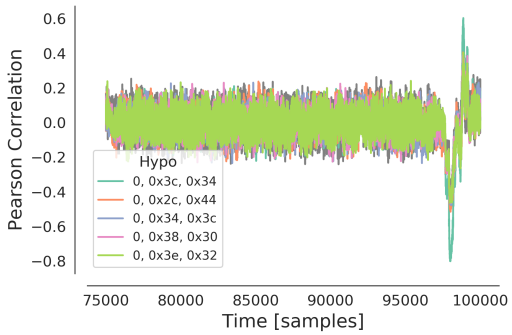


Figure 27: Pearson Correlation coefficients in Step 1 with 256 traces.

This means that $(K_{1,15}, N_{15})$ is either $(\texttt{0x3c}, \texttt{0x34})$ or $(\texttt{0xBc}, \texttt{0xB4})$. Next, we find the constants $C_2 = [\texttt{0x94}, \texttt{0xAC}, \texttt{0x2F}, \texttt{0x92}]$ (see Figure 28).

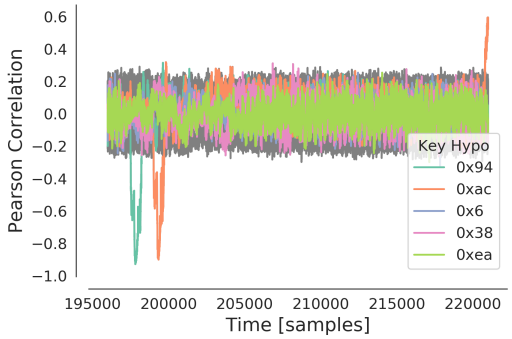


Figure 28: Pearson Correlation coefficients in Step 2 (bytes 0 and 1) with 256 traces.

Round 3. In round 3, we immediately see that the current set of traces is sufficient (see Figure 29) and we recover the following constants:

$C_3 = [\texttt{0x38}, \texttt{0xC6}, \texttt{0x16}, \texttt{0xD8}, \texttt{0xE8}, \texttt{0x54}, \texttt{0x63}, \texttt{0xE6}, \texttt{0x50}, \texttt{0xBF}, \texttt{0xF5}, \texttt{0x00}, \texttt{0x11}, \texttt{0x95}, \texttt{0x98}, \texttt{0x44}]$

At this point, we do not know whether this is C_3 or C_3'. This depends on the most significant bit of N_{15}.

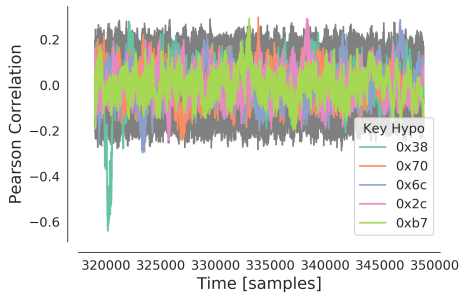


Figure 29: Pearson Correlation coefficients in Step 3 (byte 0) with 256 traces.

Round 4. In the last step, we find the fourth round key:

$K_4 = [\texttt{0x3D}, \texttt{0x80}, \texttt{0x47}, \texttt{0x7D}, \texttt{0x47}, \texttt{0x16}, \texttt{0xFE}, \texttt{0x3E}, \texttt{0x1E}, \texttt{0x23}, \texttt{0x7E}, \texttt{0x44}, \texttt{0x6D}, \texttt{0x7A}, \texttt{0x88}, \texttt{0x3B}]$

which after reversing the key schedule gives us the master key:

$K_1 = [\texttt{0x2B}, \texttt{0x7E}, \texttt{0x15}, \texttt{0x16}, \texttt{0x28}, \texttt{0xAE}, \texttt{0xD2}, \texttt{0xA6}, \texttt{0xAB}, \texttt{0xF7}, \texttt{0x15}, \texttt{0x88}, \texttt{0x09}, \texttt{0xCF}, \texttt{0x4F}, \texttt{0x3C}]$

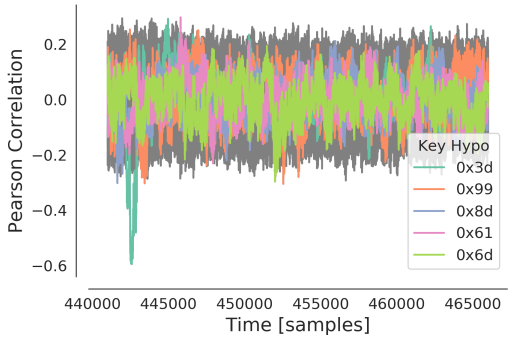


Figure 30: Pearson Correlation coefficients in Step 4 (byte 0) with 256 traces.

