How to Reveal the Secrets of an Obscure White-Box Implementation

Louis Goubin[4], Pascal Paillier[1], Matthieu Rivain[1], and Junwei Wang[1,2,3]*

[1] CryptoExperts
{firstname.lastname}@cryptoexperts.com

[2] University of Luxembourg

[3] University Paris 8

[4] University of Versailles-St-Quentin-en-Yvelines
louis.goubin@uvsq.fr

Abstract. White-box cryptography protects key extraction from software implementations of cryptographic primitives. It is widely deployed in DRM and mobile payment applications in which a malicious attacker might control the entire execution environment. So far, no provably secure white-box implementation of AES has been put forward, and all the published practical constructions are vulnerable to *differential computation analysis* (DCA) and *differential fault analysis* (DFA). As a consequence, the industry relies on home-made *obscure* white-box implementations based on secret designs. It is therefore of interest to investigate the achievable resistance of an AES implementation to thwart a white-box adversary in this paradigm. To this purpose, the ECRYPT CSA project has organized the WhibOx contest as the *catch the flag* challenge of CHES 2017. Researchers and engineers were invited to participate either as designers by submitting the source code of an AES-128 white-box implementation with a freely chosen key, or as breakers by trying to extract the hard-coded keys in the submitted challenges. The participants were not expected to disclose their identities or the underlying designing/attacking techniques. In the end, 94 submitted challenges were all broken and only 13 of them held more than 1 day. The strongest (in terms of surviving time) implementation, submitted by Biryukov and Udovenko, survived for 28 days (which is more than twice as much as the second strongest implementation), and it was broken by a single team, *i.e.*, the authors of the present paper, with reverse engineering and algebraic analysis. In this paper, we give a detailed description of the different steps of our cryptanalysis. We then generalize it to an attack methodology to break further obscure white-box implementations. In particular, we formalize and generalize the *linear decoding analysis* that we use to extract the key from the encoded intermediate variables of the target challenge.

1 Introduction

1.1 White-Box Cryptography

Recently, security critical applications, such as digital right management (DRM) systems and mobile payment services, have known a fast development and wide deployment on consumer electronic devices. New threats must then be considered by security designers and analysts, since these applications are usually hosted on untrusted environments and/or the users themselves might represent potential attackers. Ultimately, one has to consider an adversary that can access the software (on in particular cryptographic implementations) as a white box. Generally, she could arbitrary pick the inputs for the software and collect all the outputs and all the runtime information, such as the addresses and values of accessed memory; she could also tamper with the implementations, *e.g.*, altering the control flows and injecting faults. Cryptographic algorithms are usually involved in these contexts to assure the confidentiality, integrity and authenticity in several aspects. If a key embedded in a underlying implementation was improperly protected and extracted by the attacker, not only would the pursued security goal be lost, but also the associated business model would be threatened. For instance, an attacker could make illegal profits by selling the revealed key in a DRM application to some purchaser in the black market for a much cheaper price. Accordingly, it is reasonable to investigate her capability and the countermeasures to

* The fourth author has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 643161.

prevent key exposure. Informally, white-box cryptography seeks a solution to transform a cryptography algorithm with a given key into an obfuscated implementation that gives no significant advantage to the white-box adversary compared to the situation in which she could only access an oracle answering encryption queries (under the same key).

Historically, the white-box concept was introduced in two seminal papers by Chow *et al.* [CEJVO02, CEJv03] for cryptographic algorithms (DES and AES) used in DRM applications. The rough idea behind their constructions is to implement a cipher as a network of precomputed and randomly encoded look-up tables, such that an adversary is confused by seemingly useless intermediate values in the memory. Soon, several cryptanalyses broke the underlying techniques [JBF02, BGEC04], which motivated some remedial designs [LN05, BCD06, XL09, Kar11]. However, these proposals were eventually shown to be vulnerable as well [GMQ07, WMGP07, MWP10, MRP13a, LRM$^+$14, MRP13b, LR13].

On the other hand, not much formalization of white-box cryptography has been put forward. Two initial works [SWP09, DLPR14] has introduced some formal white-box security notions. Specifically, Saxena et al. [SWP09] demonstrate how to adapt security notions in black-box model [BGI$^+$01] into security notions in white-box model; while [DLPR14] formalizes the basic *unbreakability* property and several other useful notions: *one-wayness*, *incompressibility* and *traceability* for symmetric ciphers. But the question of how to achieve these properties for a standard symmetric cipher such as AES remains open. Nevertheless, a lot of works [BGI$^+$01, GGH$^+$13b, GGH13a, LT17] have been done on the related area of *indistinguishability obfuscation* [BGI$^+$01, GGH$^+$13b, SW14, Lin16, Lin17]. However, the current constructions of obfuscation are still impractical and the relation between white-box cryptography and indistinguishability obfuscation requires further investigation.

Because of the lack of practical and provably secure solutions, the industry tends to employ home-made solutions for applications that need to be protected against key extraction in pure software. Their security mainly relies on the secrecy of the related techniques, which contradicts with the classic Kerckhoffs's principle in cryptography. In this context, two generic approaches have been used to break such *obscure* white-box implementations. Similarly to differential power analysis (DPA) [KJJ99], *differential computation analysis* (DCA) [BHMT16] looks for correlation between key-dependent sensitive variables and *computation traces* composed of values processed in the execution of the implementation. On the other hand, since AES is inherently vulnerable to *differential fault analysis* (DFA), it can be directly applied to break a majority of the public implementations [JBF02, SMH15].

1.2 WhibOx Contest

Although no conclusions have been drawn about the pursued goals of white-box cryptography in scientific world, the development of white-box applications continues to increase. Needless to say, plenty of home-made solutions sold in the market, which are claimed to be secure based on the confidentiality of related technologies and tools, would be fragile in front of a motivated attacker. In this context, the ECRYPT CSA project organized the WhibOx workshop [whib] to fulfill the cognition of the academic progress and industrial experiences in 2016. At this occasion, it was suggested to organize a contest on white-box cryptography to give a playground for *"researchers and practitioners to confront their (secretly designed) white-box implementations to state-of-the-art attackers"* [Whia]. One year later, the so-called WhibOx competition was launched by ECRYPT CSA as the catch the flag challenge of CHES 2017.

In a nutshell, the participants of this contest were divided into two categories:

- the *designers* who were invited to submit the source codes of their white-box implementations of AES-128 [DR13] with freely chosen key, and
- the *breakers* who were challenged to reveal the hidden keys in the submitted implementations.

The participants could remain anonymous (based on a pseudonymity submission system) and they were not expected to reveal the designing or attacking techniques. The score system worked as follows: a white-box submission can accumulate $n(n+1)/2$ *strawberry* points if it survives for n days, and once it is broken, the strawberry points will decrease symmetrically down to 0. A designer gets as her final strawberry score the maximal peaking strawberries among all the challenges submitted. Similarly, a breaker gets as *banana* points the number of strawberry points of a challenge at breaking time. And she gets her final banana score as the highest banana score among all her breaks.

Table 1. Requirements for a valid implementation on a reference CPU.

C source code	$\leq$ 50MB
compilation time	$\leq$ 100s
executable binary	$\leq$ 20MB
running memory	$\leq$ 20MB
execution time	$\leq$ 1s

In the order to submit a valid challenge, the implementation must fulfill several requirements, recalled in Table 1, which are relatively looser than that in a practical scenario.

As a result, the contest successfully attracted 194 players with 94 submitted implementations which were all broken in the end for a total of 877 individual breaks. Only 13 implementations survived for more than 1 day. These results once again demonstrate that the attackers prevail in the current cat-and-mouse game. Nevertheless, there were interesting designs submitted that worth further discussion and investigation.

Adoring Poitras. The strongest implementation in terms of survival time, named Adoring Poitras,[5], was submitted by Biryukov and Udovenko from the University of Luxembourg. In the sequel, we sometimes refer to this implementation as *the challenge*. Its source code makes about 28MB. As it includes two long strings with extended ASCII characters [ISO], it takes more than 30 hours for some compilers (*e.g.* Clang) to finish the compilation.[6]

1.3 Our Contribution

This paper explains how we broke Adoring Poitras in several steps: reverse engineering, SSA transformation, circuit minimization, data dependency analysis, algebraic analysis. These different steps are detailed in Section 2. Then Section 3 gives a generalization of our break. It first depicts a general attack methodology against obscure white-box implementations and then formalizes and analyzes the *linear decoding analysis* that we used for our break (where any of our DCA or DFA attempts would fail).

2 Breaking Adoring Poitras

We explain in this section how to gradually extract the key from Adoring Poitras in a few steps. Firstly, we perform some reverse engineering on the source code to remove several obfuscation layers and obtain a Boolean circuit. Then, we rewrite the Boolean circuit into *single static assignment* (SSA) form which enables us to minimize it by detecting and removing many constant, redundant, and pseudorandom computations. Based on this minimized Boolean circuit, we conduct a data dependency analysis to identify some specific encoded operations (*e.g.*, first round AES s-boxes). Finally, we perform a generic algebraic analysis based on a linear decoding assumption which turned out to be true. From the processed (encoded) data over several executions, we are able to extract the 16 AES key bytes. Overall, it took us roughly 200 man-hours (spread over 3 weeks) to break this challenge: about one third of the time was spent on reverse engineering; another third was for data dependency analysis and minimization of the circuit; and the remaining time was for seeking possible attacks and applying our algebraic analysis. Undoubtedly, we spent a lot of time on investigating reverse engineering and attack strategies that turned out to be useless in the end. If we repeated our attack on an implementation from the same white-box compiler but for a different key and randomness, we could probably break it in a few hours (which could be dramatically improved with automatic tools). In the following sections, we will describe the above steps in detail.

