Exploiting Linearity in White-Box AES with Differential Computation Analysis*

Jakub Klemsa(✉) and Martin Novotný

Czech Technical University in Prague, Czech Republic
jakub.klemsa@fel.cvut.cz, novotnym@fit.cvut.cz

Abstract. Not only have all current scientific white-box AES schemes been mathematically broken, they also face a family of attacks derived from traditional Side Channel Attacks, e.g., Differential Computation Analysis (DCA) introduced by Bos et al. Such attacks are very universal and easy-to-mount – they require neither knowledge of the implementation, nor use of reverse engineering.
In this paper, we particularly focus on DCA against white-box AES by Chow et al. which shows lower than 100% success rate as opposed to other schemes studied by Bos et al. We provide an explanation of this phenomenon while unraveling another weakness in the design of white-box AES by Chow et al. Based on our theoretical results, we propose an extension of the original DCA attack which has a higher chance of key recovery and might be adapted for other schemes.

Keywords: white-box AES, differential computation analysis, linear cryptanalysis

1 Introduction

Standard ciphers like AES (*Advanced Encryption Standard*, [30]) were designed with respect to so-called *black-box model*. In this model, an adversary is only allowed to observe ciphertexts of chosen plaintexts while she does not gain *any* other information about the encryption algorithm execution – neither intermediate values, nor timing. I.e., the adversary has an access to an *encryption oracle* while her goal is to recover the key or employ the oracle for effective decryption.

However, real-world hardware implementations like smart cards *do* leak certain portion of internal information through various side-channels, e.g., power consumption or electromagnetic radiation. This attack scenario is referred to as the *gray-box model*.

Later, there has emerged a need for the most extreme scenario where the adversary has a full control over the execution environment. Such a model is called the *white-box model*. Note that in this model, the adversary is free to observe or alter all intermediate values as well as instructions. It follows that the

* This work was supported by the Grant Agency of CTU in Prague, grant No. SGS19/109/OHK3/2T/13.

original cipher's intermediates—which typically allow for key recovery—must be somehow hidden or masked. In the wild, several techniques and layers of protection are being put in place, ranging from software obfuscation to mathematical approaches. In our paper, we will particularly focus on the mathematical point of view, however, our results will turn out to be highly practical.

1.1 White-Box Cryptography

In 2002, Chow et al. proposed white-box implementations of AES and DES [12,11] (WBAES, WBDES). These implementations aim at protecting the keying material from an adversary who is in possession of the implementation which includes the (masked) key. Even though many years have passed, all scientific white-box AES schemes got eventually broken (to the best of our knowledge), especially since the usage of side-channel attack techniques like Differential Computation Analysis (DCA) [9], Differential Fault Analysis (DFA) [17,14] and/or their recent enhanced variants [2,5,7,31].

However, the business need is stronger, hence this field is still very active, despite relying on software obfuscation techniques and secret design, i.e., violating the Kerckhoffs' principle [18]. Applications of white-box cryptography include—but not limited to—*Digital Rights Management* (DRM) for protected content distribution, *Host Card Emulation* (HCE) on mobile devices for mobile payments, or memory-leakage resilient software; see Bogdanov et al. [6] for a detailed description of each. For an extensive literature research regarding white-box cryptography, we recommend a recent work by Goubin et al. [16].

1.2 Our Contributions

In this paper, we point out the atypically low success rate of the DCA attack against Chow's WBAES presented by Bos et al. [9]. For this phenomenon, we propose a theoretical explanation which identifies a vulnerability of Chow's WBAES to the DCA attack. Based on our results, we further generalize and extend the set of targets that were employed by Bos et al. in their original attack. We also motivate to use our novel targets for a DCA attack against other implementations that use (semi-)linear masking of intermediates.

In the experimental part, we provide a description of our attack toolkit and employed algorithms, and we provide detailed numerical results including timing. Notably, we confirm the vulnerability that was identified during the theoretical analysis. Finally, we study the behavior of false positives in case of a blind attack and derive an optimal number of traces in terms of computational effort.

1.3 Paper Organization

The paper is organized as follows: In Section 2, we give a brief description of Chow's WBAES, we provide a short introduction to side-channel attacks and highlight the usage of DCA in the white-box attack context. We analyze the

DCA attack against Chow's WBAES in Section 3. In Section 4, we describe the practical attack in detail, we support our explanation by an experiment and propose a methodology for practical usage based on a comprehensive testing set. We conclude our work in Section 5.

2 Preliminaries

2.1 Construction of White-Box AES by Chow et al.

One of now classical mathematical approaches how to hide an AES key—in fact all intermediate values as well—in a software implementation is to turn all AES operations into somehow masked lookup tables. Such an approach was introduced in 2002 in a seminal paper by Chow et al. [12]. In their construction, there are four types of lookup tables while the intermediate values are masked using both linear and non-linear random bijections. However, this particular design was mathematically broken two years later by Billet et al. [4].