Since $K_{1,15} = \texttt{0x3C}$, we know that $N_{15} = \texttt{0x34}$ and thus that we recovered C_3 in step 3. Finally, we find the nonce:

$N = [\texttt{0x32}, \texttt{0x43}, \texttt{0xF6}, \texttt{0xA8}, \texttt{0x88}, \texttt{0x5A}, \texttt{0x30}, \texttt{0x8D}, \texttt{0x31}, \texttt{0x31}, \texttt{0x98}, \texttt{0xA2}, \texttt{0xE0}, \texttt{0x37}, \texttt{0x07}, \texttt{0x34}]$

The entire attack thus succeeds with 256 traces. The relation between Hamming weight and the measurements of the Arduino Uno is shown in Figure 31. We calculate that the SNR at this point (byte 15 in the first SubBytes) is 5.21.

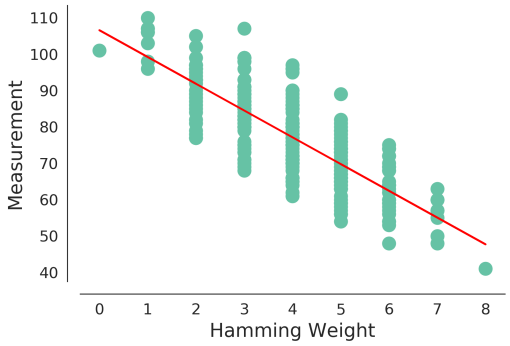


Figure 31: Measured leakages vs. actual Hamming weights on the Arduino Uno.

C Application to XMEGA

Rounds 1 & 2. We first find $(b, K_{1,15,lo}, N_{15,lo}) = (1, \texttt{0x36}, \texttt{0x0E})$ (see Figure 32), which means that $(K_{1,15}, N_{15})$ is either $(\texttt{0x36}, \texttt{0x8E})$ or $(\texttt{0xB6}, \texttt{0x0E})$. Again, we could be close to a worst-case scenario.

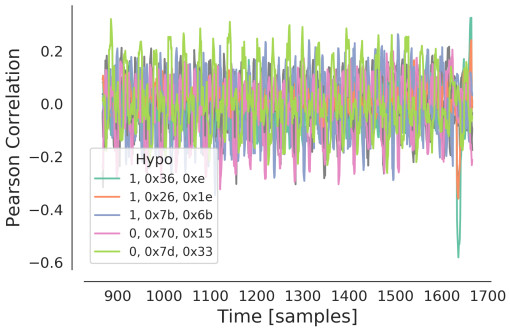


Figure 32: Pearson Correlation coefficients in Step 1 with 256 traces.

In step two, we recover $C_2 = [\texttt{0x4B}, \texttt{0x17}, \texttt{0xAE}, \texttt{0xB8}]$ (see Figure 33).

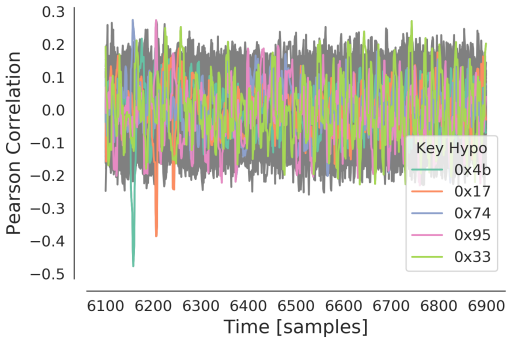


Figure 33: Pearson Correlation coefficients in Step 2 (bytes 0 and 1) with 256 traces.

Round 3. In round 3, using the complete trace set, we do not get very clear results (see Figure 34). It looks like a race between `0xB2` and `0xF6`, but even their correlation

coefficients are not significantly larger than the others.

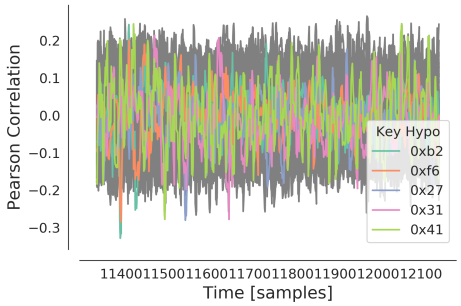


Figure 34: Pearson Correlation coefficients in Step 3 (byte 0) with 256 traces.