Overview of Original Source Code. A summarized description of the original source code of the challenge is listed in Table 2. More specifically, it consists of 2328 lines of code, 1020 function definitions

[5] The name was generated by the sever. Source code is available at `https://whibox-contest.github.io/show/candidate/777`.

[6] Experiments are done with Apple LLVM version 9.0.0 on macOS 10.12 and clang version 3.8.1 on Alpine Linux 3.5 (the reference OS for compiling and testing).

Table 2. An overview of the source code of Adoring Poitras.

#lines	2328
#functions	1020
#global variables	12
`funcptrs` size	210
`pDeoW` size	2^{21} B
`JGNNvi` size	15 284 369 B

and 12 global variables. Most of the global variables are pointers, but one global variable is an array of 210 function pointers (`funcptrs`) and two other global variables `pDeoW` and `JGNNvi` are large arrays with numerous extended ASCII characters.

2.1 Reverse Engineering

For some reason (*e.g.*, in order to obscure the design ideas), the source code Adoring Poitras is deliberately obfuscated with several code obfuscation techniques, *e.g.*, naming obfuscation, virtualization obfuscation [Rol09]. We will go through how to unpack each obfuscation layer by reverse engineering. There is no obvious boundary between any two steps. Let us start with readability processing.

Readability Processing. The names of all the variables, functions and parameters in the original source code are obfuscated as shown below:

```
void xSnEq (uint UMNsVLp, uint KtFY, uint vzJZq) {
  if (nIlajqq () == IFWBUN (UMNsVLp, KtFY))
    EWwon (vzJZq);
}

void rNUiPyD (uint hFqeIO, uint jvXpt) {
  xkpRp[hFqeIO] = MXRIWZQ (jvXpt);
}

void cQnB (uint QRFOf, uint CoCiI, uint aLPxnn) {
  ooGoRv[(kIKfgI + QRFOf) & 97603] =
    ooGoRv[(kIKfgI + CoCiI) | 173937] & ooGoRv[(kIKfgI + aLPxnn) | 39896];
}

uint dLJT (uint RouDUC, uint TSCaTl) {
  return ooGoRv[763216 ul] | qscwtK (RouDUC + (kIKfgI << 17), TSCaTl);
}
```

Actually, only the 210 of these functions listed in the `funcptrs` are invoked in the computation, in other words, nearly 80% of defined functions are never used. Besides, all these 210 useful functions are duplicate definitions of only 20 functions. With the help of the above observation, we perform a readability processing of the original code, including:

- renaming variables, functions and parameters,
- eliminating dummies and duplicates,
- rewriting constants in a meaningful way, and
- combining codes if necessary.

Technically, most of the processing here were handled manually. In the end, we acquire a source code with 20 easily understood functions shown in the code listing below. With the help of some understanding (discussed in the following sections), these functions can be classified into several categories: input reading and output writing, bitwise operations, bit shifts, table look-ups, assignments, control flow primitives and dummy functions. We will refer to the their names in the following if necessary.

```
uint a, b;              // a is used in table lookup, b is used for updating
const uint T[] = "..."; // 2^18 uint array

/* input reading and output writing */
void read_plaintext(uint addr, uint pos)   { assign(addr, plaintext[pos]); }
void write_ciphertext(uint pos, uint addr) { ciphertext[pos] = lookup1(addr); }
void expand_bit(uint to, uint from, uint pos) { // expand bit to unsigned long integer
  T[(a + to) & 0x3ffff] = -((T[(a + from) & 0x3ffff] >> pos) & 1);
}

/* bitwise operations */
void not(uint to, uint from) {
  T[(a + to) & 0x3ffff] = ~T[(a + from) & 0x3ffff];
}
void or(uint to, uint from1, uint from2){
  T[(a + to) & 0x3ffff] = T[(a + from1) & 0x3ffff] | T[(a + from2) & 0x3ffff];
}
void xor(uint to, uint from1, uint from2){
  T[(a + to) & 0x3ffff] = T[(a + from1) & 0x3ffff] ^ T[(a + from2) & 0x3ffff];
}
void and(uint to, uint from1, uint from2){
  T[(a + to) & 0x3ffff] = T[(a + from1) & 0x3ffff] & T[(a + from2) & 0x3ffff];
}

/* bit shifts */
void right_shift_xor(uint to, uint from, uint pos) {
  if (pos > 63) // always false
    return;
  T[to & 0x3ffff] ^= T[(a + from) & 0x3ffff] >> pos;
}
void left_shift_xor(uint to, uint pos, uint from) {
  uint tmp = (T[(a + from) & 0x3ffff]) & 1;
  T[(a + to) & 0x3ffff] ^= tmp << pos;
}

/* table look-ups */
uint lookup1(uint addr)      { return T[(a + addr) & 0x3ffff]; }
uint lookup2(uint x, uint y) { return T[(x + y) & 0x3ffff]; }
void update_a() { a = lookup2(1592, (b >> 6) + ((b & 63) << 12)); }
void update_b() { b = 0x7fff & lookup2(522, (b >> 6) + ((b & 63) << 12)); }

/* assignments */
void assign_a(uint val)        { a = val; }
void assing_b(uint from)       { b = T[from] & 0x07fff; }
void assign(uint to, uint val) { T[(a + to) & 0x3ffff] = val; }
void copy(uint to, uint addr)  { assign(to, lookup1(addr - a)); }

/* control flow primitives */
void goto_func(uint pos) {               // ``goto'' in the virtual machine
  pc = bop + pos;
}
void jump_if(uint x, uint y, uint pos) { // conditional jump
  if (lookup2(2979, (b >> 6) + ((b & 63) << 12)) == lookup2(x, y))
    goto_f(pos);
}

```

```
/* dummy function */
void mistery(uint to, uint from) {
  T[(a + to) & 0x3ffff] = T[((((~T[(a + from) & 0x3ffff]) & 0x7fff) >> 6) + 2979
                            + ((((~T[(a + from) & 0x3ffff]) & 0x7fff) & 63) << 12)];
}
```

De-Virtualization. After the readability processing, the source code is much easier to understand, and we can observe that the overall program relies on a virtual machine as illustrated in the code listing hereafter, which is a common obfuscation technique in modern software protection and malwares [Rol09].

```
uint T[] = "...";              // 2^18 uint memory, renamed from pDeoW
char program[] = "...";        // 15284369 bytes, renamed from JGNNvi
void * funcptrs = {"..."};

void interpretor() {
  uchar *bop = (uchar *) program;
  uchar *eop = bop + sizeof (program) / sizeof (uchar);
  uchar *pc = bop;
  while (pc < eop) {
    uchar args_num = *pc++;
    if (args_num == 0) {
      void (*func_ptr) ();
      func_ptr = (void *) funcptrs[*pc++];
      uint *arg_arr = (uint *) pc;
      pc += args_num * 8;
      func_ptr ();
    } else if (args_num == 1) {
      void (*func_ptr) (uint);
      func_ptr = (void *) funcptrs[*pc++];
      uint *arg_arr = (uint *) pc;
      pc += args_num * 8;
      func_ptr (arg_arr[0]);
    } else if (args_num == 2) {
      void (*func_ptr) (uint, uint);
      func_ptr = (void *) funcptrs[*pc++];
      uint *arg_arr = (uint *) pc;
      pc += args_num * 8;
      func_ptr (arg_arr[0], arg_arr[1]);
    }
    // similar branches for ags_num = 3, 4, 5, 6
  }
}

void AES_128_encrypt(uchar * ciphertext, uchar * plaintext) {
  interpretor();
}
```

Specifically, the authors of the challenge implemented a virtual environment with an interpreter of a bytecode program. The program is a sequence of instructions, each of which is either a conditional jump to a previous instruction or a function call written in the following format:[7]

`[number of arguments][function pointer index][argument list]`,

[7] In fact, the conditional jump is is also implemented as a function in the same format (see `goto_func` and `jump_if` functions above). Particularly, it is used for simulating the `do ... while` loop in a high-level language, where the first two arguments are used for condition checking and the third arguments is the destination.

Code 1 Structure of the bitwise program

Input: plaintext bits $(b_1, b_2, \cdots, b_{128})$, unsigned long integer table T of length 2^{18} with initial values
Output: ciphertext bits $(c_1, c_2, \cdots, c_{128})$

```
for i = 1 to 128 do
    T[addr_{1,i}] ← −b_i                          ▷ equivalent to expand b_i to unsigned long integer
    for j = 1 to 63 do
        T[addr_{1,i} + j * 2^12 mod 2^18] ← T[addr_{1,i}]
    end for
end for

BitwiseOperationLoop1                             ▷ see Code 2
BitwiseOperationLoop2
...
BitwiseOperationLoop2573

BitCombination                                    ▷ see Code 3

BitwiseOperationLoop2574
...
BitwiseOperationLoop2582

for i = 1 to 128 do
    c_i ← T[addr_{2,i}]
end for
```

where `[number of arguments]` is one byte indicating the number of arguments, `[function pointer index]` is one byte giving the index of the called function within the array of function pointers (*i.e.* the global variable `funcptrs`), and `[argument list]` is the sequence of arguments, each taking eight bytes. In the runtime, the interpreter loads an instruction, then translates it into a function call with corresponding arguments.