In the following, we give a high-level description of tables Type II of Chow's WBAES because this is where the attack of our interest will show to be operating. Note that we will be using plain AES without input and output encodings which is technically just an obfuscation layer—we need a plain AES encryption oracle anyways. For further details, we refer to Muir's tutorial [28] which we highly recommend over the original paper by Chow et al.

Lookup tables Type II combine several AES operations together with both linear and non-linear masking, see (1). Description of each operation follows.

$$\text{plaintext} \rightarrow \underbrace{\text{AddKey} \rightarrow \text{SBox} \rightarrow \text{MB} \circ \text{MC} \rightarrow \text{Enc}^{-1}}_{\text{in table Type II}} \rightarrow 1^{\text{st}}\ \text{intermediate} \rightarrow \dots \tag{1}$$

plaintext: The table inputs 1 byte of an AES plaintext block, i.e., the table contains 256 entries.

AddKey**:** This operation XORs respective byte of the (unknown) AES key with the plaintext byte.

SBox**:** This operation is a standard AES SBox, i.e., a 1-byte non-linear bijection.

MB $\circ$ MC**:** This operation is a composition of two 4-byte linear bijections: MC, which stands for standard AES MixColumns, and MB, which is a random linear bijection (hence unknown). Their 4-byte input is split into four 1-byte values, which are handled in separate tables and XORed together in subsequent tables using linearity. Hence this operation inputs 1 byte and outputs 4 bytes (32 bits).

Enc^{-1}**:** This operation is a random 4-bit non-linear bijection, which is applied to each of the eight 4-bit nibbles of the 32-bit input value. Note that it is re-randomized for each nibble and each table while its correct counterpart Enc must be applied at the input to the subsequent lookup table.

1st intermediate: The output value. It can be found by the adversary in the lookup table.

Note 1. Enc bijection is only 4 bits wide, because two such 4-bit nibbles are later XORed together, hence making the input for the subsequent table 8 bits wide. If Enc were 8 bits wide, the subsequent table would need to input 16-bit values, which would make the table very large, however, this approach is used in some white-box implementations.

2.2 Side-Channel Attacks in White-Box Cryptography

Let us briefly recall the principle of side-channel attack and particularly one of its variants upon which Bos et al. [9] built their attack in the white-box context.

Side-Channel Attack (SCA) is a large family of attacks that exploit any weakness of a real-world implementation of a cryptographic algorithm to recover the key (i.e., SCA assumes the gray-box context). SCA was pioneered by Kocher in 1996 in [24] where he focuses on public key cryptography. However, the general idea can be ported to symmetric cryptography as well.

On the one hand, SCA may exploit passively observable measures coming from different sources of information leakage, e.g., power consumption [25], electromagnetic radiation [15], or timing [24]. On the other hand, there exist also active attacks that attempt to alter the computation data or flow and observe corrupted results. The phenomenon of faults in cryptographic algorithms was first addressed by Boneh et al. [8]. For a comprehensive reading we refer to Koç [23, Chapters 13–18].

Differential Power Analysis There are several types of passive SCA's against AES depending on type of the leakage, among them, we will particularly focus on a specific case of *Differential Power Analysis* (DPA). Let us consider that we can measure voltage on a system bus where we expect to capture transfers of AES intermediates. Such records will be referred to as the *traces*. Given a set of plaintexts and respective traces, there exists a moment in time t_0 when certain intermediate value is being transferred over the bus. The goal is to guess a small portion of the key and precompute the expected intermediate value. If the guess is correct, we will find a big correlation between the precomputed intermediates and values across the traces at t_0. Otherwise, no significant correlation shall occur at any position within the traces.

Specifically, we will consider individual bits of the first SBox output as the intermediates, i.e., $t = \mathrm{SBox}(PT[i] \oplus k)[b]$ for i-th byte of a plaintext PT, a key guess k and b-th bit of the SBox output. Such values will be referred to as the *targets* or *hypotheses*, i.e., values that we expect to occur across the traces.

The attack proceeds as follows: we loop all 16 key byte positions i, all 256 guesses on i-th key byte k and all 8 target bit positions b. For each trace, indexed by j, we compute the expected target value as

$$t_j = \mathrm{SBox}(PT_j[i] \oplus k)[b]. \tag{2}$$

Based on the value of $t_j \in \{0, 1\}$, we split the traces into two sets S_0 and S_1, respectively. Note that for the correct key guess, the traces in S_0 are expected to have a low value at t_0 and a high value in S_1, respectively. Therefore we compute absolute difference of means of the two sets $D = \left|\bar{S}_1 - \bar{S}_0\right|$ where $\bar{S}$ denotes a mean trace, i.e., $\bar{S} = \frac{1}{|S|} \sum_{t \in S} t$ using point-wise trace addition. Then, for the correct key guess, there shall emerge a clear peak at time t_0, otherwise the differences of means shall be small and blurry. For each key candidate, we refer to the magnitude of the peak as the *rank* of the candidate. A pseudocode for a derived attack will be given later.