In this section, we demonstrate how to use the availability of more traces to obtain exactly 256 traces for any nonce. We therefore acquired 512 traces (indexed 0 to 511) instead of 256. We know that the least significant byte of the nonce N_{15} is either `0x0E` or `0x8E` and the race in Figure 34 again makes us suspect that it is the latter. However, even if it is not that obvious, we can try out both cases. In the first case, the optimal trace set would be from trace number $256 - \texttt{0xE} = 242$ to 498. Otherwise, it would be better to use traces $256 - \texttt{0x8E} = 114$ to 370. Figure 35 shows both possibilities and confirms clearly that N_{15} must be `0x8E`.

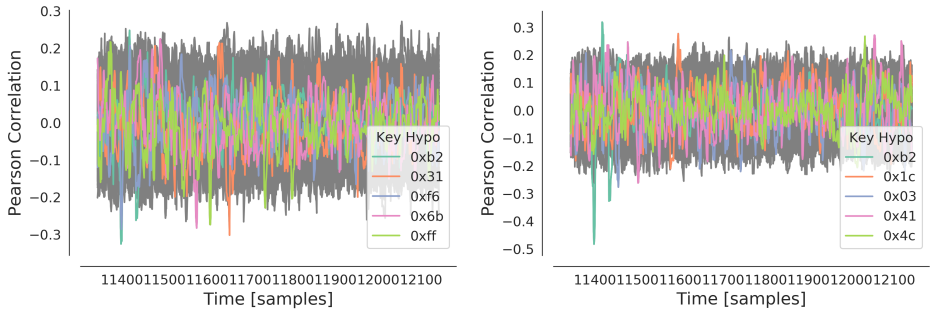


Figure 35: Pearson Correlation coefficients in Step 3 (byte 0) with traces 242-498 (left) and traces 114-370 (right).

Under this assumption and using traces 114 to 370, we obtain the following constants:

$$C_3' = [\texttt{0xB2}, \texttt{0x18}, \texttt{0x25}, \texttt{0x33}, \texttt{0x5A}, \texttt{0x91}, \texttt{0x31}, \texttt{0x8F}, \texttt{0x9C}, \texttt{0x80}, \texttt{0x46}, \texttt{0x43}, \texttt{0xEB}, \texttt{0xBD}, \texttt{0x04}, \texttt{0x8F}]$$

Pay attention that we are now intentionally targeting the traces *after* $X_{1,14}$ toggles, so we know we recovered C_3' and not C_3.

Round 4. In step 4, we obtain the fourth round key and thus also the master key:

$$K_4 = [\texttt{0x17}, \texttt{0x40}, \texttt{0x04}, \texttt{0xFF}, \texttt{0x6E}, \texttt{0xC1}, \texttt{0x0C}, \texttt{0x17}, \texttt{0x0C}, \texttt{0x25}, \texttt{0xF3}, \texttt{0xD4}, \texttt{0x2C}, \texttt{0x9C}, \texttt{0xD1}, \texttt{0x0A}]$$
$$K_1 = [\texttt{0x0D}, \texttt{0xF7}, \texttt{0xB8}, \texttt{0xAF}, \texttt{0x37}, \texttt{0x40}, \texttt{0xAC}, \texttt{0xC4}, \texttt{0x27}, \texttt{0x37}, \texttt{0xE6}, \texttt{0x8B}, \texttt{0x59}, \texttt{0x38}, \texttt{0x2A}, \texttt{0x36}]$$

When we reverse the AES rounds, we obtain X_1.

$$X_1 = [\texttt{0x06}, \texttt{0xA3}, \texttt{0x0A}, \texttt{0x52}, \texttt{0x69}, \texttt{0x0A}, \texttt{0xC9}, \texttt{0xA0}, \texttt{0x91}, \texttt{0x49}, \texttt{0xE4}, \texttt{0xAF}, \texttt{0xFB}, \texttt{0xC1}, \texttt{0x8A}, \texttt{0x8E}]$$

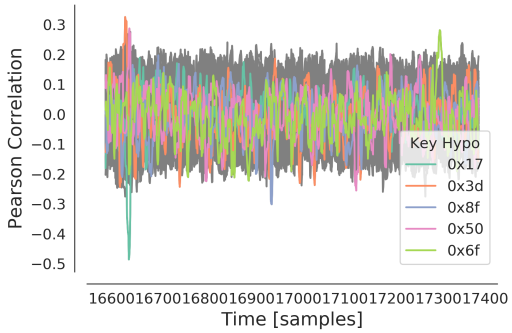


Figure 36: Pearson Correlation coefficients in Step 4 with traces 114-370 (byte 0).

