IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2017, No. 1, pp. 307–328. DOI:10.13154/tosc.v2017.i1.307-328

# Analysis of Software Countermeasures for Whitebox Encryption

Subhadeep Banik[1], Andrey Bogdanov[2], Takanori Isobe[3] and Martin Bjerregaard Jepsen[2]

[1] Temasek Labs, Nanyang Technological University, Singapore, Singapore
bsubhadeep@ntu.edu.sg

[2] Department of Applied Mathematics and Computer Science (DTU Compute), Technical University of Denmark, Kongens Lyngby, Denmark
anbog@dtu.dk,martin@jepsen.no

[3] Sony Corporation, Tokyo, Japan
takanori.isobe@jp.sony.com

**Abstract.** Whitebox cryptography aims to ensure the security of cryptographic algorithms in the whitebox model where the adversary has full access to the execution environment. To attain security in this setting is a challenging problem: Indeed, all published whitebox implementations of standard symmetric-key algorithms such as AES to date have been practically broken. However, as far as we know, no whitebox implementation in real-world products has suffered from a key recovery attack. This is due to the fact that commercial products deploy additional software protection mechanisms on top of the whitebox implementation. This makes practical attacks much less feasible in real-world applications.

There are numerous software protection mechanisms which protect against standard whitebox attacks. One such technique is control flow obfuscation which randomizes the order of table lookups for each execution of the whitebox encryption module. Another technique is randomizing the locations of the various Look up tables (LUTs) in the memory address space. In this paper we investigate the effectiveness of these countermeasures against two attack paradigms. The first known as Differential Computational Analysis (DCA) attack was developed by Bos, Hubain, Michiels and Teuwen in CHES 2016. The attack passively collects software execution traces for several plaintext encryptions and uses the collected data to perform an analysis similar to the well known differential power attacks (DPA) to recover the secret key. Since the software execution traces contain time demarcated physical addresses of memory locations being read/written into, they essentially leak the values of the inputs to the various LUTs accessed during the whitebox encryption operation, which as it turns out leaks sufficient information to perform the power attack. We found that if in addition to control flow obfuscation, one were to randomize the locations of the LUTs in the memory, then it is very difficult to perform the DCA on the resultant system using such table inputs and extract the secret key in reasonable time. As an alternative, we investigate the version of the DCA attack which uses the outputs of the tables instead of the inputs to mount the power analysis attack. This modified DCA is able to extract the secret key from the flow obfuscated and location randomized versions of several whitebox binaries available in crypto literature.

We develop another attack called the Zero Difference Enumeration (ZDE) attack. The attack records software traces for several pairs of strategically selected plaintexts and performs a simple statistical test on the effective difference of the traces to extract the secret key. We show that ZDE is able to recover the keys of whitebox systems. Finally we propose a new countermeasure for protecting whitebox binaries based on insertion of random delays which aims to make both the ZDE and DCA attacks

Received: 2016-11-23, Accepted: 2017-01-23, Published: 2017-03-08

practically difficult by adding random noise in the information leaked to the attacker.

**Keywords:** Whitebox Cryptography, differential computational analysis, zero enumeration attack. control randomization, dummy operation

## 1 Introduction

Whitebox cryptography was introduced by Chow et al. in 2002 [CEJvO02a] as a technique to protect software implementations of cryptographic algorithms in untrusted environments. An increasing number of applications are emerging that require substantial security in purely software environments, e.g. set-top boxes, PCs, tablets, smartphones, HCE, digital rights management (DRM) systems, or client software running in the cloud. The major goal of whitebox cryptography is to protect the confidentiality of secret keys when the adversary has *full* access to the execution environment with the aid of a decompiler, debugger tools and dynamic binary analysis tools, e.g. IDA Pro, IL DASM, Valgrind and PIN. In the wake of this seminal paper, several further variants of whitebox implementations for AES were proposed [BCD06, XL09, Kar10, LN05].

The whitebox setting assumes that the executable of the encryption module would run on untrusted and often malicious platforms over which an adversary has total control. As such, potentially every intermediate value computed by the executable is liable to be leaked and would provide the adversary a channel of additional information for cryptanalysis. Thus, for a given scheme to be whitebox secure, it has to withstand key recovery attack, even under the assumption that every temporary/intermediate value it calculates is available to the adversary. Although, all published whitebox solutions for AES to date have been *practically* broken [BGE04, WMGP07, MWP10, MRP12, LRM$^+$13, Mul14], due to increasing demand, a large number of companies still sell and deploy whitebox AES products and solutions. The reason for this is that the security model of whitebox cryptography is too strong in many real-world applications. In practice white box cryptography is often only a small part of the software protection mechanism, where it is used in conjunction with additional protection techniques like [ARX14, whi15, Mic15]

- Control flow obfuscation: For each execution, the path of the computation is randomized to confuse and force an adversary to reverse engineer a complex node graph, including a runtime randomization and dummy operations.
- Tamper resistance: Integrity protection to ensure that the application code and read-only data are not modified.
- Device binding: Binding the code/binary to the current device, and therefore preventing it from executing in the adversaries environments.
- Anti-debug protection: Detecting whether the binary is executed under a debugger, and performing counteractions.

The device binding and the anti-debug protection hamper the usage of disassembler/debugging and binary analysis tools, and it defends the adversary from the full control of the execution environments and lifting the code/binary. The control flow randomization and the tamper resistance prevent the adversary from performing attacks which may require finding a *correct* byte position and *overwriting* it. To bypass these protections, considerable reverse engineering efforts by means of analysis tools with high skills and experience are required on the adversary's side [Wys12].

A number of other practical key recovery attacks against whitebox AES implementations have been proposed [BGE04, MWP10, MRP12, LRM$^+$13, Mul14] in which, by decomposing the obfuscated table, the secret key is derived with practical time complexity. These attacks require read/*write* access to *correct* internal states during the execution to exploit the relation between input and output of the target obfuscated table. Sanfelix et al. propose a differential fault analysis on whitebox AES and DES [SMdH15] in which faults are injected into the cryptographic process at the *correct* locations within the algorithm. This attack also requires write access to the specific state where faults are injected. The integrity protection and the control flow randomization make these attacks infeasible in the real-world products with a barrier to the full control by the device binding and the anti-debug protection. In fact, until now, no white-box implementation in a real-world product has suffered from a key recovery attack. There is a clear gap between theory and practice.

In summary, it is still questionable if previous attacks work for the real-world products where additional countermeasures are deployed and the adversary has limited control of the environment.

## 1.1 Contributions and Organization

In this paper we investigate the effectiveness of countermeasures against side-channel attacks on whitebox implementations (Section 4). These countermeasures are similar to some of the software protection mechanisms that protect real-world whitebox systems[ARX14, whi15, Mic15]. We perform and verify the DCA proposed by Bos et. al., which uses techniques from power side-channel attacks on hardware to recover the key of software systems. Since the attack is feasible by passively collecting software execution traces for several plaintext encryptions, integrity check protections is useless against this attack unlike previous attacks. The techniques are verified on a challenge[1] released for CHES 2016, in several variants that also include protection mechanisms (Section 6):

- Control flow obfuscation: We created an implementation of the CHES 2016 code that randomly shuffles the order of the instructions in each run without changing the result of the encryption.
- Randomization of table locations: We used a PRNG to randomly disperse lookup tables in the memory address space.

We find that if one were to randomize the locations of the whitebox tables in the memory, then it is very difficult to perform the DCA on the resultant system and extract the secret key in reasonable time. As an alternative, we investigate the version of the DCA attack which uses the outputs of the tables instead of the inputs to mount the power analysis attack. This modified DCA is able to extract the secret key from the flow obfuscated and location randomized versions of the whitebox binaries. We propose a new countermeasure for protecting whitebox binaries based on insertion of random dummy operations, which aims to make DCA attacks practically difficult by adding random noise in the information leaked to the attacker.

Finally we propose and implement another attack paradigm called the Zero Difference Enumeration Attack (ZDE, Section 5). We show that the ZDE attack can recover the key of whitebox systems, by applying it to publicly available whitebox challenges:

- The Hack.lu challenge of 2009.
- A public implementation of the Chow whitebox available on the Internet[2].

We then show that the ZDE attack is applicable to any combination of the countermeasures listed above. Finally we propose a new countermeasure for protecting whitebox

[1]Found at `https://ctf.newae.com/flags/`
[2]at `https://github.com/ph4r05/Whitebox-crypto-AES`

binaries based on the "random disarrangement of time" countermeasure, originally proposed in [Man04] to counteract power attacks against symmetric ciphers. This measure essentially aims to make both the ZDE and DCA attacks practically difficult by adding a random number of dummy operations before performing table accesses.