Note 2. In case of a noisy measurement, the peak might be unclear. For this reason, we rank the candidates – the higher rank, the more likely the guess is correct. This might be later used also for brute force key recovery if the initial key guess is incorrect – first we search the candidate bytes with lowest rank.

2.3 Adaptation of SCA to White-Box Attack Context

As introduced by Bos et al. [9], the powerful tools of SCA can be advantageously used for an attack against white-box implementations of cryptographic algorithms. Regarding white-box challenge implementations—de facto encryption oracles—the main benefit of such attacks is that they do not need knowledge of particular implementation, often neither use of reverse engineering, which makes them very universal and easy-to-mount. In this paper, we focus on passive attacks, however, active attacks like *Differential Fault Analysis* (DFA, first introduced against DES [3], later also against obfuscated implementations [17] and in particular against AES [14]) might be adapted as well while making it probably the most powerful attack in the white-box attack context.

In their paper, Bos et al. adapted DPA (as introduced in Section 2.2) for an attack against several white-box implementations; they call this adaptation the *Differential Computation Analysis* (DCA). Instead of physical measurements, they employed instrumentation tools like Valgrind [29] or PIN [26] to capture program-memory interactions, i.e., addresses and/or contents of memory reads and/or writes, referred to as the *memory traces* or *memtraces*.

For all challenges but one attacked by Bos et al., the results showed 100% success rate while using only a couple of memtraces. Neither of these challenges used any form of mathematical obfuscation of intermediates, i.e., the intermediates were directly observable in the memtraces; these challenges relied solely on software obfuscation techniques.

In one particular challenge, Klinec [22] implemented Chow's WBAES, hence the AES intermediates were not directly observable in the memtraces. However, even this implementation got, maybe surprisingly, broken. For their attack, Bos et al. used an augmented set of targets:

$$T_1 = \mathrm{SBox}(PT[i] \oplus k), \tag{3}$$

i.e., the output of the first SBox—the classical targets, cf. (2)—and

$$T_2 = (PT[i] \oplus k)' \tag{4}$$

where $(\cdot)'$ stands for *Rijndael inverse* – a multiplicative inverse in *Rijndael field* $\mathrm{GF}(2^8)$ modulo $x^8 + x^4 + x^3 + x + 1$. The idea behind was motivated by the construction of the original AES SBox:

$$\mathrm{SBox}(X) = A(X') + B, \tag{5}$$

where A is a linear mapping and B a constant byte.

Note 3. T_1 targets are affine mappings of T_2 targets and vice versa, cf. (3), (4) and (5).

Note 4. In the rest of this paper, we will neglect constant bits, i.e., all affine mappings will be considered as linear. Indeed, flipping the target bit only swaps the sets S_0 and S_1 as defined for SCA, hence has no effect on the final result – we are only interested in absolute difference of their means.

Bos et al. employed 500 memtraces with T_1 targets and 2 000 memtraces with T_2 targets. In both cases, they achieved similar success rate – a key byte leaked in about 30% of cases.

3 Analysis of DCA against Chow's White-Box AES

First of all, recall that all of the target bits in T_1 and T_2 can be obtained by a linear mapping of the first SBox output, cf. Note 3, and let us refresh the operations within the first lookup table:

$$\text{plaintext} \to \underbrace{\text{AddKey} \to \text{SBox} \to \text{MB} \circ \text{MC} \to \text{Enc}^{-1}}_{\text{in table Type II}} \to 1^{\text{st}} \text{ intermediate} \to \dots$$

Let us assume that we can get the intermediate value before the final Enc^{-1}, i.e., right after $\mathrm{MB} \circ \mathrm{MC}$. Such a value consists of 32 bits while each bit t' can be computed as a linear mapping of the first SBox output, i.e., $t' = R^T \cdot \mathrm{SBox}(PT[i] \oplus K[i])$ for some vector R. Since MB is a random linear bijection, then R is a random-like non-zero vector. Therefore, in some cases, R might happen to be a standard basis vector, e.g., $(0, 1, 0, \dots, 0)$, or it might be equal to a row of A^{-1}, cf. (5). Note that in such cases, a target from T_1 or T_2, respectively, would perfectly fit t'. However, there are another $255 - 16 = 239$ cases which are not covered by $T_{1,2}$ – let us define a complete set of such targets.