Note however that this is not the nonce N, since we used the set of traces *after* $X_{1,14}$ toggles. We thus recovered $X_{1,14} = N_{14} + 1 \mod 256$. The nonce is therefore:

$N = [\texttt{0x06}, \texttt{0xA3}, \texttt{0x0A}, \texttt{0x52}, \texttt{0x69}, \texttt{0x0A}, \texttt{0xC9}, \texttt{0xA0}, \texttt{0x91}, \texttt{0x49}, \texttt{0xE4}, \texttt{0xAF}, \texttt{0xFB}, \texttt{0xC1}, \texttt{0x89}, \texttt{0x8E}]$

We plot some hamming weights and their measurements on the XMEGA in Figure 37. The SNR at this point (byte 15 in the first SybBytes) is calculated to be approximately 4.03.

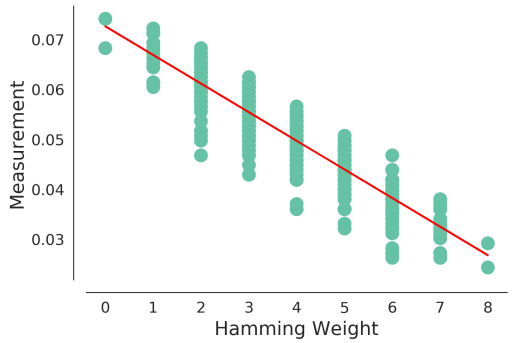
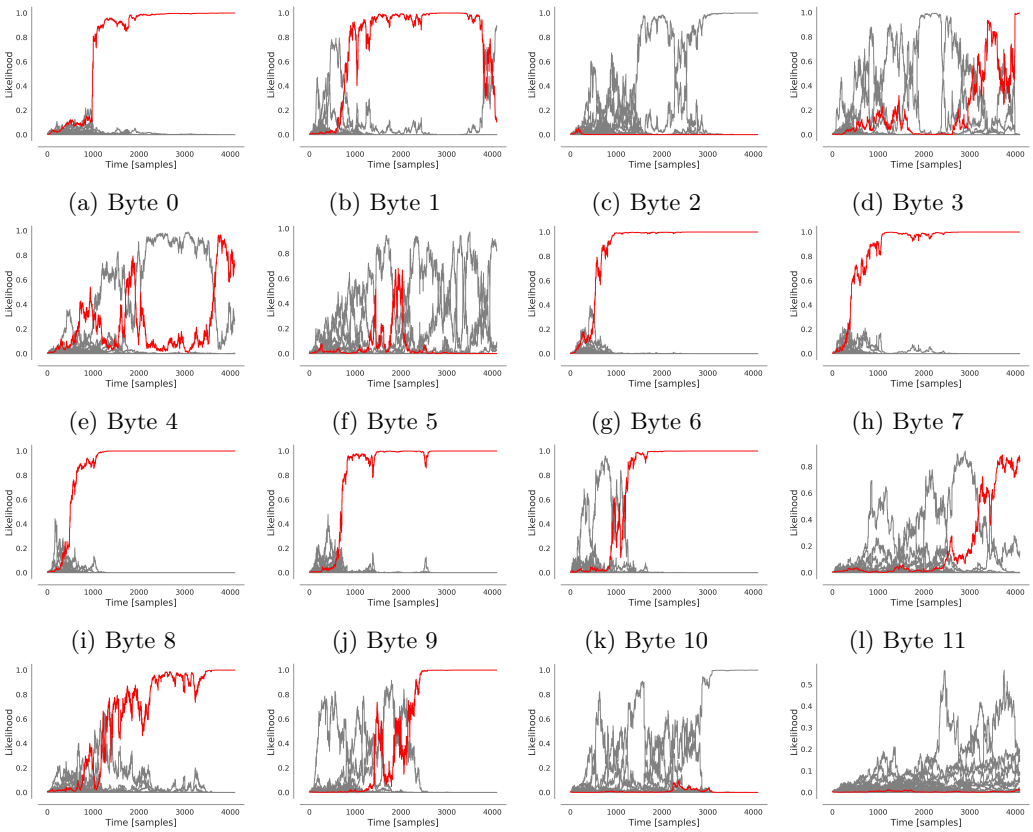


Figure 37: Measured leakages vs. actual Hamming weights on the XMEGA.