The rest of the paper is organized in the following manner. In Section 2, we give a brief description of the AES-128 block cipher and Chow's whitebox scheme. In Section 3, we describe the Differential Computation Attacks (DCA) introduced in [BHMT16] and further introduce the notions of address and value based attacks, (ADCA and VDCA respectively). In Section 4, we describe in brief the software countermeasures we have considered in this paper. In Section 5, we introduce the concept of Zero Difference Enumeration attacks. All experimental results are included in Section 6. Section 7 concludes the paper.

## 2 Whitebox AES Implementations

This section first describes the algorithm of AES, and fixes notations of the whitebox AES that we will use throughout the paper.

### 2.1 Description of AES

AES-128 is a block cipher with a 128-bit internal state and a 128-bit key $K$. The internal state $S$ and the key are represented by two $4 \times 4$ byte matrices. For example, the internal state after round $r$ is represented as follows (where $s_{i,j}^{(r)}$ are byte values).

$$S^{(r)} = \begin{bmatrix} s_{0,0}^{(r)} & s_{0,1}^{(r)} & s_{0,2}^{(r)} & s_{0,3}^{(r)} \\ s_{1,0}^{(r)} & s_{1,1}^{(r)} & s_{1,2}^{(r)} & s_{1,3}^{(r)} \\ s_{2,0}^{(r)} & s_{2,1}^{(r)} & s_{2,2}^{(r)} & s_{2,3}^{(r)} \\ s_{3,0}^{(r)} & s_{3,1}^{(r)} & s_{3,2}^{(r)} & s_{3,3}^{(r)} \end{bmatrix}$$

AES consists of a data processing part and a key schedule. The data processing part adopts a substitution-permutation network whose round function consists of four layers: `SubBytes`, `ShiftRows`, `Mixcolumns` and `AddRoundKey`. Subkeys are generated by a key schedule. 128-bit subkeys are denoted as $K^{(r)}$, $1 \leq r \leq 11$, and each byte is indexed by $i$ and $j$ as $k_{i,j}^{(r)}$ in the same manner of the state $S$. For the details of the key schedule of AES, we refer to [oST01].

`SubBytes` is a set of sixteen 8-bit nonlinear transformation $Sb$: $\{0,1\}^8 \to \{0,1\}^8$ applying a 8-bit S-box to each cell. `ShiftRow` rotates four bytes in the $a$-th row by $a$ positions to the left. `MixColumns` is a linear transformation $MC$: $\{\{0,1\}^8\}^4 \to \{\{0,1\}^8\}^4$ which multiplies each column by a $4 \times 4$ diffusion matrix with branch number 5. Let the coefficients of the matrix be denoted by $mc_{i,j}$, $0 \leq i,\ j \leq 3$. `AddRoundKey` adds a 128-bit subkey $K^{(r)}$ to a 128-bit state by an XOR operation. Note that `AddRoundKey` is also performed before the first round as whitening and that `MixColumns` is omitted in the last round.

We define a 32-bit function `Subround` as follows. Let $\hat{K}^{(r)}$ and $\hat{S}^{(r)}$ denote the key and the state obtained after applying `ShiftRows` to $K^{(r)}$ and $S^{(r)}$, respectively. $\hat{k}_{i,j}^{(r)}$, $\hat{s}_{i,j}^{(r)}$ are respectively each byte of $\hat{K}^{(r)}$ and $\hat{S}^{(r)}$. $T_{i,j}^{(r)}$: $\{0,1\}^8 \to \{0,1\}^8$ is defined as $T_{i,j}^{(r)}(x) = Sb(x \oplus \hat{k}_{i,j}^{(r)})$.

**Definition 1. (Subround)** $\mathsf{Subround}_j^{(r)}$*:* $\{\{0,1\}^8\}^4 \to \{\{0,1\}^8\}^4$ *for* $1 \leq r \leq R$ *and* $0 \leq j \leq 3$ *is defined as* $(y_0, y_1, y_2, y_3) = \mathsf{Subround}_j^{(r)}(x_0, x_1, x_2, x_3)$ *(note* $R = 10$ *for AES-128) with*

- $1 \le r \le R-1$

$$\begin{aligned} y_i &= (mc_{i,0} \otimes T^{(r)}_{0,j}(x_0)) \oplus (mc_{i,1} \otimes T^{(r)}_{1,j}(x_1)) \\ &\quad \oplus (mc_{i,2} \otimes T^{(r)}_{2,j}(x_2)) \oplus (mc_{i,3} \otimes T^{(r)}_{3,j}(x_3)), \quad 0 \le i \le 3. \end{aligned}$$

- $r = R$

$$y_i = (k^{(R+1)}_{i,j} \oplus T^{(r)}_{i,j}(x_i)), 0 \le i \le 3.$$

Here $\otimes$ denotes finite field multiplication in $GF(2^8)$ represented in the polynomial basis $x^8 + x^4 + x^3 + x + 1$. As an intended consequence of the design of AES, the structure of the data processing function can be rewritten such that each round uses four Subround modules. The data processing par of the AES-128 is described as Algorithm 1.

**Algorithm 1** AES-128 implementation (data processing part)

```
$S^{(0)} \leftarrow$ Plaintext
/* Round 1 to 10 */
for $r = 1$ to 10 do
    $\hat{S}^{(r-1)} \leftarrow$ ShiftRow($S^{(r-1)}$)
    for $j = 0$ to 3 do
        $(s^{(r)}_{0,j}, s^{(r)}_{1,j}, s^{(r)}_{2,j}, s^{(r)}_{3,j}) \leftarrow \mathsf{Subround}^{(r)}_j(\hat{s}^{(r-1)}_{0,j}, \hat{s}^{(r-1)}_{1,j}, \hat{s}^{(r-1)}_{2,j}, \hat{s}^{(r-1)}_{3,j})$
    end for
end for
Ciphertext $\leftarrow S^{(10)}$
```

## 2.2 Chow et al's Whitebox Implementation

Whitebox implementations of AES were first proposed by Chow et al. in [CEJvO02b]. The basic idea is to combine the key and the algorithm by partial evaluation of parts of the cipher. The approach was to find a representation of the algorithm as a network of look-ups in randomized and key-dependent tables. These tables are protected with secret mappings, with the aim of making extracting the key from them difficult. A trivial (but not practical) example of this is to simply precompute a full mapping from all plaintexts to their corresponding ciphertexts. Encryption is then a single table lookup on the plaintext, and recovering the key should be as hard as attacking the cipher in a black-box scenario. However such a lookup table will need at least $2^{128}$ entries (one for every plaintext) and hence it is infeasible.

The main idea in the paper by Chow et. al. is a method for constructing a set of randomized key-dependent tables that have a practical size, allowing whitebox AES to be implemented in practice. For a detailed description of Chow's whitebox we refer to [CEJvO02b] and an excellent tutorial available at [Mui13]. However for the completeness of the paper we provide a small structural description.

In the case of the Chow et. al. whitebox design, the principal idea is to create an encoded subround function EnSubround. Specifically, the whitebox implementation adds nonlinear bijection functions $P^{(r)}_{i,j}$ and $Q^{(r)}_{i,j}$: $\{0,1\}^8 \to \{0,1\}^8$, $0 \le i \le 3$ before and after $\mathsf{Subround}^{(r)}_j$ as shown in Fig. 1. The first and last rounds disregard input and output encodings, respectively. Define an encoded subround $\mathsf{EnSubround}^{(r)}_j$ as follows.

**Definition 2. (Encoded Subround)**

*$\mathsf{EnSubround}^{(r)}_j$: $\{\{0,1\}^8\}^4 \to \{\{0,1\}^8\}^4$ for $2 \le r \le 9$ and $0 \le j \le 3$ is defined as $(y_0, y_1, y_2, y_3) = \mathsf{EnSubround}^{(r)}_j(x_0, x_1, x_2, x_3)$ with*

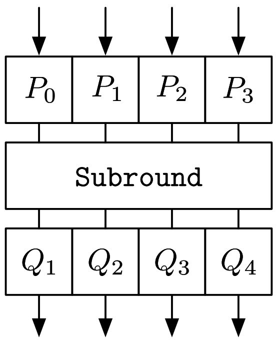

**Figure 1:** High-level view of an encoded subround of the Chow et. al. AES.

- $r = 1$ (*first*)

$$(y_0, y_1, y_2, y_3) = (Q^{(r)}_{0,j}(z_0), Q^{(r)}_{1,j}(z_1), Q^{(r)}_{2,j}(z_2), Q^{(r)}_{3,j}(z_3)),$$

*where* $(z_0, z_1, z_2, z_3) = \mathsf{Subround}^{(r,j)}(x_0, x_1, x_2, x_3),\ z_i \in \{0,1\}^8$

- $2 \le r \le R - 1$

$$(y_0, y_1, y_2, y_3) = (Q^{(r)}_{0,j}(z_0), Q^{(r)}_{1,j}(z_1), Q^{(r)}_{2,j}(z_2), Q^{(r)}_{3,j}(z_3)),$$

*where* $(z_0, z_1, z_2, z_3) = \mathsf{Subround}^{(r,j)}(P^{(r)}_{0,j}(x_0), P^{(r)}_{1,j}(x_1), P^{(r)}_{2,j}(x_2), P^{(r)}_{3,j}(x_3)),\ z_i \in \{0,1\}^8$

- $r = R$ (*last*)

$$(y_0, y_1, y_2, y_3) = \mathsf{Subround}^{(r,j)}(P^{(r)}_{0,j}(x_0), P^{(r)}_{1,j}(x_1), P^{(r)}_{2,j}(x_2), P^{(r)}_{3,j}(x_3)).$$

The output encodings $Q_i$ and input encodings $P_i$ of successive rounds are pairwise annihilating between `ShiftRows` to maintain the functionality of AES. For the first and last round there are no input and output encodings, respectively. Chow et al's whitebox implementation of AES-128 is provided as Algorithm 2.