Definition 1. *Let P and K be a plaintext and key byte, respectively. We define the set of all* linear AES-DCA targets *as*

$$T_{lin} = \left\{ R^T \cdot \mathrm{SBox}(P \oplus K) \;\middle|\; R \in \mathrm{GF}(2)^8 \setminus (0, 0, \dots, 0) \right\}. \tag{6}$$

It follows that the set of targets T_{lin} fully covers the intermediates before the final Enc^{-1}, however, in the real implementation, Enc^{-1} is employed as well. It follows that Enc actually poses the only protection against our linear AES-DCA

targets. As per the results of Bos et al., some targets leak the key byte anyways, hence let us focus on the (non-)linearity properties of Enc.

In the design of Chow's WBAES, Enc is defined to be a random 4-bit bijection while posing the only non-linear (confusion) element. They provide the following argumentation: "Ideally for security, we would explicitly avoid linear transformations. But randomly choosing bijections, essentially all will be non-linear: ... less than 0.000 002% are affine." Hence they do not encourage for any non-linearity check although it is a widely studied topic for regular ciphers by methods of *linear cryptanalysis*, introduced by Matsui et al. [27].

On the one hand, the ratio of fully linear mappings is indeed extremely low, on the other hand, DCA can exploit the intermediates even when there occurs only one bit in Enc output that is linear in its input. Note that there are lot more such 4-bit bijections – indeed, there are $2 \cdot 4 \cdot (2^4 - 1) \cdot 8! \cdot 8!$ of them among 16! bijections which is almost 1%. Since there are several Enc instantiations for each key byte, 1% chance is very much non-negligible. Furthermore, DCA is based on a physical SCA, i.e., it is designed to handle errors, cf. Note 2. For this reason, even such Enc bijections that are linear in single output bit on majority of inputs pose a weakness. There are obviously much more than 1% of such bijections making the protection vulnerable to DCA with our T_{lin} targets.

Since our linear AES-DCA targets address *all* linear transformations of the first-round AES intermediates, they can be advantageously applied to other schemes that employ linear protection (or semilinear, like Chow's WBAES). Note that a random linear bijection is a handy masking technique since it can easily combine several bytes together – thanks to its linearity. See, e.g., a recent report by Goubin et al. [16] recovering the hardest challenge submitted to WhibOx 2017 Contest Workshop [13] – the Adoring Poitras challenge[1]. In their work, they introduce *linear decoding analysis* which also correctly assumes linear encoding of intermediates.

4 Practical Attack & Results

First, we describe the DCA bitwise attack in detail and provide an overview of the whole attacking procedure. Next, as the main goal of our experiments, we confirm our hypothesis about leakage as introduced in Section 3, i.e., leakage from the first set of tables Type II. Then we focus on a scenario with unknown key while inspecting properties of false positives. Finally, we suggest a methodology to estimate an optimal number of traces for this type of attack and evaluate the optimum for Chow's WBAES. Note that we performed all experiments on a single core of Intel Core i5-7600K processor @ 4.1GHz, i.e., all execution times are with respect to this hardware.

[1] Available at https://whibox-contest.github.io/show/candidate/777. Accessed: August, 2019.

4.1 Bitwise DCA

The most demanding part of our attack is the bitwise DCA/DPA attack as outlined in Section 2.2 – lots of memtrace-, i.e., vector-, additions are performed. We implemented that part of the attack in C++ [19]. We summarize this attack in Algorithm 1 where P, Trc, trg and B denote the arrays of plaintexts, respective traces represented in bits, target tables as per Definition 1, and attacked byte number (i.e., $1 \dots 16$), respectively. The output array dif is a list of key candidates sorted by their rank, for each target bit.

Algorithm 1 Bitwise DCA/DPA attack.