In order to remove this virtualization layer, we construct a new equivalent program in C language by simulating the interpreter. In detail, after the decoding of all the instructions, we rewrite the conditional jumps as do ... while loops, and construct function calls with their arguments from the bytecode program. We thus get a C program composed of do ... while loops and some calls to the 20 useful functions with hard-coded arguments.

Simplification of the Bitwise Program. The overall structure of the bitwise program is shown in Code 1. The default data type is unsigned 64-bit integer (`uint`). The program contains a globally-accessible table T (renamed from `pDeoW`) of 2^{18} 64-bit words (*i.e.*, 2^{21} bytes) initialized to some hard coded values. In the beginning of the program, each bit b_i is expanded to a full word (by the operation $-b_i \mod 2^{64}$) which is assigned to some location $\mathsf{addr}_{i,1}$ in T. Then, each expanded bit $T[\mathsf{addr}_{i,1}]$ is copied to 63 locations $\mathsf{addr}_{i,1}^{(1)}, \mathsf{addr}_{i,1}^{(2)}, \cdots, \mathsf{addr}_{i,1}^{(63)}$ in the table, where

$$\mathsf{addr}_{i,1}^{(n)} = \mathsf{addr}_{i,1} + 2^{12} \cdot n \mod 2^{18}.$$

Then the program performs a sequence of 2573 bitwise operation loops, followed by one bit combination loop (pictured in Code 3 below), then by 9 additional bitwise operation loops. The bit combination loop is the only one to involve bit shifts. In comparison, bitwise operation loops only perform bitwise operations (*i.e.*, binary operations applied in parallel to each bit slot of 64-bit operands). In the end, the program outputs each ciphertext bit from a different location $\mathsf{addr}_{2,i}$ in table T.

Loops before BitCombination. Through basic debugging methods, we observe that the bitwise operation loops are each composed of 64 iterations performing up to 504 statements (except the very last loop which has 2051 statements). The basic structure of these loops is depicted in Code 2 hereafter. A statement simply consists in a bitwise operation (xor, or, and, not) with one or two operands picked from

different locations in the table T. The result of the bitwise operation is stored at another location in T. We denote by $\{\mathsf{addr}_1, \mathsf{addr}_2, \cdots, \mathsf{addr}_N\}$ the accessing address sequence, namely, the locations read and written in table T by the statements (in chronologic order) in the first round of loop.

Code 2 Example of a bitwise operation loop

for $i = 0$ to 63 **do**
  $j \leftarrow P(i)$ ▷ P is a permutation of $\{0, 1, \cdots, 63\}$ and $P(0) = 0$
  $T[\mathsf{addr}_3 + j * 2^{12} \mod 2^{18}] \leftarrow T[\mathsf{addr}_1 + j * 2^{12} \mod 2^{18}] \oplus T[\mathsf{addr}_2 + j * 2^{12} \mod 2^{18}]$
  $T[\mathsf{addr}_5 + j * 2^{12} \mod 2^{18}] \leftarrow T[\mathsf{addr}_3 + j * 2^{12} \mod 2^{18}] \wedge T[\mathsf{addr}_4 + j * 2^{12} \mod 2^{18}]$
  $T[\mathsf{addr}_8 + j * 2^{12} \mod 2^{18}] \leftarrow T[\mathsf{addr}_6 + j * 2^{12} \mod 2^{18}] \vee T[\mathsf{addr}_7 + j * 2^{12} \mod 2^{18}]$
  $T[\mathsf{addr}_9 + j * 2^{12} \mod 2^{18}] \leftarrow \neg T[\mathsf{addr}_8 + j * 2^{12} \mod 2^{18}]$
  ⋮
end for

All these addresses are computed from a global variable `a` which is updated in each loop iteration using a second global variable `b` and an update mechanism as follows:

```
int a, b; // global variables

assign_b(219964);
do{
  update_a();
  // bitwise operations
  // ...
  // ...
  update_b();
 } while(lookup2(2979,(b>>6)+((b&63)<<12))!=lookup2(815257, 237931));
```

Let us denote by a_0, a_1, ..., a_{63}, the successive values taken by the global variable `a` in the 64 iterations, so that the ith instruction $\mathsf{addr}_i = a_j + c_i$ in iteration j, where c_i is constant and $0 \leq j \leq 63$. By inspecting the sequence of a_j's, we observe that it satisfies

$$a_j = a_0 + p_j \cdot 2^{12} \mod 2^{18} , \tag{1}$$

where $p_j \in \{0, 1, \ldots, 63\}$ for every j. Moreover, a closer inspection shows that $p_j = P(j)$ for some permutation P defined over $\{0, 1, \ldots, 63\}$. We did not try to understand whether there was some underlying mathematical principle in P (beyond the fact it is a permutation).

At this point, we aim to identify some properties of these loops that would reveal some structure in the program. One interesting observation is that for some loops, there exist $1 \leq i, j \leq N$ and $i \neq j$ such that addr_i is a reading address, and addr_j is a writing address, and $\mathsf{addr}_i \equiv \mathsf{addr}_j \mod 2^{12}$ (that is $c_i \equiv c_j \mod 2^{12}$). This implies that some memory locations are both read and written during the loop execution. Such loops are said to be *overlapping*; the other loops are said to be *non-overlapping*. There are 1020 overlapping loops and 1562 non-overlapping loops in the program. Besides, there is no isolated (non)-overlapping loop in the program. With this observation, the programs is divided into 27 parts, each of which only consists of either overlapping or non-overlapping loops. In the beginning, we thought this partition was related to the AES round operations, but we did not extract any useful information out of this observation.

Afterwards, by inspecting some arbitrary overlapping loop, we can observe that its inner statements simply consist in some swaps between memory locations in the table T. These swaps are implemented through different sequences of bitwise operations. A sample code is listed in Appendix A.1. Moreover we can further observe that two swapped addresses are always equivalent modulo 2^{12}. More noteworthy, these swaps seemed useless with respect to the functional correctness of the program. We thus obtain our first simplified program by removing all overlapping loops (except for the BitsCombintaion discussed in the next paragraph). We believe the simplified code is functionally equivalent to the original program since their outputs always match on many randomly chosen inputs. Furthermore, since the remaining

loops are non-overlapping (*i.e.* all the written memory locations are not used during the execution of the current loop), the permutation P can be replaced with the identity function (*i.e.*, $P(j) = j, 0 \geq j \geq 63$). Or even better, we can rewrite the do ... while loop as a for loop from 0 to 63. We again verify our conjecture by comparing the program outputs before and after modification for a large number of encryptions (of random plaintexts). Now we acquire a new simpler version in which the permutations before BitCombination are all removed.

Code 3 BitCombination (reconstructed for comprehension)

```
for ℓ = 1 to 129 do
    T[addr_{3,ℓ}] ← v_ℓ                                          ▷ v_ℓ ∈ GF(2) is a constant
    for j = 1 to 64 do
        T[addr_{3,ℓ}] ← T[addr_{3,ℓ}] ⊕ Parity(T[addr_{4,ℓ} + j * 2^12  mod 2^18])
        T[addr_{3,ℓ}] ← T[addr_{3,ℓ}] ⊕ Parity(T[addr_{5,ℓ} + j * 2^12  mod 2^18])
    end for
end for

Parity(x)                                                        (the number of 1-bits in x modulo 2)
    r ← 0
    for i = 0 to 63 do
        r ← r ⊕ (x ≫ i)&1
    end for
    return r
```

BitCombination *and the remaining loops.* Code 3 illustrates how BitCombination works. It first assigns 129 locations $(\mathsf{addr}_{3,\ell})_{1\leq\ell\leq129}$ in T with Boolean constants (namely either `0x00...00` or `0x00...01`). Then each of these table locations is further xor-ed with the parity bits (each of which is computed through 64 simple instructions,) of 128 different values stored in $\mathsf{addr}_{4,\ell} + j * 2^{12} \mod 2^{18}$ and $\mathsf{addr}_{5,\ell} + j * 2^{12} \mod 2^{18}$, for some addresses $\mathsf{addr}_{4,\ell}$ and $\mathsf{addr}_{5,\ell}$ and for $1 \leq j \leq 64$. The 129 64-bit words output from BitCombination are hence Boolean variables. Moreover, after the remaining loops, all the ciphertext bits are the least significant bits of some specific 64-bit words in T. Therefore, we deduce that only the least significant bits of the remaining computations after BitCombination take effects in the outputs, *i.e.*, everything happening after BitCombination can be seen as a Boolean circuit.

Besides, we observe that only a single iteration in the last bitwise operation loop affects the output ciphertext, which means that we can replace this loop by a single iteration (for a given value of the loop index i). Then we can reiterate this observation with the loop before, and so on until the BitCombination loop. In the end, the operations after BitCombination is simplified as a Boolean circuit made of one iteration of each former loop.