**Algorithm 2** Chow et al's Whitebox AES implementation

```
$S^{(0)} \leftarrow \texttt{Plaintext}$
for $r = 1$ to 10 do
    $\hat{S}^{(r-1)} \leftarrow \texttt{ShiftRow}(S^{(r-1)})$
    for $j = 0$ to 3 do
        $(s^{(r)}_{0,j}, s^{(r)}_{1,j}, s^{(r)}_{2,j}, s^{(r)}_{3,j}) \leftarrow \mathsf{EnSubround}^{(r)}_j(\hat{s}^{(r-1)}_{0,j}, \hat{s}^{(r-1)}_{1,j}, \hat{s}^{(r-1)}_{2,j}, \hat{s}^{(r-1)}_{3,j})$
    end for
end for
$\texttt{Ciphertext} \leftarrow S^{(10)}$
```

### 2.2.1 Composition of Encoded Subround

In practice, each EnSubround is created from four types of smaller lookup tables (LUTs):

- Level 1 T-boxes (Table A): These LUTs combine the effect of `AddRoundKey` and `SubBytes`, `ShiftRows` and a part of the `Mixcolumns` transformations. Each round consists of 16 such T-boxes that map a byte value to a 4-byte (i.e 32 bit) value. Hence there are essentially 160 such T-boxes in total. In order to diffuse the information, the 32 bit output of partial `MixColumns` layer is subject to an invertible linear transformation called Mixing bijection. Table A consists of $R_j^{(r)} \circ MC_i \circ T_{i,j}^{(r)} \circ (L_{i,j}^{(r)})^{-1}$, where $MC_i$: $\{0,1\}^8 \to \{0,1\}^{32}$ is the $i$-th column operation of MixColumn. $(L_{i,j}^{(r)})^{-1}$: $\{0,1\}^8 \to \{0,1\}^8$ is a secret 8-bit linear bijection and $R_j^{(r)}$: $\{0,1\}^{32} \to \{0,1\}^{32}$ is the Mixing bijection, referred to previously. Note that for each of the 4 bytes input EnSubround modules in one AES round, only a partial Mixcolumn multiplication with the $i^{th}$ column of the MDS matrix is done. In order to complete the Mixcolumn operation, the results of the multiplication needs to be summed. This is precisely what is done in the tables of the next level.

- Level 2 Xor-tables (Table C1): These tables are essentially used to perform the addition operations of the `MixColumns` layer that were not performed in the construction of the Level 1 T-boxes. Table C is an 8 to 4 bits XOR table. 32-bit XOR operation is realized by 2-layer six Table C1 calls.

- Level 3 T-boxes (Table B): These LUTs partially cancel the effect of Mixing bijections applied in the level 1 T-boxes, by multiplying each byte input with one column of the inverse Mixing bijection matrix. As a result these tables also map byte values to 4-bytes. Table B comprises $L_j^{(r+1)} \circ (R_{i,j}^{(r)})^{-1}$. $(R_{i,j}^{(r)})^{-1}$: $\{0,1\}^8 \to \{0,1\}^{32}$ basically represents multiplication by the $8i - 8i + 7^{th}$ columns of the inverse matrix of $R_j^{(r)}$. Since only a partial multiplication is done with the $(R_j^{(r)})^{-1}$ matrix, as in level 1, the addition operations omitted in this level is performed in the tables of the next level. $L_j^{(r+1)}$: $\{0,1\}^{32} \to \{0,1\}^{32}$ is a 32 bit linear bijection which is chosen to cancel the effect of $(L_{i,j}^{(r+1)})^{-1}$. In fact functionally, it is simply the concatenation of the 8-bit bijections $L_{0,j}^{(r+1)}||L_{1,j}^{(r+1)}||L_{2,j}^{(r+1)}||L_{3,j}^{(r+1)}$.

- Level 4 Xor-tables (Table C2): These tables are essentially used to perform the addition operations that are not performed in the construction of the Level 3 T-boxes.

Thus each round in Chow's whitebox AES consists of a number of Level 1 to 4 lookup tables. Of these, only the level 1 tables are key-dependent. To be more precise, each round $r$ can be seen as a parallel application of 4 encoded subrounds $\mathsf{EnSubround}_j^r$ for $0 \le j \le 3$. Each $\mathsf{EnSubround}_j$ would operate on a quarter of the 128 bit state i.e. 32 bits and produce a 32 bit output. Each $\mathsf{EnSubround}_j$ is thus composed of the four types of tables (Table A, B, C1, C2) as shown in Figure 2.

### 2.2.2 Visible States.

We use the following notations. For each $\mathsf{EnSubround}_j$

- Let the 128-bit states after Table A and B in round $r$ be denoted as $TA_j^{(r)} = \{TA_j^{(r)}[0], \ldots, TA_j^{(r)}[15]\}$ and $TB_j^{(r)} = \{TB_j^{(r)}[0], \ldots, TB_j^{(r)}[15]\}$, $TA_j^{(r)}[i] \in \{0,1\}^8$ and $TB_j^{(r)}[i] \in \{0,1\}^8$, respectively.

- Let the 32-bit state of at the input of the Table B be layer $TC_j^{(r)} = \{TC_j^{(r)}[0], \ldots,$ $TC_j^{(r)}[3]\}$, $TC_j^{(r)}[i] \in \{0,1\}^8$ is the byte input to the $i^{th}$ Table B in $\mathsf{EnSubround}_j$.

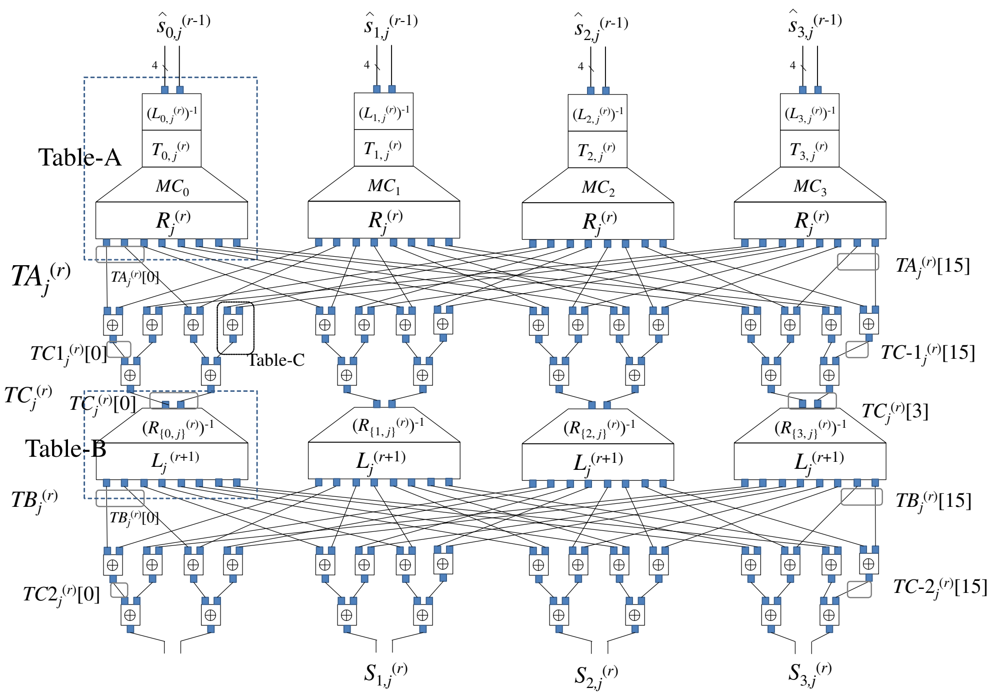

**Figure 2:** Table-based composition of the encoded subround of the Chow et. al. AES.

- Define 64-bit internal states of table C1 and C2 as $TC1_j^{(r)} = \{TC1_j^{(r)}[0], \ldots, TC1_j^{(r)}[15]\}$, $TC1_j^{(r)}[i] \in \{0,1\}^4$, and $TC2_j^{(r)} = \{TC2_j^{(r)}[0], \ldots, TC2_j^{(r)}[15]\}$, $TC2_j^{(r)}[i] \in \{0,1\}^4$ (see Fig. 2).