function BitwiseDCA(P, Trc, trg, B)
  // a 256-tuple of 8-tuples of triples: key guess, rank and leakage position
  $dif = \Big(\big((\texttt{0x00}, 0.0, 0), \dots, (\texttt{0x00}, 0.0, 0)\big), \dots, \big((\texttt{0xff}, 0.0, 0), \dots, (\texttt{0xff}, 0.0, 0)\big)\Big)$
  for $kg = \texttt{0x00} \dots \texttt{0xff}$ **do** // key guess
    $absdif = \big((0.0, \dots, 0.0), \dots\big)$ // an 8-tuple of vectors of trace bit-size
    $mean_{0,1} = \big((0.0, \dots, 0.0), \dots\big)$ // both an 8-tuple of vectors of trace bit-size
    $num_{0,1} = (0, 0, 0, 0, 0, 0, 0, 0)$ // both an 8-tuple
    for $i = 1 \dots |P|$ **do**
      $p = P[i]$, $trc = Trc[i]$
      $hyp = trg[p[B] \oplus kg]$ // hypothesis, i.e., "SBox" output, cf. (2)
      for $b = 1 \dots 8$ **do** // target bit
        if $hyp[b] == 0$ **then**
          $mean_0[b] \mathrel{+}= trc$ // most demanding
          $num_0[b] \mathrel{+}= 1$
        else
          $mean_1[b] \mathrel{+}= trc$ // most demanding
          $num_1[b] \mathrel{+}= 1$
    for $b = 1 \dots 8$ **do**
      if $num_0[b] \neq 0$ **then** $mean_0[b] \mathrel{/}= num_0[b]$
      if $num_1[b] \neq 0$ **then** $mean_1[b] \mathrel{/}= num_1[b]$
      $absdif[b] = \big|mean_1[b] - mean_0[b]\big|$
      // maximal absolute difference and its position is found & saved
      $dif[kg][b] = \big(kg, \max(absdif[b]), \arg\max(absdif[b])\big)$
      sort $dif[\cdot][b]$ by rank
  return dif

4.2 Steps of the Attack

The practical implementation of our attack consists of several tools and follows the steps described in Algorithm 2.

Algorithm 2 Steps of the practical attack.

1: acquire memtraces
2: filter constant values from memtraces

3: generate memtrace preview & identify leakage range
4: **if** found leakage range **then go to** 6
5: attack some byte (possibly first and/or last) & identify leakage range
6: crop traces to leakage range
7: run full attack, process & display results

All the tools are written in Ruby and published in our `White-Box-DPA-Processing` toolkit [21]. Next we describe each step and/or component of the toolkit.

Trace Acquisition Our acquisition tool generates random AES plaintexts, feeds them to the target WBAES implementation while acquiring the memtrace. For this purpose, we employ Intel PIN [1] with our custom memory tracing tools [20]. There are total four tools that enable to acquire contents or addresses of memory reads or writes, respectively. As a reasonable initial number of traces, for an unprotected implementation, even 25 is sufficient, for an implementation with a semi-linear protection similar to Chow's WBAES, lower hundreds of traces are needed[2]. Acquisition of 200 traces of Klinec's implementation took us roughly 4 minutes. If the number of traces shows to be insufficient during the attack, our acquisition tool enables to acquire additional traces. Note that we acquired contents of memory reads, i.e., we expect that there occur values from those aforementioned white-box lookup tables.

Last but not least, it is highly important to have the traces well aligned, hence it is recommended to turn off Address Space Layout Randomization (ASLR). On Unix-like systems, this can be done by the command

```
$ setarch `uname -m` -R /bin/bash
```

Filtering Constant Regions In the memtraces, there occur several regions which are identical across all traces, hence carry no information for the attack. Our acquisition tool automatically creates a filtering mask based on a couple of traces (by default 30 traces) and filters these regions out. As a result, there remains no constant value across traces while the traces remain aligned. For 200 of Klinec's traces, this step took us about 15 seconds.

Identify the Leakage Range from a Memtrace Preview Next, our tool creates a memtrace preview: the x-axis represents the address space (partial and usually "zoomed out" to fit reasonable dimensions), the y-axis represents the execution time (top-bottom), memory writes are represented by a red pixel; see Figure 1 (the green marker will be explained later). If we can clearly recognize where SBoxes of the first AES round take place, we can skip the next step and continue to cropping the traces to the leakage range.

[2] Later we will discuss optimal number of traces for this type of WBAES and recommend 200 traces.

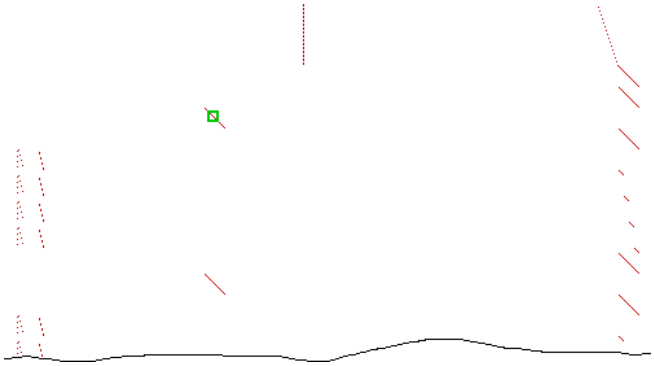

Fig. 1. Partial memtrace of memory writes of a naive AES implementation with 7^{th} byte leakage position emphasized in green. The memtrace is cropped within the 2^{nd} AES round.