D Results of Blind SCA

The results in Figures 38 and 39 depict for each byte, the likelihood computed for each key guess in a blind SCA [CR17]. The correct key guess is indicated in red. Naturally, the attack succeeds if the likelihood of the correct key byte succeeds the others.



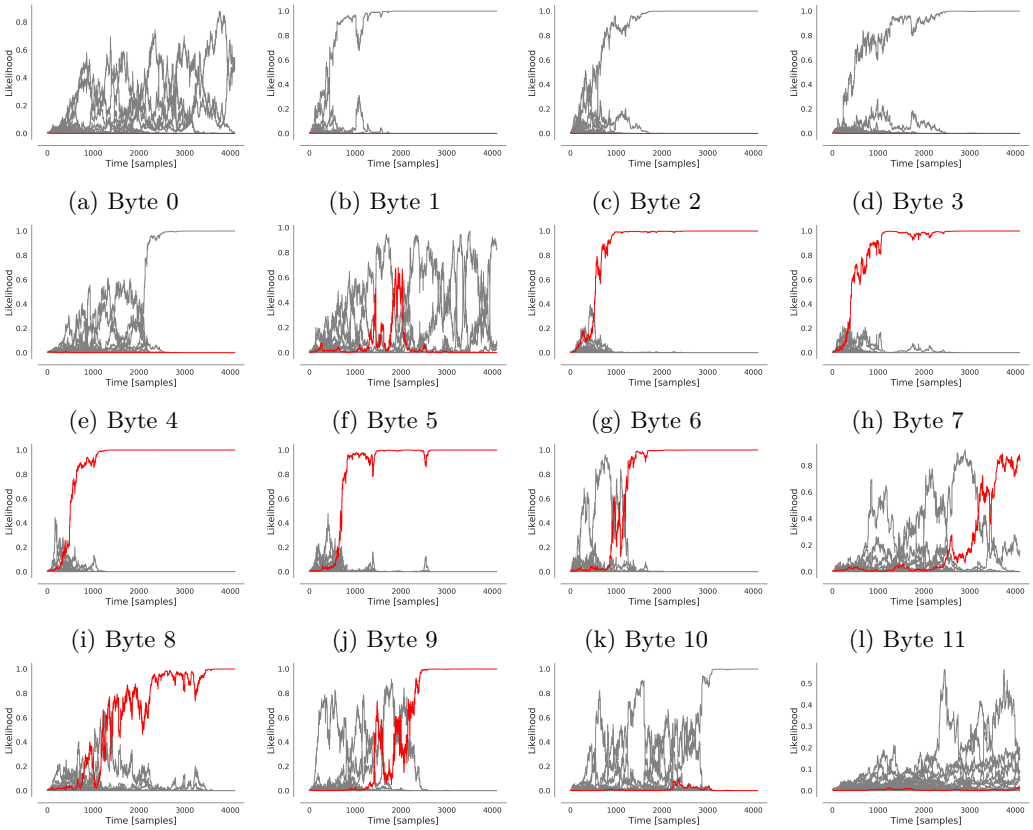
(a) Byte 0 (b) Byte 1 (c) Byte 2 (d) Byte 3

(e) Byte 4 (f) Byte 5 (g) Byte 6 (h) Byte 7

(i) Byte 8 (j) Byte 9 (k) Byte 10 (l) Byte 11

(m) Byte 12 (n) Byte 13 (o) Byte 14 (p) Byte 15

Figure 38: Recovery of the 16 key bytes in the last round of AES-CTR on Arduino Uno with blind SCA, using 4 096 traces. The red lines indicate the correct hypotheses.



(a) Byte 0 (b) Byte 1 (c) Byte 2 (d) Byte 3

(e) Byte 4 (f) Byte 5 (g) Byte 6 (h) Byte 7

(i) Byte 8 (j) Byte 9 (k) Byte 10 (l) Byte 11

(m) Byte 12 (n) Byte 13 (o) Byte 14 (p) Byte 15

Figure 39: Recovery of the 16 key bytes in the last round of AES-CTR on CortexM4 with blind SCA, using 4 096 traces. The red lines indicate the correct hypotheses.