The internal states which appear in the memory during the computation are as follows: $S^{(r)}$ for $1 \leq r \leq 9$, and $TA_j^{(r)}$, $TB_j^{(r)}$, $TC_j^{(r)}$, $TC1_j^{(r)}$ and $TC2_j^{(r)}$ for $0 \leq j \leq 3$ and $1 \leq r \leq 9$. The total size of visible states is estimated as 16128 bits $(= (128 \times 9) + ((128 \times 2 + 32 + 64 \times 2) \times 4) \times 9)$ bits, namely 2016 bytes. Essentially, the designer's challenge is to use protection techniques like obfuscation and randomization to ensure security assuming that all these bytes are leaked to the adversary. Similarly, the adversary's challenge would be to use the information in the leaked bytes and counteract any additional protection measure employed by the designer to recover the secret key.

### 2.2.3 External Encoding.

128-bit external encodings $IN$ and $OUT$, which are randomly and uniformly selected linear mixing bijections, are added to the input and output of AES, respectively, to protect the tables of the first and last rounds and mitigate code lifting attacks. The action of the composite transform is $OUT \circ E_K \circ IN^{-1}$. Thus, the algorithm becomes an encoded variant of AES, i.e. a different cipher. The affects of $IN^{-1}$ and $OUT$ are canceled out in the contents server and the user device, respectively. In real-world applications such as banking or the standard DRM protocols such as Marlin, implementations without external encodings are deployed for interoperability and standard-compliance. Also, as discussed in the [BHMT16], in practice, at least one encoding usually is known to the adversary in the whitebox model, because the external encodings are canceled locally by the software in the same device. In this paper, we consider the variant without external encodings as with recent results [BHMT16, BI15, SMdH15].

## 3 Differential Computation Analysis (DCA)

A side channel attack called a differential computation analysis has been proposed by Bos et al. [BHMT16] and Sanfelix et al. at Black Hat Europe [SMdH15]. This attack exploits memory access patterns during the software execution of whitebox AES, and is essentially the same as Differential Power Analysis (DPA) on block ciphers (for a complete analysis of DPA, please refer to [KJJR11]). This is a form of side channel attack in which the attacker studies the power consumption of a cryptographic hardware device and tries to deduce the secret key by performing statistical analysis on the power traces.

DPA usually targets some intermediate variable signal of the cryptosystem that depends on a small number of bits of the secret key and the plaintext. For example a simple DPA on an unprotected version of AES-128 could target the byte of the state $V = Sb(PT \oplus K)$ just after the `SubBytes` layer in the first round.The attacker collects power traces $\mathcal{P}_i(t)$ for numerous plaintexts $PT_i$ (the variable $t$ in the parenthesis denotes the time index). In order to do so the attacker selects a model for power consumption which tells him how much power would be consumed if the internal variables change in a certain fashion. For CMOS circuits the most realistic model is the Hamming weight model, since for CMOS transistors switching a bit from 0 to 1 or from 1 to 0 requires the same amount of energy, and usually all the machine bits handled at a given time consume the same energy. As a result if one were to eavesdrop on the power signal on the CPU bus carrying the signal $V$, the power consumption would likely be proportional to $HW(Sb(PT \oplus K))$, where $HW$ denotes Hamming weight. However, there are two issues in this approach

- The power consumed is usually has an additive noise component. It is realistic to assume that the noise is zero mean and cancels out over multiple traces.
- The power traces are obtained for each plaintext over the entire duration of the encryption process. Therefore power consumed during the computation of $V$ is likely to be reflected in a small range of time.

For each guess $K_j$ of the key byte $K$, the attacker computes the values $H_{ij} = HW(Sb(PT_i \oplus K_j))$ (where $i$ ranges over all the plaintexts for which power traces are recorded). If $K_j$ is the true value of $K$ then for some value of $t$, the sequence $\mathcal{P}_i(t)$ would be perfectly correlated with the sequence $H_{ij}$. Thus, the attacker can compute a simple correlation coefficient (for all $t$)

$$\rho_j(t) = \frac{\sum_i (H_{ij} - \overline{H}_j) \cdot (\mathcal{P}_i(t) - \overline{\mathcal{P}}_t)}{\sqrt{\sum_i (H_{ij} - \overline{H}_j)^2 \cdot \sum_i (\mathcal{P}_i(t) - \overline{\mathcal{P}}_t)^2}}$$

where $\overline{H}_j$ denotes the arithmetic mean of the $H_{ij}$'s for a particular keyguess $K_j$ and $\overline{\mathcal{P}}_t$ denotes the arithmetic mean of the power traces $\mathcal{P}_i(t)$ for a particular time instance $t$. The correct guess of $K_j$ is likely to maximize $\rho_j(t)$ for some value of $t$, and the attacker deduces the value of the secret key byte in this manner.

In DCA the adversary instead monitors software execution to exploit side-channel leakages. The program that is being analyzed is run using a binary instrumentation framework such as PIN [LCM+05] or Valgrind [NS07]. These frameworks allow the execution of a program to be observed and modified by inserting code into it during runtime. In the case of the DCA tools, every memory operation is instrumented with a function that records the values and addresses the operation reads or writes. This is done while encrypting random plaintexts, and the recordings (referred to as *software traces*) are then serialized as if they were power traces. In the context of whitebox encryption, the memory address accesses leak revealing information. Since the memory address space is used to store the several LUTs, the exact address of the memory accesses can be used to extract the inputs to the LUTs. Any standard DPA tools expecting power traces can then be used to extract the key. Since the traces directly observe the values being read and written, they are equivalent to power traces made directly on internal chip lines.

For each guess of a key byte the attacker computes the hypothesized intermediate values in the AES encryption of the plaintexts that were traced. If a key guess is correct, then at some offset in the traces the hypothesized state would be perfectly correlated with the sequence recorded values. Thus, the attacker can compute a simple correlation coefficient to recover the key byte. At this point we categorize DCA attacks into two paradigms

- Address based DCA (ADCA): This is basically the attack technique used in [BHMT16], which uses the memory address accesses available in the software traces to construct a power attack. As further pointed out in this paper, in a software setup all observations are made in the absence of measurement noise which is observed while recording power traces in a hardware device, the attack can be performed efficiently on every bit of information rather (which is akin to eavesdropping on every single wire with a probe). However as we will show, if one were to randomize the locations of the tables in the memory space, an attack using memory address accesses becomes practically difficult.

- Value based DCA (VDCA): We observe that if table locations are randomized a more useful source of information may be the values stored in the memory addresses, rather than the addresses itself. Thus in such an event we can instrument the whitebox binary to additionally include the values stored in memory locations in each software trace. These are nothing but the table outputs which can similarly used to mount a power analysis attack.

## 4 Software Countermeasures

Software countermeasures have become essential to counteract whitebox attacks. In absence of any countermeasures, the whitebox encryption schemes of Chow et. al. [CEJvO02b], Karroumi [Kar10], and Xiao-Lai [XL09] have all been broken in practical time complexity. We test a number of software countermeasures to counteract the existing attacks, including the one we propose in Section 5. These countermeasures are carried over from the literature on side-channel analysis, where they play the same role in defending against DPA. In previous articles on DCA it has been argued that these countermeasures do not add complexity to the attacks, since they seem to be easy to defeat using automated tools. However we argue that this might not always be the case. If the binary that is being analyzed is protected using obfuscation and integrity checking, it can become prohibitively difficult to fully analyze its functionality. Therefore side-channel attacks, where there is no need to reverse-engineer the implementation to extract the key, are more interesting to attackers. This means that attacks which are resistant to countermeasures are valuable since they work even if the countermeasures cannot be defeated automatically.

### 4.1 Control flow obfuscation

One example of such a case is in control flow obfuscation (or randomization). Control flow obfuscation essentially tries to shuffle the order of table accesses that are performed in the execution of each round of the encryption operation. We know that each round in AES can be seen as a parallel application of 4 encoded subrounds. The order of execution of the 4 subrounds can be shuffled using a PRNG as it does not make any difference in the final result. Furthermore, inside each subround the order of table accesses can be shuffled. In the general case we can build a dependency graph for table accesses to assess which tables must be accessed before the others. The nodes that reside at the same level in the graph, represent table accesses whose execution order can be randomly shuffled.

Listing 1: Example of data dependencies.

```
1 v0 = table_0[v1];
2 v1 = table_1[v0];
3 v2 = table_2[v0];
4 v0 = table_3[v3];
```

As an example of this technique, consider the simple example C code in Listing 1. Note that the computation of lines 2 and 3 depend on line 1, since v0 is used as the offset into the tables. For line 4 it is important that both lines 2 and 3 have run, since v0 is overwritten. The corresponding dependency graph is illustrated in Figure 3. In this case we may shuffle the order of lines 2 and 3, since they are independent.