Initial Attack to Identify the Leakage Range In case we are not sure where exactly the leakage takes place, we recommend to attack single key byte (the first and the last byte could possibly show the beginning and the end of that range, respectively) and use our marker tool to emphasize the exact place within the memory trace; cf. Figure 1. Details to the attacking procedure are given in Section 4.2. For 200 of Klinec's traces, attacking single key byte with full traces took us less than 2 minutes.

Crop Traces to the Leakage Range Once we identified the leakage range, we can further crop the traces by specifying the address and row intervals in our cropping tool. This step is the most important one for the overall attack acceleration. For Klinec's traces, we cropped the traces from originally 2 456 entries to 197 entries, i.e., we reduced the traces as well as the attack complexity by a factor of 12.

Once we decide to repeat the attack on the same implementation, only with a different key, we can make use of the exact leakage position, hence making the attack yet faster and ready for use with automated tools [10].

Full Attack At this point, everything is ready for the full attack. First, the bitwise DCA attack is performed as per Algorithm 1 and detailed results are saved, i.e., for each key byte, each attack target, each target bit, key candidates are sorted by their rank together with the position of the maximum (i.e., the leakage index). Second, these results are processed: for each such a piece of result, relative gap between the rank of the two top candidates is computed and used as a measure of candidate quality. With 200 of filtered Klinec's traces, the first step took us roughly 2 minutes for all 16 key bytes, the second step cca 20 seconds. In Table 1, we show the results of the attack with Rijndael inverse taken as the target.

Note 5. In our results, we recognize two kinds of best candidates based on the gap: if the gap is greater than 10%, it is referred to as the *strong candidate*, otherwise it is referred to as the *weak candidate*. Since we know the key, we can identify the position of the correct key byte within all guesses. In order to recognize a successful attack, we emphasize it in case it occupies the top position: in black ■ if it is a strong candidate, and in gray ■ for a weak candidate.

4.3 Confirmation of the Leakage Hypothesis about Chow's WBAES

In order to confirm our hypothesis on the leakage point in Chow's WBAES as introduced in Section 3, we decided to perform two experiments:

1. reproduce the attack by Bos et al. which takes the values from memtraces,
2. modify Klinec's implementation to dump the intermediates coming from the first set of tables Type II—this is where leakage in the original attack is expected to take place—and use them directly instead of memtraces.

We performed both attacks with identical setup, the only difference was in the trace data origin – it either came from memtraces, or from a direct manual dump. To our satisfaction, both results were *perfectly identical*. This confirms our hypothesis that the vulnerable intermediates are those identified in Section 3, i.e., the output of the first set of tables Type II. In the following experiments, we used direct dumps from the modified implementation instead of memtraces for performance reasons.

4.4 Blind Attack on Chow's WBAES

In a real-world scenario where the key is unknown, we do not know at which position the correct key byte is within the list of candidates. In order to suggest a methodology to recognize the correct candidate, we need further observations about how both correct and incorrect candidates behave. For this purpose, we ran a set of attacks: we created 8 instantiations of the white-box tables, captured 500 traces and used all 255 targets as per Definition 1 – this makes altogether 32 640 attacks on individual key byte which took us almost 50 minutes.

The most significant problem is that there often occurs a strong, yet incorrect top candidate (i.e., a *false positive*), cf. Table 1 (e.g., 10^{th} key byte and 5^{th} target bit with almost 20% gap). In our overall results, 22% of top candidates were strong and correct with an average and maximal gap of 38% and 76%, respectively. However, there were also 8% of (strong) false positives with an average and maximal gap of 14% and 35%, respectively. It follows that a simple rule using single gap threshold would work bad. On the other hand, we observed that the same false positive does not appear to repeat very much across the 255 targets for given key byte: the average number of repetitions of the best false positive (for given key byte) was 1.75, the global maximum was only 3. A summary of results will be given after we introduce another quantity in Section 4.5.