Entire Transformation to a Boolean Circuit. Similar observations and conjectures can be applied to the loops before BitCombination. Specifically, observing that all the operations are bitwise and that any two bits in different positions of the operands never communicate with each other until BitCombination, we conjecture that

(1) the ith bit of the intermediate values in the jth loop iteration corresponds to one independent *partial* AES computation (*i.e.* not complete without the operations after BitCombination),
(2) only one (or odd number of) such independent computation(s) in 64*64 of them is (are) real.

To verify this conjecture, we tried to execute BitCombination while skipping one bit index $1 \leq i \leq 64$ in the parity computation for one loop index $1 \leq j \leq 64$. For three pairs (i, j), we observed the 129 outputs of BitCombination were constant to 0 over several plaintexts. We deduced that real AES computations are performed in the ith bit slot of the jth iteration for $(i, j) \in \{(42, 26), (58, 32), (10, 48)\}$ before BitCombination. Therefore, we can simplify the code by picking any single separate AES computation and verify our guess in the usual way. Accordingly, the bitwise program is fully transformed into a Boolean circuit.

2.2 Single Static Assignment Form

Although we get a Boolean circuit, we still lack knowledge about how it works, *e.g.*, where each round is computed. As in a typical unpacking story, we perform some static and dynamic analyses to acquire more information. In the current representation, many intermediate variables are both written and read several times, which presumably hides some facts on the data flow. In compiler theory, a program in *single static assignment* (SSA) form means that every variable is assigned (defined or written) once, but can be read for multiple times after its assignment. (The memory used in a SSA formatted program is then about its number of instructions.) The SSA form of a program thus looses the data dependency by reducing the meaningless interlaced dependences introduced by variable reuse. In order to transform our Boolean circuit into SSA form, we rewrite through the few following steps:

1. Declare a global counter $c = 0$, and an empty associative map (hashmap) H.
2. For each statement, replace
   a) each its reading address(es) $\texttt{addr}_r$ with $H(\texttt{addr}_r)$,
   b) and its writing address $\texttt{addr}_w$ with c,

   then we set $H(\texttt{addr}_w) = c$ and $c = c + 1$.

After this transformation, the program is in SSA form: every memory location is written exactly once and only read after its assignment.

2.3 Boolean Circuit Minimization

After SSA transformation, we attempt to minimize the program in several aspects. Our goal here is to decrease the computation complexity in the subsequent analysis techniques that will then target a smaller circuit. We define a few minimization steps (described below) and we iterate over these steps several times until we cannot reduce it any more.

Detection and removal of constants. We execute the Boolean circuit for a large (*e.g.*, 2048) number of times with randomly sampled inputs and record the *computation traces* (which consist of the ordered sequences of written values). Then, for each location in these computation traces, we check if the written value is always the same. Formally, denoting ith computation trace by $(v_1^{(i)}, v_2^{(i)}, \cdots, v_t^{(i)})$, where t is the size of the trace (*i.e.* the number of Boolean instructions), we check whether

$$v_j^{(1)} = v_j^{(2)} = \cdots = v_j^{(N)} = c \in \{0, 1\},$$

for some index j and for sufficiently large N. If so, we consider that the jth instruction calculates a constant and we replace the corresponding variable by the constant c. We then propagate this constant according to the following Boolean relations:

$$\begin{aligned} v \wedge 0 = 0,\ v \wedge 1 = v, \\ v \vee 0 = v,\ v \vee 1 = 1, \\ v \oplus 0 = v,\ v \oplus 1 = \neg v, \end{aligned} \tag{2}$$

where $v \in \{0, 1\}$. This propagation results in the saving of further instructions.

In an idealized model where all the variables are uniformly distributed, the probability of false judgement is 2^{-N}. The complexity to perform the detection is of $\mathcal{O}(N \cdot t)$.

Detection and removal of duplicates. We proceed in a similar way as above to detect and remove duplicates. Namely, we observe whether for two locations in the computation traces the written values are always the same. Formally, we check whether

$$(v_{j_1}^{(1)} = v_{j_2}^{(1)}) \wedge (v_{j_1}^{(2)} = v_{j_2}^{(2)}) \wedge \cdots \wedge (v_{j_1}^{(N)} = v_{j_2}^{(N)}),$$

for some pair of indexes (j_1, j_2) and for sufficiently large N. If so, we consider that the related statements are duplicated computations and that the j_1th and j_2th variables are a pair of duplicates. Then we remove one of the instance and replace all its apparitions in the program by the other variable.

As above, the probability of false judgement in a idealized model is of 2^{-N}. The complexity to perform the detection is of $\mathcal{O}(N \cdot t^2)$.

Detection of Boolean inverse. The detection of Boolean inverse is similar to the detection of duplicates but instead, we check whether

$$(v_{j_1}^{(1)} = \neg v_{j_2}^{(1)}) \wedge (v_{j_1}^{(2)} = \neg v_{j_2}^{(2)}) \wedge \cdots \wedge (v_{j_1}^{(N)} = \neg v_{j_2}^{(N)}) \ ,$$

for some pair of indexes (j_1, j_2) and for sufficiently large N. If so, we can replace the statement computing v_{j_2} by a simple NOT instruction on input v_{j_1} (assuming $j_1 < j_2$), which is likely to induce further simplifications while looping on the minimization steps.

Detection and removal of pseudorandomness. Here we look for *pseudorandom* which are variables used to randomize subsequent intermediate results without affecting the final result. In order to check whether an intermediate variable serves as pseudorandom, we try to flip its value and check whether the output always matches the output in a normal execution. Formally, denoting x_i and y_i the input and output of the ith execution, we flip the jth variable by inserting a statement $v_j = \neg v_j$ right after the assignment of v_j. Then we check whether

$$(y_1 = y_1') \wedge (y_2 = y_2') \wedge \cdots \wedge (y_N = y_N'),$$

where y_i' denotes the output of the execution with the flipping statement on input x_i. If so, we consider v_j to be some pseudorandomness and we replace it by a constant, *e.g.*, 0. This constant is then propagated as described above which results in the saving of further instructions.

The probability of false judgement is not clear but is should quickly becomes negligible as N grows (as v_j might affect several bits of the output). The complexity to perform the detection is of $\mathcal{O}(N \cdot t)$.

Remark 1. A variable might impact the output result and be used as pseudorandomness at the same time. In the above detection, we can only detect the variables solely for pseudorandomness. Rather to flip an intermediate variable, a more effective way is to flip an operand in a statement. In this sense, the flippable operand corresponds to a pseudorandom usage of the variable and it can be replaced by a constant.

Detection and removal of dead (dummy) code. A dead statement is an instruction writing a value which is never used in the subsequent computation. Dead might be introduced by the above minimization steps or by the removal of subsequent dead code. The detection and removal process is a progressive iteration procedure.

Application to Adoring Poitras. We apply these minimization steps to reduce the Boolean circuit recovered after the reverse engineering of Adoring Poitras. We apply each step between 2 and 5 times except for the removal of dummy variables that is applied a dozen of times. We obtain a minimized circuit of 280K gates (Boolean instructions), which is half of the original size.

2.4 Data Dependency Analysis

A visual way to analyze data dependency of a circuit is to plot its *data dependency graph* (DDG), a directed acyclic graph (DAG) in which a vertex stands for an intermediate variable (an address in T in our case) and a directed edge means a variable (ending vertex) is computed from another variable (starting vertex). We extract and plot data dependency graph of our minimized circuit using Mathematica.[8] Specifically, for each statement in the minimized circuit, we first generate one/two directed edges from the addresses of its operands to the address of its destination; then we get an ordered sequence of edges according to the order in which the relevant gates appear in the circuit. Then we invoke the `Graph` function of Mathematica with the sequence of edges to plot the DDG. At first, we attempt to plot a figure for the whole DDG, but fail since it is too costly to produce such a large graph for Mathematica with a standard computer. Then we try to plot some smaller part of the circuit DDG, starting with the first 20% which looks like a mess as shown in the left of Figure 1. Afterwards, we try plotting the first 10% of the DDG as shown in the right of Figure 1, but we cannot still extract too much valuable information except that we observe some kind of symmetry as illustrated by the red line on the figure.

[8] See `https://www.wolfram.com/mathematica/`.

We keep going and plot the 5% of the DDG as represented in Figure 2 which reveals much more structure than our previous observations. A mysterious "ball" is located in the center of the graph, which is mainly composed of the first edges (*i.e.* the beginning of the circuit), and 16 "branches" come out from this central ball, divided into four groups for which the four branches eventually join. The plotted circuit starts from the center and ends with flake structures. Seemingly, the beginning of the circuit has a highly complex data dependency and the variables inside are deeply mixed together and then extensively used in the future computation since our minimization process cannot get rid of them.

Extracting S-Box Encodings. Based on our knowledge of the AES structure, we make the heuristic assumption that the "branches" correspond to the 16 s-box computations in the first round of AES which are then mixed four by four through the MixColumns operations.