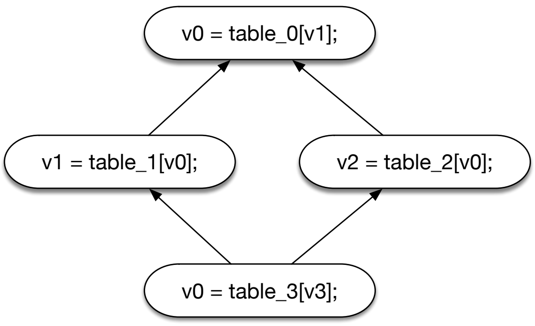

**Figure 3:** Dependency graph of example program in Listing 1.

To counter this one might argue that the attacker can then remove the entropy to the PRNG, rendering the control flow static. But this might not always be so simple. Instead of relying on an external PRNG, the whitebox system can supply it's own. The generator can then be seeded with the plaintext, which ensures that no external entropy is needed to randomize the control flow. If the binary is sufficiently obfuscated we find this unlikely to be defeated by an automated tool. An alternative technique to defeat control flow obfuscation is to automatically realign the traces using the addresses of the instructions that were executed. If each leak of a byte from memory notes the current instruction pointer, we can later reorder the traces such that the instruction addresses match. However this will not work if the shuffled implementation runs only 1 basic block of instructions, and instead randomizes a set of pointers that define the order of operations.

Most of the attacks present in cryptographic literature against whitebox schemes are against binaries that do not employ software protection, and so control flow obfuscation is an effective tool to counteract them. However as we will later show, binaries applying only control flow shuffling as a software countermeasure are susceptible to both ADCA and ZDE attacks. Thus one requires additional protection measures.

## 4.2 Table location randomization

This DCA attack described by Bos et. al. uses the memory address accesses available in the software traces to construct a power attack. Since these memory addresses essentially reveal the inputs to the various tables accessed, they reveal enough information to perform a DCA. Table location randomization is akin to the masking countermeasure applied in hardware architectures to thwart power attacks. The memory addresses are protected by

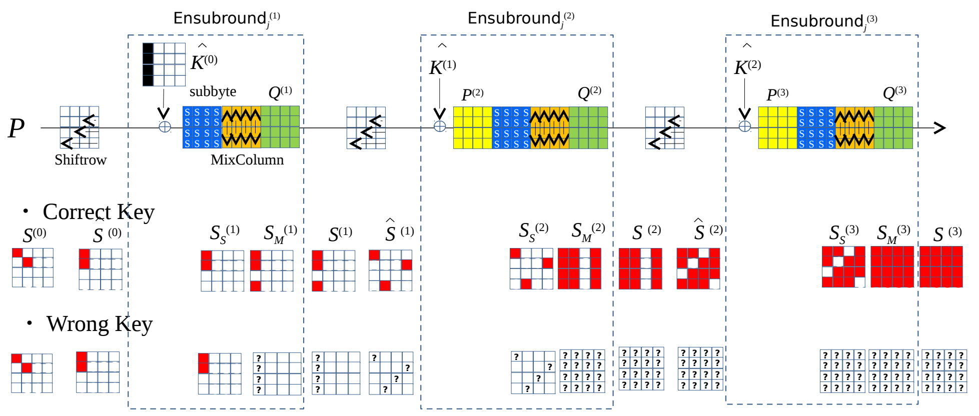

**Figure 4:** Truncated differential characteristics of Chow et al's whitebox AES-128 implementation

adding random offsets to them. This effectively disperses the tables at random locations in the address space. Although this countermeasure protects against Address based DCA (ADCA), it does not provide protection if a Value based DCA (VDCA) attack is employed. The zero difference enumeration attack described in Section 5 is also able to attack binaries employing location randomization.

## 4.3 Dummy operations

An effective way of thwarting power attacks was proposed in [Man04] that utilizes random disarrangement of time. The idea works as follows: recall that in a standard DPA attack, the attacker computes the correlation coefficient $\rho_j(t)$ for each key guess $K_j$ for all the time range for which he obtains power traces. The internal variable $V$ which the power attack targets is usually computed in the first round itself, and thus the samples that indicate the power consumption for $V$ are likely to be located in a small time range at the beginning of the power trace for each plaintext. For a system that does not employ disarrangement the time range over which the intermediate variable $V$ is calculated is likely to align itself for each new plaintext for which traces are recorded. Thus it becomes easy to compute the correlation coefficients. The correct key guess is likely the one which maximizes $\rho_j(t)$ for some $t$.

The idea of disarrangement is to randomize the time instance at which the intermediate $V$ is computed for each execution of the encryption operation. As a result for each new plaintext, $V$ is likely to be computed at different time instances. As a result, in the power traces, the time instances at which $V$ is computed no longer align with each other. And so it becomes more difficult to mount a power attack. In the context of whitebox encryption, disarrangement is achieved by adding random number of dummy table lookups in between each legitimate table access. This essentially breaks the alignment pattern for table accesses for each new plaintext for which traces are recorded.

## 5 Zero Difference Enumeration (ZDE) attack

Since the countermeasures that randomize the control flow or insert dummy operations will misalign the DCA memory traces, we may consider an attack model where this does not matter. We propose the *zero difference enumeration* attack inspired by concepts of differential cryptanalysis of symmetric-key cryptosystems. The attack works by choosing special pairs of plaintexts which are encrypted using the whitebox implementation. The pairs are constructed such that if our key guess is correct, a significant number of byte values in the two corresponding AES states will be identical during encryption. Because whitebox implementations often use many tables to process the bytes of the state, this will further amplify the number of such bytes. Determining which key is correct is then a matter of testing which pairs have the highest number of equal bytes in the AES state during the encryption.

The pair of plaintexts we will henceforth refer to as a $\beta$-plaintext pair. The attack proceeds as follows:

- The attacker starts by guessing a small portion of the Secret key. In this case, he begins by guessing 2 bytes of the key.
- For all possible values of the key guess $K_j$, the attacker prepares a $\beta$-plaintext pair $PT_1, PT_2$ so that the difference $PT_1 \oplus PT_2 = f(K_j)$ is some function of the keyguess.
- The function $f$ is chosen in a manner so that, if the keyguess is correct, then many internal state variables incurred during rounds 1, 2, 3 of the encryption module for both $PT_1$ and $PT_2$ are the same.
- As a result, the corresponding table inputs and outputs accessed during the encryption of $PT_1$ and $PT_2$ are the same if the keyguess $K_j$ is correct.
- Using the above as a distinguisher, the attacker goes through all the possible keyguesses and selects the keyguess for which the number of similar table inputs/outputs for the $\beta$-pair are maximized.

The main advantage of ZDE is that the attack seems robust to countermeasures like control flow obfuscation and randomization of table locations in the memory.

## 5.1 How to get a key-dependent $\beta$-plaintext pair

We will now describe the process to obtain a $\beta$-plaintext pair for any given keyguess. We will refer to the Figure 4 for this. Let states after SubBytes and MixColumns in EnSubround$^{(r)}$ be $S_s^{(r)}$ and $S_M^{(r)}$, respectively.

- First, the attacker guesses the first two key bytes $\{\hat{k}_{0,0}^{(0)}, \hat{k}_{0,1}^{(0)}\}$.
- The attacker then chooses arbitrarily the entire first column, $\{s_{M0,0}^{(1)}, \ldots, s_{M0,3}^{(1)}\}$, of the internal state $S_M$ which is the state just after the `MixColumns` layer of the first round.
- The logic behind choosing the entire first column is as follows: it allows the the attacker to invert the `MixColumns` layer on the first column and calculate the first column of $S_S$ i.e. the state after the `SubBytes` layer.
- The `SubBytes` layer can be inverted to get the entire first column which is the state just after the first `AddRoundKey`.
- Since the first 2 bytes of the key is already guessed, the attacker can thus compute the first two bytes in the main diagonal of the plaintext $PT_1$ by inverting `AddRoundKey` and `ShiftRow` operations.
- For the plaintext $PT_2$, we do the following: Generate any byte difference $\Delta$, and add the 32 bit difference $\{\Delta, 3\Delta, 0, 2\Delta\}$ to the first column of $S_M$ to get the state $S'_M$.