Key byte	Target bit															
	1. bit		2. bit		3. bit		4. bit		5. bit		6. bit		7. bit		8. bit	
1.	14.5	148	19.9	■	7.6	■	0.3	241	0.5	81	4.7	226	3.7	3	17.0	■
2.	4.5	164	12.9	218	3.3	187	33.7	■	7.3	54	0.3	117	0.2	167	30.4	■
3.	0.7	183	0.2	205	0.2	4	18.8	■	2.5	192	4.2	115	1.4	184	8.5	72
4.	2.3	91	1.5	■	12.9	163	2.2	59	2.5	68	0.5	152	0.2	219	2.6	162
5.	1.9	15	2.8	68	5.7	60	1.1	153	0.2	42	5.1	161	0.0	35	0.3	127
6.	30.2	■	9.7	210	7.6	101	6.5	135	1.0	2	35.6	■	28.0	■	0.2	58
7.	2.7	7	6.0	57	0.7	179	6.6	241	1.8	137	5.2	1	2.4	123	6.5	198
8.	50.8	■	3.3	211	1.4	198	2.1	251	1.7	155	2.4	255	35.2	■	4.6	176
9.	18.5	181	5.9	111	0.6	52	0.3	235	3.9	86	5.0	154	33.2	■	1.3	121
10.	0.8	38	6.2	152	20.5	■	26.0	■	19.3	111	0.9	137	27.8	■	34.5	■
11.	4.5	141	17.3	■	6.8	35	10.2	176	2.9	137	8.8	66	3.2	79	1.3	136
12.	24.1	■	4.0	206	5.6	113	2.5	213	5.6	69	2.3	210	34.7	■	2.1	77
13.	4.2	24	5.9	246	2.8	244	0.2	15	30.4	■	3.3	■	9.1	125	11.1	34
14.	49.7	■	1.7	248	4.9	33	20.5	■	7.6	98	16.7	252	14.7	■	29.9	■
15.	6.5	139	0.6	126	6.3	16	5.4	37	2.3	64	3.6	1	3.6	■	5.8	100
16.	17.5	157	40.7	■	5.8	105	0.1	37	23.9	■	14.2	184	0.1	211	2.1	17

Table 1. DCA using 200 memtraces and 8 bits of Rijndael inverse as targets. For each key byte and each target bit, percentual gap of the best candidate and position of the correct key byte are given. Note that the position ranges from 0 to 255 while 0 is replaced with ■ or ■ for a strong or a weak candidate, respectively; cf. Note 5.

Suggested Strategy We suggest the following strategy: for each key byte, keep looping the 255 targets until any strong candidate exceeds a cumulative bound of 50% with its gaps. Note that if such a candidate were a false positive, it would need about 4 average gaps of a false positive to exceed the bound, which is still more than ever observed number of repetitions of a false positive. Even if bad things happen in a rare case, we can possibly increase the cumulative bound or brute-force the least confident key byte(s). Note that a similar strategy can be derived to other schemes than Chow's WBAES.

4.5 Optimal Number of Traces

In their attacks, Bos et al. used 500 and 2 000 traces to attack Chow's WBAES, let us now have a look at results of the attack with much less traces, namely 100, 200, 300 and 500 traces. For each number of traces, we attacked all of our 8 instantiations and observed ratios of strong candidates (both correct and incorrect) together with their average gap; see results in Table 2. Note that the number of repetitions of false positives remained up to three.

With less traces, the number of correct candidates and their average gap decrease, i.e., we need to use more targets in order to reach the cumulative bound, and vice versa. Hence our goal is to give a reasonable estimate on the optimal number of traces in terms of computational effort. For this purpose, we introduce the *reduced cost of gap* as

$$C(n, s, g) = \frac{n}{s \cdot g}, \tag{7}$$

where n stands for the number of traces, s for the average success rate and g for the average gap of a strong candidate. Note that this quantity corresponds with the average computational effort: indeed, the more traces, the more effort, the better success rate or the bigger average gap, the less effort. According to Table 2, the lowest value of reduced cost of gap was achieved for 200 traces, therefore we suggest to use 200 traces in this scenario.

Traces	100	200	300	500
Correct candidates	6.5%	17%	19%	22%
Average gap of correct candidates	22%	29%	34%	38%
False positives	2.3%	7.5%	8.2%	8.3%
Average gap of false positives	9.8%	14%	14%	14%
Reduced cost of gap	7 000	4 100	4 600	6 000

Table 2. Ratios and average gaps of correct and incorrect strong candidates, respectively, and reduced cost of gap, for different numbers of traces.

5 Conclusions

After a brief overview of white-box cryptography and Chow's WBAES, we recalled the idea of SCA usage in the white-box context, pioneered by Bos et al. We highlighted the abnormal behavior of their attack against Chow's WBAES, for which we proposed a theoretical explanation. The problem of Chow's WBAES has shown to be linearity of the Enc bijection which was intended to be non-linear. Although Chow et al. provided a reasoning about its non-linearity, it is not sufficient against DCA anymore, in particular when using our extended set of linear AES-DCA targets. We motivated the use of our targets against other implementations that use (semi-)linear masking of intermediates.

In the experimental part, we described our tools and, in particular, we confirmed our hypothesis by a comparison of two differently obtained sets of detailed results. Next, we focused on the behavior of false positives in case of a blind attack and suggested a strategy for this purpose. Finally, we derived an optimal number of traces for this kind of attack in terms of average computational cost to make the attack effective. With resulting 200 of filtered traces of Klinec's implementation, we ran the attack in less than two and a half minutes on our hardware.