If our assumption is correct, the set of outgoing variables of a branch (*i.e.* the set of variables computed inside the branch and which are used later in the program) must be an encoding of the output s-box value. In order to extract the set of outgoing variables, we apply modularity-based clustering algorithms [New04] to the data dependency graph. Specifically, we apply the Mathematica function `FindGraphCommunities` to the first 5% of the DDG. The graph is then divided into several communities (clusters) in a way that the vertices in the same community have a denser connection than a set of vertices from different communities. This way, we can isolate each "branch" in Figure 2 and obtain the corresponding set of vertices from which we extract the set of outgoing variables. Note that in practice, the clustering algorithm was not necessarily applied the first 5% of the DDG but a tuning over the search window was manually applied (see details in Table 3 below). The number of vertices in the recovered clusters is between 439 and 615 per cluster, and the number of outgoing variables scales from 29 to 57.

At this step we have 16 sets of variables which are presumably 16 encodings of the first round s-box outputs. We now explain how we could break these encodings and recovered the corresponding secret key bytes.

2.5 Algebraic Analysis

Let us denote by v_1, v_2, ..., v_t, the t outgoing (binary) variables of an s-box cluster, that presumably encode an s-box output. Let us denote by x the plaintext byte and by k^* the secret key byte corresponding to this s-box computation. Then, if our data dependency analysis is correct (namely if the v_i's indeed encode the s-box output), there exists a deterministic decoding function $\mathsf{dec} : \{0,1\}^t \to \{0,1\}^8$ satisfying:

$$\mathsf{dec} : (v_1, v_2, \ldots, v_t) \mapsto \big(\mathsf{Sbox}(x \oplus k^*)[0], \ldots, \mathsf{Sbox}(x \oplus k^*)[7]\big) \tag{3}$$

where $\mathsf{Sbox}(\cdot)[j]$ denotes the jth Boolean coordinate function of the AES s-box.

Our algebraic analysis works by assuming that dec is linear (actually affine) over $\mathsf{GF}(2)$. As we show hereafter, this is enough to break Adoring Poitras but it can be generalized to higher degree decoding functions (see Section 3). This linear decoding assumption specifically states that for each output coordinate $j \in \{0, 1, \ldots, 7\}$, there exists a constant vector $\mathbf{a} = (a_0, a_1, a_2, \cdots, a_t) \in \mathsf{GF}(2)^{t+1}$ such that

$$a_0 \oplus \bigoplus_{i=1}^{t} a_i \cdot v_i = \mathsf{Sbox}(x \oplus k^*)[j] \ . \tag{4}$$

Note that the coefficients a_i are different for each output coordinate but we avoid an additional index for the sake of clarity. In other words, the jth output bit of the s-box is encoded by a simple Boolean sharing and its shares are distributed among the v_i variables according to the a_i coefficients: if $a_i = 1$ then v_i is a share of $\mathsf{Sbox}(x \oplus k^*)[j]$ and if $a_i = 0$ then $\mathsf{Sbox}(x \oplus k^*)[j]$ is independent of v_i.

To validate our assumption, we collect a set of N *computation traces* for the presumed s-box encoding $(v_1, v_2, \ldots, v_t)$. That is, we execute the white-box implementation N times with random palintexts and record the values $(v_1^{(i)}, v_2^{(i)}, \ldots, v_t^{(i)})$, $1 \le i \le N$, taken by the encoding variables for these N executions. Then we iterate over the 256 possible key guesses k for the 16 possible s-box positions and try to solve

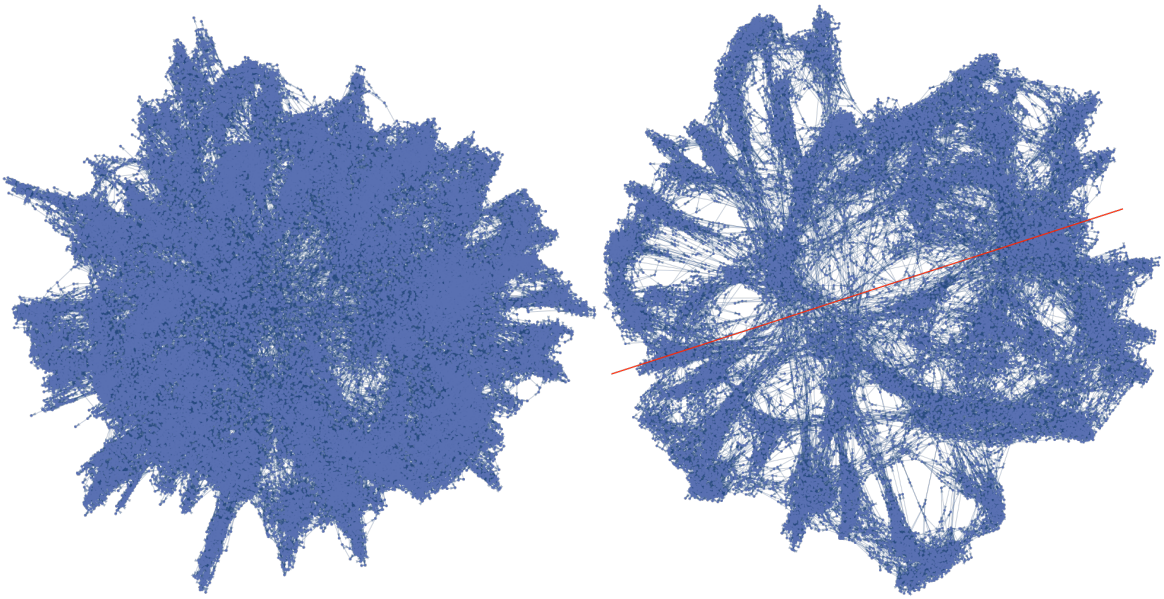

Fig. 1. The data dependency graph for the 20% (left) / 10% (right) edges plotted by Mathematica.

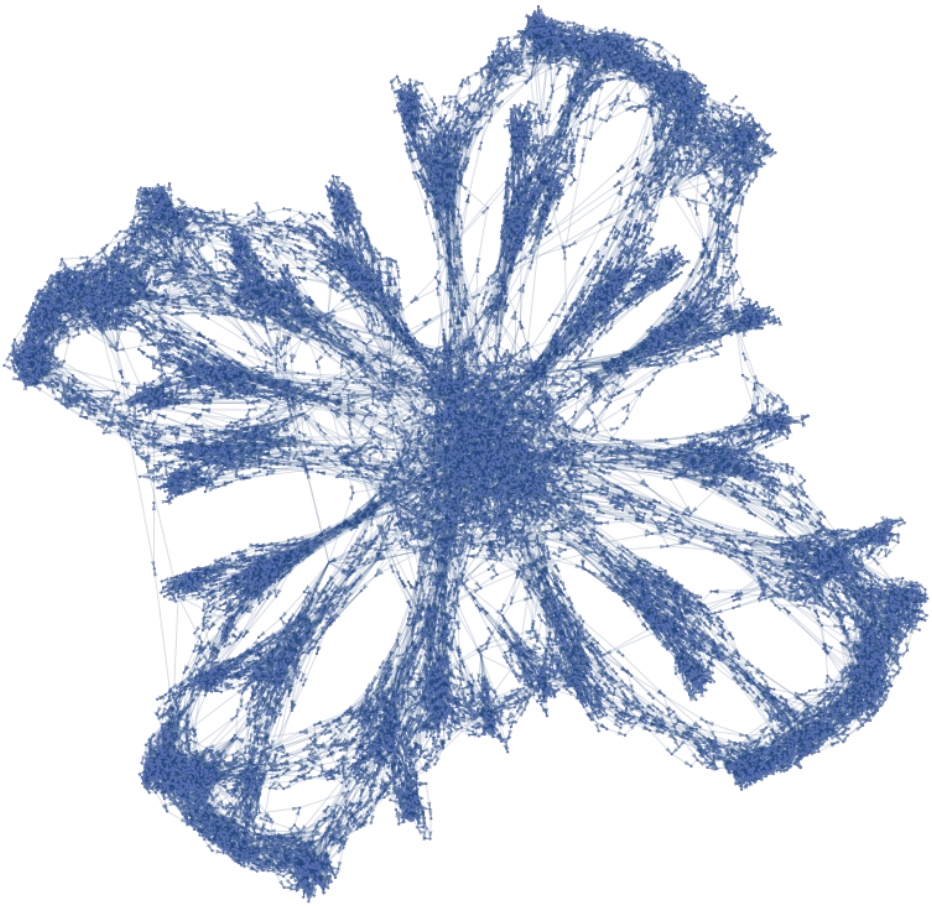

Fig. 2. The data dependency graph for the 5% codes plotted by Mathematica.

the following system of linear equations (with a_0, a_1, ..., a_t as unknowns):

$$\begin{bmatrix} 1 & v_1^{(1)} & v_2^{(1)} & \cdots & v_t^{(1)} \\ 1 & v_1^{(2)} & v_2^{(2)} & \cdots & v_t^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_1^{(N)} & v_2^{(N)} & \cdots & v_t^{(N)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_t \end{bmatrix} = \begin{bmatrix} \mathsf{Sbox}(x^{(1)} \oplus k)[j] \\ \mathsf{Sbox}(x^{(2)} \oplus k)[j] \\ \vdots \\ \mathsf{Sbox}(x^{(N)} \oplus k)[j] \end{bmatrix}, \tag{5}$$

where $x^{(i)}$ denote the values taken by the plaintext byte x in the ith execution. If our linear decoding assumption is true, then the above system is solvable for the right s-box position and the right key guess $k = k^*$, which directly follows from (4), and the solution reveals the decoding function dec. On the other hand, for an incorrect key guess, the chance to solve the system quickly becomes negligible as the number of traces N increases above t, which will be formally discussed in Section 3.