**Table 1:** Probability-one zero for correct and wrong keys when encrypting $\beta$-plaintext pair.

| Correct key (307 bytes) |
|---|
| $s_{0,2}^{(1)},\ s_{i,j}^{(1)},\ s_{2,j}^{(2)} \quad (0 \le i \le 3, 1 \le j \le 3)$ |
| $TA_0^{(1)}[8-15],\ TA_j^{(1)} \quad 1 \le j \le 3$ |
| $TA_0^{(2)}[4-15],\ TA_1^{(2)}[0-11],\ TA_2^{(2)},\ TA_3^{(2)}[0-3, 7-15],$ |
| $TA_0^{(3)}[7-11],\ TA_1^{(3)}[4-7],\ TA_2^{(3)}[0-3],\ TA_3^{(3)}[11-15],$ |
| $TB_0^{(1)}[2, 6, 10, 14],\ TB_j^{(1)}\ (1 \le j \le 3),\ TB_2^{(2)}$ |
| $TC_j^{(1)}\ (1 \le j \le 3),\ TC_2^{(2)}$ |
| $TC1_0^{(1)}[3-7],\ TC_j^{(1)}\ (1 \le j \le 3)$ |
| $TC1_0^{(2)}[3-7],\ TC_1^{(1)}[0-3],\ TC_1^{(2)},\ TC_1^{(3)}[3-7]$ |
| $TC2_0^{(1)}[2, 6],\ TC2_j^{(1)}\ (1 \le j \le 3),\ TC2_2^{(2)}$ |
| Wrong key (244 bytes) |
| $s_{i,j}^{(1)}\ (0 \le i \le 3, 1 \le j \le 3)$ |
| $TA_0^{(1)}[8-15],\ TA_j^{(1)}\ (1 \le j \le 3)$ |
| $TA_0^{(2)}[4-15],\ TA_1^{(2)}[0-11],\ TA_2^{(2)}[0-7, 12-15]$ |
| $TA_3^{(2)}[0-3, 7-15],\ TB_j^{(1)},\ TC_j^{(1)}, TC2_j^{(1)} \quad (1 \le j \le 3)$ |
| $TC1_0^{(1)}[3-7],\ TC_j^{(1)}\ (1 \le j \le 3)$ |
| $TC1_0^{(2)}[3-7],\ TC_1^{(1)}[0-3],\ TC_1^{(2)}[0-3],\ TC_1^{(3)}[3-7]$ |

- Thereafter as before, the `MixColumns`, `SubBytes`, `AddRoundKey`, and `ShiftRows` layers can be inverted to get the first two bytes in the diagonal of $PT_2$ by using guessed two key bytes $\{\hat{k}_{0,0}^{(0)}, \hat{k}_{0,1}^{(0)}\}$.
- The remaining 14 bytes of $PT_1$ and $PT_2$ can be assigned with the all zero byte value or the same random byte value. (Note that a similar exercise can be done for other double byte values of the key).

Due to the property of the MDS matrix used for the linear `MixColumns` layer in AES-128, the difference $\{\Delta, 3\Delta, 0, 2\Delta\}$ between $S_M$ and $S'_M$ ensures that the difference between the states $S_S$ and $S'{}_S$ is given by $\{\Delta, \Delta, 0, 0\}$. As described in Figure 4, if the two key bytes are guessed correctly, then the $\beta$-plaintext pairs will produce a differential trail shown by the upper half of of the figure. If not, the differential trail becomes unpredictable from the second round onward. Note that the bytes in red indicate a probability 1 difference, the bytes with ? indicate an unpredictable difference and a byte in white indicate that the corresponding bytes in the internal state produced during the encryption of the $\beta$-pair are same with probability 1.

## 5.2 Zero Difference Bytes

In essence the table accesses involving the white bytes in Figure 4 will lead to a larger amount of equal values when tracing an encryption of the $\beta$-plaintext pair, when the 16-key bits are guessed correctly. For the Chow et. al. AES encryption we found that if the keyguess is correct we have 307 equal bytes, and if the keyguess is incorrect we only have 244 as shown in Table 1.

The DBI techniques allow the attacker to plant $\beta$-pairs into the binary framework and record software traces. The attack can be performed offline after sufficient traces have been recorded. The attacker constructs multiple $\beta$-pairs for all possible keyguesses, and the keyguess which results in the highest average number of equal table outputs for the $\beta$-pairs in the first 3 encryption rounds is likely to be the correct key value.

## 6 Experimental results

Along with the paper introducing the concept of DCA [BHMT16], a toolset for practical analysis of binaries was provided by Bos et. al. The toolset can be used to record execution traces of target binaries, and apply DPA attacks to trace files of AES and DES encryptions. To evaluate the impact of hiding on DCA we apply this toolset to binaries containing countermeasures, and use the number of traces required for key recovery as a measure of complexity. Since each trace is an independent recording of an encryption of a random plaintext, they may be collected in parallel. If the resources needed to perform one trace are known, the final time requirement for the tracing can therefore be scaled to the hardware available to an attacker. Unless otherwise noted we collected the traces on a laptop with a 1.7GHz Core i7 (I7-4650U, which has 2 cores) and 8GB of RAM. Since the DCA tools and challenges run on linux, all the experiments were run in a linux virtual machine (VM).

For each set of recorded traces we attempt to recover the key by attacking the state after the level one T-box in the first AES round. With the number of traces we have available this might not always recover the whole key successfully, but it shows the effort required to complete the whole attack. For our results we record the number of correct key bytes that are returned as the top DCA candidate, ranked on the absolute value of the bit correlations. Since some key bytes might rank slightly lower due to a too low number of traces, we also note if the correct key bytes were found in the list of the top 10 results.

### 6.1 CHES 2016 whitebox challenge

**Table 2:** DCA on the CHES 2016 challenge.

| Attack type | Countermeasures | Time per trace | Size per trace | # Traces | # Key bytes (found in top 10) | # Key bytes (found as best) | Correlation time (h:m:s) |
|---|---|---|---|---|---|---|---|
| ADCA | None | 0.65 seconds | 19,248 bytes | 4,000 | 16 | 15 | 48:36 |
| | Shuffling | 3.14 seconds | 6,224 bytes | 4,000 | 13 | 12 | 16:22 |
| | | | | 10,000 | 13 | 12 | 40:06 |
| | Shuffling and random offsets | 3.35 seconds | 51,680 bytes | 10,000 | 0 | 0 | 6:41:59 |
| | Dummy operations | 4.61 seconds | 38,768 bytes | 4,000 | 1 | 0 | 1:41:09 |
| VDCA | None | 0.65 seconds | 19,248 bytes | 4,000 | 11 | 4 | 49:15 |
| | Shuffling | 3.14 seconds | 6,224 bytes | 4,000 | 5 | 0 | 16:34 |
| | | | | 10,000 | 5 | 1 | 41:08 |
| | Shuffling and random offsets | 3.35 seconds | 40,448 bytes | 10,000 | 13 | 12 | 5:15:57 |
| | Dummy operations | 4.61 seconds | 38,768 bytes | 4,000 | 0 | 0 | 1:49:29 |

Metrics for DCA on the CHES 2016 whitebox challenge with countermeasures. The time per trace is the total time to record and store one execution trace. Address and value traces are recorded at the same time, which is why the time per trace is equal in ADCA and VDCA. The size per trace is the bytes of storage used per execution trace, when recording 1/3 of the encryption function with appropriate filtering (see subsections). We record the number of correct key bytes that are ranked in the top 10 and best position according to the correlation value. The time is the total time to run the DPA correlation tool on the traces, excluding tracing.

For the CHES conference of 2016, which highlights new results in the design and analysis of cryptographic hardware and software implementations, a set of challenges were given as a competition. The challenges involve power analysis and creating secure implementations of AES encryption, and an implementation of the Chow et. al. whitebox scheme was included amongst these. The challenge is delivered as both a compiled linux binary, and the corresponding C source code. When run the program encrypts an input plaintext using AES and prints the resulting ciphertext. The goal is to recover the AES key from the encryption function, which we do by applying the Bos et. al. DCA tools to the compiled binary.