References

1. Pin 3.11 User Guide. https://software.intel.com/sites/landingpage/pintool/docs/97998/Pin/html/. Accessed: August, 2019
2. Banik, S., Bogdanov, A., Isobe, T., Jepsen, M.: Analysis of software countermeasures for whitebox encryption. IACR Transactions on Symmetric Cryptology pp. 307–328 (2017)
3. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Annual international cryptology conference, pp. 513–525. Springer (1997)
4. Billet, O., Gilbert, H., Ech-Chatbi, C.: Cryptanalysis of a white box AES implementation. In: Selected Areas in Cryptography, pp. 227–240. Springer (2004)
5. Bock, E.A., Brzuska, C., Michiels, W., Treff, A.: On the ineffectiveness of internal encodings-revisiting the dca attack on white-box cryptography. In: International Conference on Applied Cryptography and Network Security, pp. 103–120. Springer (2018)
6. Bogdanov, A., Isobe, T., Tischhauser, E.: Towards practical whitebox cryptography: Optimizing efficiency and space hardness. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 126–158. Springer (2016)
7. Bogdanov, A., Wang, J.M., Vejre, S.: Higher-order dca against standard side-channel countermeasures. In: Constructive Side-Channel Analysis and Secure Design: 10th International Workshop, vol. 11421, p. 118. Springer (2019)
8. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: International conference on the theory and applications of cryptographic techniques, pp. 37–51. Springer (1997)
9. Bos, J., Hubain, C., Michiels, W., Teuwen, P.: Differential Computation Analysis: Hiding your White-Box Designs is Not Enough. In: International Conference on Cryptographic Hardware and Embedded Systems, pp. 215–236. Springer (2016)

10. Breunesse, C.B., Kizhvatov, I., Muijrers, R., Spruyt, A.: Towards fully automated analysis of whiteboxes: Perfect dimensionality reduction for perfect leakage. IACR Cryptology ePrint Archive **2018**, 95 (2018)
11. Chow, S., Eisen, P., Johnson, H., Van Oorschot, P.: A white-box DES implementation for DRM applications. In: Digital Rights Management, pp. 1–15. Springer (2002)
12. Chow, S., Eisen, P., Johnson, H., Van Oorschot, P.: White-box cryptography and an AES implementation. In: Selected Areas in Cryptography, pp. 250–270. Springer (2002)
13. CryptoExperts: WhibOx 2017. Online: https://whibox-contest.github.io/2017/ (2017). Accessed: August, 2019
14. Dusart, P., Letourneux, G., Vivolo, O.: Differential fault analysis on AES. In: International Conference on Applied Cryptography and Network Security, pp. 293–306. Springer (2003)
15. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Cryptographic Hardware and Embedded SystemsCHES 2001, pp. 251–261. Springer (2001)
16. Goubin, L., Paillier, P., Rivain, M., Wang, J.: How to Reveal the Secrets of an Obscure White-Box Implementation. Tech. rep., Cryptology ePrint Archive, Report 2018/098. https://eprint.iacr.org/2018/098 (2018)
17. Jacob, M., Boneh, D., Felten, E.: Attacking an obfuscated cipher by injecting faults. In: Digital Rights Management, pp. 16–31. Springer (2002)
18. Kerckhoffs, A.: La Cryptographie Militaire. Journal des sciences militaires **9**, 538 (1883)
19. Klemsa, J.: Bitwise DPA. Git repository. https://github.com/fakub/BitwiseDPA
20. Klemsa, J.: Memory Tracing Tools for Intel PIN. Git repository. https://github.com/fakub/MemoryTracingTools
21. Klemsa, J.: White-Box-DPA-Processing toolkit. Git repository. https://github.com/fakub/White-Box-DPA-Processing
22. Klinec, D.: White-box attack resistant cryptography (2013)
23. Koç, Ç.: Cryptographic Engineering. Springer US (2008)
24. Kocher, P.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Annual International Cryptology Conference, pp. 104–113. Springer (1996)
25. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual International Cryptology Conference, pp. 388–397. Springer (1999)
26. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM Sigplan Notices, vol. 40, pp. 190–200. ACM (2005)
27. Matsui, M.: Linear cryptanalysis method for DES cipher. In: Advances in CryptologyEUROCRYPT93, pp. 386–397. Springer (1993)
28. Muir, J.A.: A Tutorial on White-box AES. Tech. rep., Cryptology ePrint Archive, Report 2013/104. http://eprint.iacr.org/2013/104 (2013)
29. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan notices, vol. 42, pp. 89–100. ACM (2007)
30. PUB, N.F.: 197: Advanced Encryption Standard (AES). Federal Information Processing Standards Publication **197**, 441–0311 (2001)
31. Rivain, M., Wang, J.: Analysis and improvement of differential computation attacks against internally-encoded white-box implementations. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 225–255 (2019)