Remark 2. Note that the selection of the outgoing variables v_1, v_2, ..., v_t (which are basically the fringe edges of a cluster) is crucial for this attack to work. When a single one happens to be missing then the system becomes unsolvable. This stresses the importance of a sound clustering step for the subsequent success of this attack.

Practical Results. We perform the above algebraic analysis based on our linear decoding assumption to extract the key from our minimized Boolean circuit. For each presumed s-box cluster, we extract the outgoing variables and record a set of computation traces. Thanks to the data dependecy analysis (and the clustering step) described above, the number t of outgoing variables is never more than a few dozens (specifically at most 59). Moreover, we use up to $N = 100$ computation traces, which overall yields some linear systems of dimensions lower than 80×100 solvable within a few microseconds on a desktop computer.

For each cluster, we try to solve the linear systems obtained for all the pairs (k, j) (key guess and s-box coordinate), and all the 16 s-box positions. For most clusters, all the 8 systems obtained for a single s-box position and a single key guess are solvable whereas the others are unsolvable (giving a strong presumption that we have found the correct key byte). For one cluster, less than 8 systems are solvable, but still for a single s-box position and a single key byte. And for a few other clusters, no system is solvable at all. The two latter cases occur as a consequence of a wrong cluster selection (see Remark 2). In these cases, we have to fine-tune the clustering step by varying the range of the input edges to eventually get some solvable systems (each time for a single key guess). After recovering 14 out of 16 key bytes, we exhaust the remaining ones (the 6th and 12th) by brute-force search[9] (over a plaintext-ciphertext pair computed with the white-box implementation) and finally recover the full AES key.

Table 3 depicts our practical results in details. For each of the 16 s-boxes (but the 6th and the 12th for which we use exhaustive search) it gives the range of edges in the DDG used for clustering, the number of vertices (or variables) in the extracted cluster, the corresponding number of outgoing variables (parameter t), the number of Boolean shares in the encoding of each s-box output bit (*i.e.* the Hamming weight of the coefficient vector $\boldsymbol{a}$), and the recovered key byte. Note that for the 8th s-box we cannot solve the 8 systems corresponding to the right key guess but only 3 of them (which explains '?' for the number of shares).

For instance, for the third s-box, we can extract a cluster with 530 variables in the edges ranging between 4000 and 13500 and among which 34 are outgoing variables. For this cluster we can solve the 8 linear systems. For further illustration, Table 4 exhibits the solutions of these 8 systems, where the encoding coefficients are ordered chronologically. We observe that only 15 consecutive variables of the 34 outgoing variables are used as Boolean shares to encode the 8 output bits of the s-box. Moreover some of these variables are involved as shares for more than one output bit of the s-box. In other words, the decoding function is a 15-bit to 8-bit linear mapping.

[9] We could probably extract these bytes through the algebraic analysis as well, but it was faster to search exhaustively.

Table 3. Clustering and algebraic analysis results..

s-box	edge range	#cluster	#outgoing vars (t)	#shares (HW($\boldsymbol{a}$))	key byte
1	9,500 - 18,000	541	30	{8,6,7,5,8,3,3,7}	0x71
2	4,000 - 18,000	543	29	{7,7,6,9,8,7,7,8}	0x3c
3	4,000 - 13,500	530	34	{8,10,8,8,6,2,6,4}	0xcf
4	9,500 - 18,000	515	38	{6,9,8,6,6,11,9,9}	0x9f
5	9,500 - 20,000	571	41	{8,6,6,4,4,9,8,10}	0x27
6	n/a	n/a	n/a	n/a	0x45
7	9,500 - 20,000	615	42	{2,11,5,6,7,10,3,8}	0xe5
8	9,500 - 24,000	500	59	{?,10,11,?,?,14,?,?}	0xbc
9	9,500 - 20,000	448	57	{4,6,7,8,7,6,6,12}	0x04
10	9,500 - 18,000	568	36	{8,6,6,6,6,12,6,8}	0x64
11	4,000 - 18,000	523	35	{9,5,7,9,3,3,8,7}	0xb9
12	n/a	n/a	n/a	n/a	0x07
13	9,500 - 18,000	514	30	{8,5,4,7,5,5,5,6}	0x78
14	9,500 - 20,000	454	45	{14,9,13,12,14,15,10,16}	0xf4
15	9,500 - 18,000	505	30	{8,6,6,8,8,7,4,8}	0x77
16	9,500 - 18,000	439	49	{6,8,8,8,4,6,10,6}	0x07

Table 4. The solution of the system of equations for each bit in the third byte.

bit	encoding coefficients
1	0,0,0,0,0,0,**1,0,1,0,1,1,1,0,0,0,1,0,1,0,1**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2	0,0,0,0,0,0,**1,0,0,1,1,0,1,1,1,1,1,1,1,0,0**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
3	0,0,0,0,0,0,**0,0,1,0,1,0,0,0,1,1,1,0,1,1,1**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4	0,0,0,0,0,0,**0,0,0,1,1,0,0,0,1,1,1,0,1,1,1**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
5	0,0,0,0,0,0,**0,1,1,0,0,1,0,0,0,0,0,0,1,1,1**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
5	0,0,0,0,0,0,**0,0,0,0,1,0,0,0,0,0,0,0,0,0,1**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
7	0,0,0,0,0,0,**1,0,0,0,1,0,0,1,0,1,0,1,0,1,0**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
8	0,0,0,0,0,0,**0,1,0,0,0,0,1,0,0,1,1,0,0,0,0**,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

3 Generalization

3.1 A White-Box Attack Methodology

We present hereafter a general attack methodology to break obscure white-box implementations following the outline of our cryptanalysis of Adoring Poitras. This methodology is organized into the five following steps. Note that depending on the white-box implementation, it might not be necessary to apply all these steps.

1. *Initial reverse engineering.* The targeted implementation is usually protected by several obfuscation techniques (as *e.g.* described in [CTL97]). This first step consists in removing these obfuscation layers, either manually or by using some automatic tools [YJWD15]. The goal is to understand the role of each part of the code, and remove any virtualization. Of course, this step is difficult to fully generalize and automatize. It should somehow rely on some human handwork and intuition. The ultimate goal, for the next steps of our methodology, is to transform the implementation into an arithmetic circuit (or a Boolean circuit as particular case). Namely, this first step must produce a straight line program (*i.e.* without conditional branching) in which every instruction is of the form $v_i \leftarrow v_j * v_\ell$ for some operation $*$ lying in a defined set of operations. For instance, in the Boolean case we would have $* \in \{\oplus, \wedge, \vee\}$. But a white-box implementation could be defined over a larger finite field (such as $\mathsf{GF}(2^n)$ or $\mathsf{GF}(p)$), an integer/polynomial ring, etc. An arithmetic circuit would then be composed of additions and multiplications. But some more complicated operations could occur and in all generality, which could be represented, *e.g.*, by look-up tables, taking possibly more than two input operands.
2. *SSA transformation.* The arithmetic circuit is then rewritten into SSA form, in which each variable is only assigned once and accessed after its assignment. The SSA transformation is depicted in Section 2.2.

3. *Circuit minimization.* Our minimization techniques described in Section 2.3 are generic and they can be easily extended to any algebraic structure beyond the Boolean case. Specifically, we can detect removable intermediate variables, including constants, duplicates, pseudorandomness and dummy variables, by executing the implementation with a large number of randomly sampled plaintexts (along with flipping variables for detecting pseudorandomness). Then we can replace the pseudorandomness by 0's, remove duplicates and dummy variables and propagate the constants according to the different operations. The circuit minimization is an iterative process and should be conducted for several rounds.
4. *Data dependency analysis.* In order to extract the key from a white-box implementation, it is usual to focus on some specific early round operations, *e.g.*, the first round s-boxes in a block cipher. Observing the DDG is very insightful to locate a given operation depending on the structure of the target cryptographic algorithm. This step can be partly automated through a cluster analysis (though in our breaking of Adoring Poitras, the visual inspection of the DDG was necessary to parameterize the clustering). An alternative approach is to try different windows of intermediate variables which can be fully automated but this approach is likely to substantially increase the attack complexity compared to an accurate localization of the target operation. Once the target operation has been localized (or for each guessed location), we identify the corresponding set of outgoing variables which presumably constitutes an encoding of the target variable.
5. *Algebraic analysis.* This last step consists in extracting some key information by analyzing the (presumed) encoding obtained from the data dependency analysis. To this purpose, we generalize and formalize hereafter the algebraic analysis previously described in the Section 3.2. But this step could alternatively rely on further attack techniques such as, *e.g.*, differential computation analysis (DCA) or differential fault analysis (DFA) [BHMT16, SMH15].

3.2 Linear Decoding Analysis

We formalize the algebraic analysis described in the Section 3.2 which we shall call *linear decoding analysis* (LDA). An LDA attacker against a white-box implementation can extract the key information contained in a set of encoded intermediate variables, provided that the underlying plain variable can be recovered through a linear decoding.