To determine the area to trace, one can perform a superficial analysis in a disassembler such as IDA Pro. Since the binary is not obfuscated, it is easy to identify the address range corresponding to the instructions in the encryption function. We may then choose to only

**Table 3:** ZDE on the CHES 2016 challenge.

| Attack type | Countermeasures | Time per trace | Size per trace | # Traces | # Key bytes found | Total time (h:m:s) |
|---|---|---|---|---|---|---|
| ZDE | None | 0.000012 seconds | 2,048 bytes | $500 \cdot 2^{17}$ | 2 | 0:3 |
| | Shuffling | 0.001641 seconds | 2,048 bytes | $500 \cdot 2^{17}$ | 2 | 7:0 |
| | Shuffling and random offsets | 0.003594 seconds | 2,048 bytes | $500 \cdot 2^{17}$ | 2 | 15:20 |
| | Dummy operations | 0.071543 seconds | 4,096 bytes | $5000 \cdot 2^{17}$ | $0^{\star}$ | 5:5:15 |

Metrics for ZDE on the CHES 2016 whitebox challenge with countermeasures. The attack was run as 256 parallel instances on a cluster, and the final scores of candidate keys was filtered to identify the top choice. The time is the total time to run the attack, including tracing. Only 2 key bytes were attacked. $\star$: The correct key bytes were ranked as number 2.

record when the binary is executing in this address range. Since the code for the challenge consists of a long line of table lookups we have chosen to trace for approximately 1/3 of the encryption function. We will be attacking the state after the level one T-box lookup in the first round, so this gives us some margin of error when choosing the range. Because all the whitebox tables used in the challenge are static, we can also narrow recording to only values that are read and written from the data section. If this was not the case, one could instead use the trace graphing tool that is provided as part of the toolkit. This tool allows one to visualize the addresses that are read and written, and spot patterns corresponding to the encryption algorithms. Narrowing the range of addresses to consider is then easy. For all instructions referencing memory we record the lowest 8 bit of addresses and 8-bit

**Table 4:** Ranking for individual key bits when attacking the CHES 2016 challenge.

| | **Key byte** | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Target bit** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | 244 | 212 | 255 | 0 | 0 | 0 | 0 | 255 | 249 | 255 | 0 | 0 | 250 | 0 | 233 | 255 |
| 1 | 255 | 243 | 255 | 255 | 0 | 255 | 255 | 252 | 0 | 255 | 255 | 0 | 254 | 255 | 235 | 0 |
| 2 | 135 | 204 | 244 | 171 | 0 | 173 | 240 | 1 | 0 | 203 | 186 | 80 | 0 | 237 | 0 | 204 |
| 3 | 0 | 0 | 254 | 199 | 209 | 219 | 0 | 0 | 128 | 191 | 0 | 202 | 243 | 0 | 206 | 130 |
| 4 | 255 | 0 | 0 | 0 | 250 | 0 | 236 | 255 | 0 | 0 | 242 | 253 | 245 | 247 | 0 | 249 |
| 5 | 251 | 41 | 250 | 216 | 129 | 0 | 0 | 244 | 251 | 228 | 60 | 157 | 251 | 153 | 0 | 215 |
| 6 | 255 | 211 | 194 | 82 | 255 | 179 | 254 | 220 | 0 | 135 | 4 | 0 | 179 | 232 | 249 | 0 |
| 7 | 0 | 255 | 0 | 255 | 0 | 248 | 254 | 0 | 255 | 255 | 0 | 232 | 0 | 253 | 0 | 1 |
| **Recovered** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | - | ✓ | ✓ |

Here we show the rank of the correlation score for the individual key bits when solving the challenge using 4,000 address traces. The candidates are ranked according to the absolute value of the correlation. As also noted in [BHMT16] the correct guesses tend to either be the top (0th) or bottom (255th) ranked candidate. We recover 15/16 key bytes, as shown by the checkmarks.

data values separately, since both can be used to perform the DCA. When looking up in a whitebox table the implementation uses the output of another table, thus memory addresses will be correlated with the output data from another memory operation. In total we collected 4,000 traces of encryptions, which took 43 minutes and 148MB of disk space. We then searched for the key using the included DPA tool on the address traces, finding what turned out to be 15/16 of the key bytes. After also running the attack on the traces of values we complete the whole key, which we then verified by decrypting ciphertexts again using openssl. The time taken for key recovery was 48 minutes of correlation on the address traces, and 49 minutes on the value traces. For the traces of values the correlations are not as strong, and we find only 4/16 bytes as the top 10 candidates. In Table 4 we show the rank of the correlation score for each byte of the key, when attacking the state after the level one T-box in the first round using address traces.

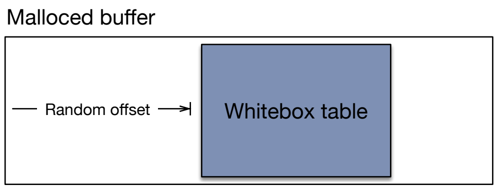

**Figure 5:** Random offset of table in memory.

## 6.2 Shuffling of operations

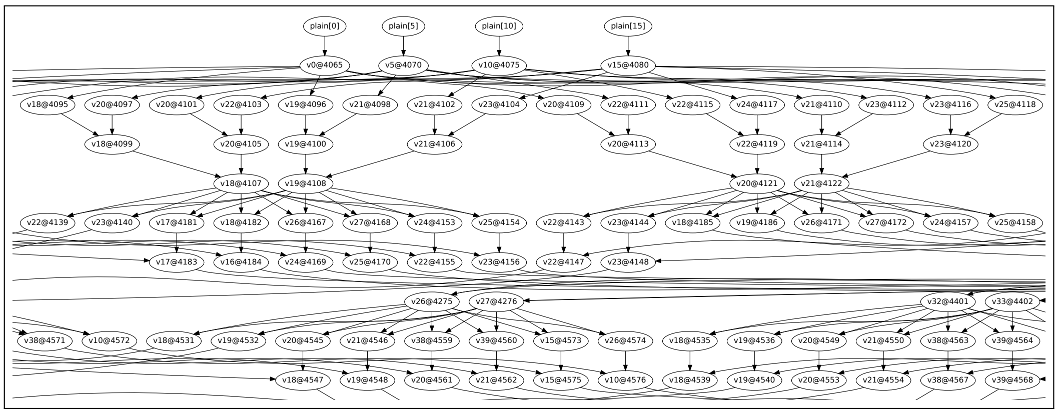

**Figure 6:** Data flow in the CHES 2016 challenge. The graph shows how the data flows from 4 bytes of the input plaintext through the tables of the CHES 2016 challenge whitebox. Each node represents a line in the source code, and is annotated with the variable that is modified followed by the line number. What is seen is part of a masked MixColumns step on nibbles, followed by masked xor tables to combine the results.

To experiment with the effect of countermeasures in practice, we apply the techniques to the CHES 2016 challenge and compare the difficulty of finding the key using DCA. The C code contains 4048 static tables of data that implement the whitebox scheme, and which are used in an encryption function that consists only of table lookups. To visualize the actual structure of the encryption we have parsed the code and built a graph of the data flow inside the encryption algorithm. A part of the this graph can be seen in Figure 6. If it is compared with the description of the encoded AES algorithm, it is clear that this is part of the masked calculation of the first column of the AES state.

The first countermeasure we want to test is shuffling the order of operations, and to do this we built a dependency graph for the lines of the encryption function as previously specified. The graph records which table lookups depend on data from previous lookups, and which operations must be run before a line can overwrite the contents of a variable. From Figure 6, it should be clear that some of the operations can be performed in arbitrary order. However since only a few local variables are used to store the state, we must be careful not to overwrite a result that is needed later. After constructing the dependency graph the code can be transformed into a shuffled implementation. For each batch of operations that may run in parallel we emit a block of C code that randomly shuffles their order on each run using a random number generator. After compiling the code we then apply the DCA attack to the resulting binary. Now 4,000 traces are no longer enough to recover the correct key when taking the top 10 key byte candidates for each position.

When we increased the number of recorded traces to 10,000, which took approximately 10 hours and used 120 MB of disk space, there was no significant increase in the number of recovered key bytes.

## 6.3 Masking memory addresses

Since DCA attacks may rely on the addresses of memory lookups to perform correlations, we also attempt to mask these in the CHES 2016 challenge. We have written a parsing script to extract the 4048 tables in the C code and emit code to relocate them randomly on each encryption call. This is done by allocating a large buffer that the tables are copied to on each invocation of the encryption function. A table is randomly moved to any of 256 offsets in memory, such that the least significant byte of the addresses of lookups is randomly chosen (see Figure 5). This is equivalent to adding a random constant modulo 256 to the addresses that are recorded for table lookups. After compiling the binary with both random shuffling and masking of memory addresses, we again apply the DCA tools.

This time recording 10,000 traces takes approximately 12 hours, and the total size of the traces is approximately 1 GB. This is because we cannot use the address range of the data segment to limit recording. Now that the buffer is allocated on each run, all memory accesses that have a size of 1 byte must be recorded. With these countermeasures, 10,000 traces of memory addresses result in no identified key bytes. As expected, using the data instead of addresses has a higher success rate, since the values themselves are untouched. The time taken to recover 1 key byte is approximately 25 minutes in both cases.

## 6.4 Dummy operations

In DPA attacks a common strategy is to hide the key-dependent operations in the time dimension by inserting dummy operations. This can be done either with random sleeps, or with actual operations using random data. Since DCA only records on memory accesses, inserting sleeps into the algorithm will not change the recorded traces. Only insertion of dummy operations which do irrelevant memory lookups is therefore possible. We parse the CHES 2016 C code to extract all table lookups, and insert a block of random dummy lookups after each one. For a simplified example of this transformation see Listing 2, where only line 1 is doing real work.

Listing 2: Example of dummy lookups.

```
1 v0 = table_0[v1];
2 for (int i = 0; i < rand() % 16; i++)
3         dummy = table_0[rand() % 128];
```

Recording 4,000 execution traces takes 5 hours 7 minutes, and results in trace files of 148 MB. We again attack the state after the first T-box substitution, and with the dummy operations countermeasure we are unable to find any correct key bytes. All the results of the DCA experiments on the CHES 2016 challenge are tabulated in Table 3.

**Table 5:** Zero-difference enumeration using PIN tool.

| Method of attack | Challenge, countermeasures | Experiments | Total number of traces | Time per $2^8$ key candidates |
|---|---|---|---|---|
| Source code | CHES 2016 challenge, none | 500 | $500 \cdot 2^{17}$ | 3 seconds |
| | CHES 2016 challenge, shuffling | 500 | $500 \cdot 2^{17}$ | 7 minutes |
| | CHES 2016 challenge, shuffling and random offsets | 500 | $500 \cdot 2^{17}$ | 15 minutes |
| PIN tool | Hack.lu 2009, protected AES | 1,000 | $1,000 \cdot 2^{17}$ | 18.75 seconds |
| | Klinec, Chow et. al. AES | 1,000 | $1,000 \cdot 2^{17}$ | 115.78 seconds |

## 6.5 Zero-difference enumeration

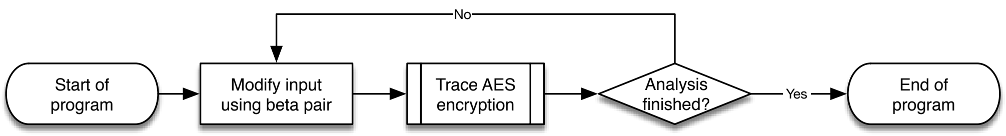

**Figure 7:** Modified control flow of instrumented binary. Our PIN tool changes the execution flow of the analyzed binary using a state machine that runs the ZDE attack. The state machine performs the required amount of experiments for each key candidate, and outputs the guess with the highest number of equal bytes in the traces.

After having tested DCA against different countermeasures, we now consider applying zero-difference enumeration attacks instead. Since these attacks only use values and not addresses, it is clear that they will not be affected by any countermeasure that adds random constants to table addresses. We can therefore focus on evaluating the random shuffling and dummy operation countermeasures.

As a comparison point we have tested ZDE against the unprotected CHES 2016 source code, recovering two key bytes as described in the section on ZDE attacks. Since all table outputs are leaked every time, we may directly compare two leaks for equality of bytes. We run the attack as 256 parallel processes that fix 8 bits of the key guess and test all remaining 8 bits. The attack finds the correct value of the 2 key bytes, with a runtime of approximately 3 seconds per $2^8$ key guesses. Because the attack is trivially parallellizable, this number can be scaled to the available computational resources. A total of 500 beta pairs are tested per key candidate, which is enough to distinguish the correct key bytes.

We now tested the ZDE attack against the same source code with shuffling that was used for the DCA experiments. Now that the order of returned values change on each encryption, we will have to compare all leaked bytes in the two traces against each other. We successfully find the 2 key bytes after spending 7 minutes per 256 key guesses. As a sanity check we have also run the zero-difference attack on the table-offsetting version of the CHES challenge. The attack time is then 15 minutes per 256 guesses, which is likely a result of the implementation being slower. As expected the correct 2 key bytes are also identified after 500 beta pairs are tested per key candidate. See Table 5 for the details details on all attacks.

With dummy operations the runtime of the encryption is slowed up to 16 times. In addition to this every random table lookup adds a leaked byte, which must be compared against all other leaked bytes in the other beta-pair encryption. This quadratic increase in bytes to test has a big impact on both runtime and experiments needed. We estimate that between 5,000 and 10,000 $\beta$-pairs are needed to find the correct key, with a runtime of approximately 10 hours for 256 key candidate tests.

To evaluate the feasibility of applying the zero-difference attacks to binaries, we have also created a PIN tool that can apply the attack. The tool works by repeatedly running the AES encryption functionality of the analyzed binary, while inserting beta pairs as the plaintext. The goal is to recover 2 bytes from the key. To identify at which points this should be done, the user must specify the start and end of encryption, and the location at which the input buffer can be found. The control flow of the binary is then modified as it is being run, as illustrated in Figure 7.

For the Hack.lu conference of 2009 a challenge based on AES encryption was published by Jean-Baptiste Bédrune[3]. This challenge is in the form of a Windows keygen-me, which

[3]http://2009.hack.lu/index.php/ReverseChallenge

requires that the input given to the program AES encrypts to a specific string. The AES implementation in the challenge uses whitebox-like ideas to hide the key, so the goal is also to recover it. This was previously done by Bos et. al. in their introductory paper on DCA. Our PIN tool can successfully recover the correct 2 key bytes of the first column in 80 minutes when running the program in a Windows VM. Note that the ZDE attack is trivially parallelized, so one could just run more instances of the cracker if this should be speeded up.

As part of the Master thesis of Dušan Klinec [Kli13], Klinec implemented a C++ version of the Chow et. al. and Karroumi et. al. AES whitebox designs. The code can generate tables for an AES key of choice and encrypt plaintexts using the table. Since the code is one of the few public implementations of whitebox AES systems available online, and has also been used when evaluating the DCA, we test our PIN tool on a compiled version of it. Without external encodings the PIN tool successfully recovers the 2 key bytes after 8 hours 14 minutes of tracing.

## 7 Conclusion

In this paper we explore some of the efficacies of software countermeasures on whitebox encryption. We show that while control flow obfuscation is essential to counteract existing attacks, it does not prevent DCA or ZDE attacks. We then looked at the table randomization technique and its relative strengths against the DCA and ZDE attacks. Finally we propose a countermeasure based on dummy table lookups which seem to counteract both ZDE and DCA attacks.

## References

[ARX14] ARXAN. TransformIT: Software-based Key Protection, 2014.

[BCD06] Julien Bringer, Hervé Chabanne, and Emmanuelle Dottax. White Box Cryptography: Another Attempt. *IACR Cryptology ePrint Archive*, 2006:468, 2006.

[BGE04] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, pages 227–240, 2004.

[BHMT16] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. Differential computation analysis: Hiding your white-box designs is not enough. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 215–236. Springer, 2016.

[BI15] Andrey Bogdanov and Takanori Isobe. White-box cryptography revisited: Space-hard ciphers. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1058–1069. ACM, 2015.

[CEJvO02a] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A White-Box DES Implementation for DRM Applications. In *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002, Washington, DC, USA, November 18, 2002, Revised Papers*, pages 1–15, 2002.

[CEJvO02b] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-Box Cryptography and an AES Implementation. In *Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers*, pages 250–270, 2002.

[Kar10] Mohamed Karroumi. Protecting White-Box AES with Dual Ciphers. In Kyung Hyune Rhee and DaeHun Nyang, editors, *Information Security and Cryptology - ICISC 2010 - 13th International Conference, Seoul, Korea, December 1-3, 2010, Revised Selected Papers*, volume 6829 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 2010.

[KJJR11] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *J. Cryptographic Engineering*, 1(1):5–27, 2011.

[Kli13] Dušan Klinec. White-box attack resistant cryptography. Master's thesis, Masaryk University, Brno, Czech Republic, 2013.

[LCM$^+$05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200. ACM, 2005.

[LN05] Hamilton E. Link and William D. Neumann. Clarifying Obfuscation: Improving the Security of White-Box DES. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, pages 679–684, 2005.

[LRM$^+$13] Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. Two Attacks on a White-Box AES Implementation. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 265–285, 2013.

[Man04] Stefan Mangard. Hardware countermeasures against DPA ? A statistical analysis of their effectiveness. In *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, pages 222–235, 2004.

[Mic15] Microsemi. WhiteboxCRYPTO: Cryptographic key hiding with tunable security and performance, 2015.

[MRP12] Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao - Lai White-Box AES Implementation. In *Selected Areas in Cryptography, 19th International Conference, SAC 2012, Windsor, ON, Canada, August 15-16, 2012, Revised Selected Papers*, pages 34–49, 2012.

[Mui13] J. A. Muir. A tutorial on white-box aes. Cryptology ePrint Archive, Report 2013/104, 2013.

[Mul14] Yoni De Mulder. White-Box Cryptography: Analysis of White-Box AES Implementations. PhD thesis, KU Leuven, 2014.

[MWP10] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a Perturbated White-Box AES Implementation. In *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings*, pages 292–310, 2010.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.

[oST01] National Institute of Standards and Technology. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, 2001.

[SMdH15] Eloi Sanfelix, Cristofaro Mune, and Job de Haas. Unboxing the White-Box Practical attacks against Obfuscated Ciphers. Black Hat Europe 2015, 2015.

[whi15] whiteCrypton. Cryptanium Overview, 2015.

[WMGP07] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In *Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*, pages 264–277, 2007.

[Wys12] Brecht Wyseur. White-box Cryptography: Hiding Keys in Software. MISC magazine, 2012.

[XL09] Yaying Xiao and Xuejia Lai. A Secure Implementation of White-box AES. In 2ndInternational Conference on Computer Science and its Applications (CSA2009), 2009.