Without loss of generality, we assume that the white-box implementation processes intermediate variables (that can be represented) on some finite field $\mathbb{F}$. Typically $\mathbb{F} = \mathsf{GF}(2)$ for a Boolean circuit, but we could have $\mathbb{F} = \mathsf{GF}(2^{32})$ for a 32-bit architecture program, or more generally $\mathbb{F} = \mathsf{GF}(q)$ for any prime (power) q. Let us denote $s = \varphi(x, k^*) \in \mathbb{F}$ the target sensitive variable where φ is a deterministic function, $k^* \in \mathcal{K}$ is a subkey for some subkey space $\mathcal{K}$, and x is a part of the input plaintext (or output ciphertext).

Similar to a DCA adversary, an LDA adversary controls a white-box implementation and she can execute it for several plaintexts and dynamically record the corresponding *computation traces*. These traces consist of ordered t-tuples $\boldsymbol{v} = (v_1, v_2, \cdots, v_t)$ of the values taken by the intermediate variables (*e.g.*, values read/stored in memory, results of CPU instructions, etc.), where $v_i \in \mathbb{F}$ for every i. As discussed above, these computation traces might be related to a small part of the full execution, *e.g.*, when targeting a specific operation either localized by data dependency analysis or guessed using an automated search. The adversary collects N such computation traces $\boldsymbol{v}^{(i)} = (v_1^{(i)}, v_2^{(i)}, \cdots, v_t^{(i)})$ that correspond to N (chosen) plaintexts $x^{(i)}$ for $1 \leq i \leq N$. Then, for every key guess $k \in \mathcal{K}$, she constructs the following system of linear equations:

$$\begin{bmatrix} 1 & v_1^{(1)} & v_2^{(1)} & \cdots & v_t^{(1)} \\ 1 & v_1^{(2)} & v_2^{(2)} & \cdots & v_t^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_1^{(N)} & v_2^{(N)} & \cdots & v_t^{(N)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_t \end{bmatrix} = \begin{bmatrix} \varphi(x^{(1)}, k) \\ \varphi(x^{(2)}, k) \\ \vdots \\ \varphi(x^{(N)}, k) \end{bmatrix}, \tag{6}$$

where $(a_0, a_1, a_2, \cdots, a_t)$ are the unknown coefficients in $\mathbb{F}$. If the system is unsolvable for every key guess k, then the attack fails. If the system is solvable for a single key guess k, there is a strong presumption that it is the right key guess *i.e.* $k = k^*$, the adversary then returns k as the (candidate) correct key.

For N sufficiently greater than t, if the above system is solvable, it means that the target intermediate variables satisfy

$$a_0 + \sum_{i=1}^{t} a_i \cdot v_i = \varphi(x, k) \ . \tag{7}$$

Namely, the white-box implementation encodes the sensitive variable s in the v_i's through the above (decoding) relation. In particular the variables $\{v_i; a_i \neq 0\}$ form a linear sharing of s. We stress that such encoding encompasses any kind of Boolean masking or linear secret sharing of *any order* (see for instance [ISW03, RP10, Bei11]). Moreover, the encoding function is not necessarily linear: one would basically generates the masks (or the shares) pseudorandomly from the full input plaintext p, implying that the encoding function $\mathsf{enc} : (p, k^*) \mapsto (v_1, v_2, \cdots, v_t)$ could be of high degree in p, whereas the decoding function $\mathsf{dec} : (v_1, v_2, \cdots, v_t) \mapsto s = \varphi(x, k^*)$ is linear.

Complexity. LDA has complexity $\mathcal{O}(|\mathcal{K}| \cdot t^{2.8})$. For each key guess $k \in \mathcal{K}$, the attack can be split into two phases: first solve a linear system of $t+1$ equations in $t+1$ variables (we assume that the corresponding square matrix is full rank without loss of generality), and then check whether the $N-(t+1)$ equations match the recovered solution. The complexity of the first phase is $\mathcal{O}(t^{2.8})$ by using the Strassen algorithm [Str69].[10] The second phase is then of complexity $\mathcal{O}(t \cdot (N-t))$ which is negligible compared to the first phase since, as shown in Section 3.3, a high success probability can be obtained by taking a (small) constant number of additional traces $N-t$. We thus obtain a total complexity of $\mathcal{O}(|\mathcal{K}| \cdot t^{2.8})$ for the recovery of one subkey $k^* \in \mathcal{K}$.

Window Search. When the adversary is not able to accurately localized the target encoding among the intermediate variables then he might apply LDA to the full computational trace (i.e. the computational trace of the full execution). If we denote by τ the size of this full trace, then the obtained complexity is of $\mathcal{O}(|\mathcal{K}| \cdot \tau^{2.8})$, which might be too huge. For instance this would have made about 2^{59} operations for a trace of size $\tau \approx 280$K as obtained for the Adoring Poitras minimized circuit before data dependency analysis (see Section 2.3).

In practice, one can significantly improve this complexity by searching the potential encoding variables in a relatively small window of the computation trace. In a practical white-box implementation, the computation for some specific (encoded) intermediate result, has some *locality* property that the related intermediate variables are located in a t-size subtrace of the full τ-size computation trace. Formally, in a full computation trace $(v_1, v_2, \cdots, v_\tau)$, t consecutive points $(v_{i+1}, v_{i+2}, \cdots, v_{i+t})$, for some index i, contain all variables to decode the target sensitive variable s. Without knowing the locality parameter t and the right position i in the full trace, the adversary can try LDA for several t and i. Specifically, we suggest to apply LDA on the subtrace obtained for every $i \in \{1, 2, \ldots, \tau - t\}$ for an increasing $t = 2^1, 2^2, 2^3, \ldots$ The total complexity is then of $\mathcal{O}(|\mathcal{K}| \cdot \tau\, t^{2.8})$, where t is the right locality parameter, which is better than the full-trace attack complexity whenever $t < \tau^{0.64}$.

3.3 Analysis of LDA

The soundness of LDA results from the fact that if a decoding relation such as (7) does exist for the target intermediate variable s, and if the shares are well selected in the computation trace $\boldsymbol{v} = (v_1, v_2, \cdots, v_t)$, then LDA will solve the system for the right key guess k^*. For a wrong key guess, on the other hand, no solution should be find unless (1) φ is a linear function w.r.t. the field $\mathbb{F}$, or (2) an encoding $\varphi(x, k)$ is computed by the implementation for a wrong key guess $k^\times \neq k^*$ (with the purpose of fooling the attacker). These two limitations can simply be mitigated: (1) can be avoided by targeting an appropriate intermediate result (such as an s-box output), and it is unlikely that (2) occurs for all the possible subkeys $k \in \mathcal{K}$ which would arguably represent a huge computational overhead for the implementation (and would become intractable as we go deeper in the computation).

We analyze hereafter the success probability of LDA under the following assumptions:

[10] This could theoretically be reduced to $\mathcal{O}(t^{2.376})$ using the Coppersmith–Winograd algorithm for very large t (see for instance [GVL96]) but in practice one shall prefer the Strassen algorithm.

– a linear decoding relation (such as (7)) does exist between $\boldsymbol{v}$ and s,
– the plaintext (part) x is uniformly distributed,
– $\boldsymbol{v}$ is uniformly distributed among the t-tuples satisfying the decoding relation $a_0+\sum_i a_i \cdot v_i = \varphi(x, k^*)$,

The two first assumptions are necessary conditions of the LDA attack context which are arguably satisfied in some real white-box design/attack use cases (as typically considered in this paper). The last assumption is *ideal* and is not necessary for LDA to work but only for the purpose of our formal analysis. It could somehow be relaxed by considering potential statistical dependences between the variables which would complicate the analysis without strongly impacting the result.

Proposition 1. *Under the above assumptions, the probability that the LDA linear system* (13) *is solvable for an incorrect key guess $k^\times \neq k^*$ is lower than $|q|^{N-t-1}$, where*

$$q \stackrel{\text{def}}{=} \max\left\{\Pr\left(\varphi(X, k^*) = \alpha \cdot \varphi(X, k^\times)\right) \ ; \ \alpha \in \mathbb{F}^*, \ (k^*, k^\times) \in \mathcal{K}^2\right\}. \tag{8}$$

for a uniform distribution of X.

Proof. Without loss of generality, we assume that there exists a subsystem $\mathcal{S}$ containing $t+1$ equations from (13) such that the corresponding matrix is full-rank (implying that $\mathcal{S}$ has one and only one solution whatever the target vector).[11] The solution of $\mathcal{S}$ is denoted $\boldsymbol{a}^* = (a_0^*, a_1^*, \cdots, a_t^*)$ for the correct key guess k^* and $\boldsymbol{a}^\times = (a_0^\times, a_1^\times, \cdots, a_t^\times)$ for the wrong key guess $k^\times$. In the following we will consider that the $t+1$ equations in $\mathcal{S}$ are the $t+1$ first equations of the system. Then, two possible cases occur:

1. There exists a constant $\alpha \in \mathbb{F}$ such that $\boldsymbol{a}^\times = \alpha \cdot \boldsymbol{a}^*$. This implies that

$$\varphi(x^{(i)}, k^\times) = \alpha \cdot \varphi(x^{(i)}, k^*) \ , \tag{9}$$

   for every $1 \leq i \leq t+1$. Moreover, the full system has a solution for the guess $k^\times$ if and only if (9) is further satisfied for every $i \in \{t+2, \ldots, N\}$. Since the $x^{(i)}$ are uniformly distributed, this happens with probability at most $q^{N-(t+1)}$.
2. There does not exist a constant $\alpha \in \mathbb{F}$ such that $\boldsymbol{a}^\times = \alpha \cdot \boldsymbol{a}^*$. In that case, from our ideal assumption, we have

$$a_0^\times + \sum_{j=1}^{N} a_j^\times \cdot v_j^{(i)} \sim \mathcal{U}(\mathbb{F}) \ ,$$

   (where $\mathcal{U}(\mathbb{F})$ denotes the uniform distribution over $\mathbb{F}$) for every $i \in \{t+2, \ldots, N\}$. Then the full system has a solution for the guess $k^\times$ if and only if

$$a_0^\times + \sum_{j=1}^{N} a_j^\times \cdot v_j^{(i)} = \varphi(x^{(i)}, k^\times)$$

   is satisfied for every $i \in \{t+2, \ldots, N\}$, which occurs with probability $(\frac{1}{|\mathbb{F}|})^{N-(t+1)} < q^{N-(t+1)}$.

□

By Proposition 1, the probability that the system (13) is solvable for the incorrect key guess $k^\times$ is exponentially small in N. In practice, an appropriately chosen φ makes q close to $\frac{1}{|\mathbb{F}|}$ and the probability quickly becomes negligible as N grows over $t+1$. Moreover, the number of extra traces required to get a given (negligible) probability of false positive depends on the target function φ, but is constant with respect to t.

As an illustration, if the target variable is a first-round s-box of AES, then

– for the Boolean case ($\mathbb{F} = \mathsf{GF}(2)$) where $\varphi(k, x) = \mathsf{Sbox}(k, x)[j]$ for some j, we obtain $q = \frac{9}{16}$ and taking, *e.g.*, 40 extra equations makes the false-positive probability lower than 2^{-32};
– for the full field case ($\mathbb{F} = \mathsf{GF}(256)$) where $\varphi(k, x) = \mathsf{Sbox}(k, x)$, we obtain $q = \frac{7}{256}$ and taking , *e.g.*, 7 extra equations makes the false-positive probability lower than 2^{-32}.

[11] According to our three assumptions, the probability that there does not exist any full rank subsystem containing $t+1$ equations is negligible.

3.4 Extension to Higher Degrees

The linear decoding assumption necessary to LDA might not be satisfied in practice for some white-box implementations. Depending on the algebraic structure of the encoding scheme used to protect intermediate variables, the decoding function might have an algebraic degree greater than 1. We explain in this section how LDA can be generalized to break implementations with higher degree decoding functions. This generalization shall be called *higher-degree decoding analysis* (HDDA) in the following.

For each collected computation trace $\boldsymbol{v}$, the HDDA adversary computes the *monomial trace* defined as:

$$\boldsymbol{w} = (1) \parallel \boldsymbol{v} \parallel \boldsymbol{v}^2 \parallel \cdots \parallel \boldsymbol{v}^d \tag{10}$$

where $\parallel$ is the concatenation operator and where $\boldsymbol{v}^j$ is the vector of degree-j monomials:

$$\boldsymbol{v}^j = (v_{i_1} \cdot v_{i_2} \cdot \ \ldots \ \cdot v_{i_j})_{1 \leq i_1 \leq i_2 \leq \cdots \leq i_j \leq t} \ . \tag{11}$$

The size of the vector $\boldsymbol{v}^j$ is the number of degree-j monomials in t variables, which equals $\binom{j+t-1}{j}$. The size of the monomial trace is the number of monomials of degree lower than or equal to d, which is

$$t' = \sum_{j=0}^{d} \binom{j+t-1}{j} = \binom{t+d}{d} \leq \frac{(t+d)^d}{d!} \ll t^d \ . \tag{12}$$

From the computation traces obtained for N executions (with random input plaintext), the adversary computes N such monomial traces $\boldsymbol{w}^{(i)} = (w_1^{(i)}, w_2^{(i)}, \cdots, w_t^{(i)})$. Then, for every key guess $k \in \mathcal{K}$, she constructs the linear system:

$$\begin{bmatrix} 1 & w_1^{(1)} & w_2^{(1)} & \cdots & w_{t'}^{(1)} \\ 1 & w_1^{(2)} & w_2^{(2)} & \cdots & w_{t'}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_1^{(N)} & w_2^{(N)} & \cdots & w_{t'}^{(N)} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{t'} \end{bmatrix} = \begin{bmatrix} \varphi(x^{(1)}, k) \\ \varphi(x^{(2)}, k) \\ \vdots \\ \varphi(x^{(N)}, k) \end{bmatrix}, \tag{13}$$

where $(a_0, a_1, a_2, \cdots, a_{t'})$ are the unknown coefficients in $\mathbb{F}$.

If the above system is solvable for N sufficiently greater than t' then (with overwhelming probability) there exists a degree-d decoding function dec (with the a_i's as coefficients) such that

$$\mathsf{dec}(v_1, v_2, \ldots, v_t) = \varphi(x, k) \ . \tag{14}$$

In particular, if the white-box encoding of the sensitive variable $s = \varphi(x, k^*)$ can be decoded with a degree-d function and if the shares of the encoding are well included in the computation trace, then the above system will be solvable for $k = k^*$ and the solution will give the right decoding function.

On the other hand, and as for the LDA case (*i.e.* the case $d = 1$) analyzed above, the probability that the system is solvable for a wrong key guess $k \neq k^*$ quickly becomes negligible as N increases (over t'), provided that there exists no degree-d relation between $\varphi(\cdot, k)$ and $\varphi(\cdot, k^*)$ (in particular φ is of degree greater than d).

Complexity. Following the complexity analysis of Section 3.2, HDDA has complexity $\mathcal{O}(|\mathcal{K}| \cdot t'^{2.8})$. For a small constant d, this makes a complexity of $\mathcal{O}(|\mathcal{K}| \cdot t^{2.8d})$. The complexity of HDDA with window search in a computation trace of size τ with an (unknown) locality parameter of t is then of $\mathcal{O}(|\mathcal{K}| \cdot \tau \, t^{2.8d})$.

4 Conclusion

In this paper, we have explained how we could break the winning challenge (presumably the hardest) in the recent WhibOx contest. This was done in several steps mixing reverse engineering, circuit minimization techniques, data dependency analysis and algebraic analysis. In a second part, we have generalized this cryptanalysis in a generic attack methodology against obscure white-box implementations and a powerful algebraic attack against any kind of encodings with a low-degree decoding function. The latter

requires to collect some computation traces as DCA, but it can efficiently break encodings of *any order* (*i.e.* whatever the number of shares) where DCA wouldn't work (or higher-order DCA would probably have a very high complexity). Our work makes a step towards a systematic analysis of obscure white-bow implementations and challenges the approach of using obscurity to build security in the context of white-box cryptography.

References

BCD06. Julien Bringer, Herve Chabanne, and Emmanuelle Dottax. White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468, 2006. `http://eprint.iacr.org/2006/468`.

Bei11. Amos Beimel. Secret-Sharing Schemes: A Survey. In Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology - Third International Workshop, IWCC 2011, Qingdao, China, May 30-June 3, 2011. Proceedings*, volume 6639 of *Lecture Notes in Computer Science*, pages 11–46. Springer, 2011.

BGEC04. Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In Helena Handschuh and Anwar Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 227–240. Springer, Heidelberg, August 2004.

BGI+01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18. Springer, Heidelberg, August 2001.

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, *CHES 2016*, volume 9813 of *LNCS*, pages 215–236. Springer, Heidelberg, August 2016.

CEJv03. Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In Kaisa Nyberg and Howard M. Heys, editors, *SAC 2002*, volume 2595 of *LNCS*, pages 250–270. Springer, Heidelberg, August 2003.

CEJVO02. Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. A white-box des implementation for drm applications. In *Digital Rights Management Workshop*, volume 2696, pages 1–15. Springer, 2002.

CTL97. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

DLPR14. Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 247–264. Springer, Heidelberg, August 2014.

DR13. Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

GGH13a. Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 1–17. Springer, Heidelberg, May 2013.

GGH+13b. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

GMQ07. Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of white box DES implementations. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 278–295. Springer, Heidelberg, August 2007.

GVL96. G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996.

ISO. ISO/IEC 8859-1:1998: Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1. `https://www.iso.org/standard/28245.html`. Accessed: October 2017.

ISW03. Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Heidelberg, August 2003.

JBF02. Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an obfuscated cipher by injecting faults. In *Digital Rights Management Workshop*, volume 2696, pages 16–31. Springer, 2002.

Kar11. Mohamed Karroumi. Protecting white-box AES with dual ciphers. In Kyung Hyune Rhee and DaeHun Nyang, editors, *ICISC 10*, volume 6829 of *LNCS*, pages 278–291. Springer, Heidelberg, December 2011.

KJJ99. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, August 1999.

Lin16. Huijia Lin. Indistinguishability obfuscation from constant-degree graded encoding schemes. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 28–57. Springer, Heidelberg, May 2016.

Lin17. Huijia Lin. Indistinguishability obfuscation from SXDH on 5-linear maps and locality-5 PRGs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 599–629. Springer, Heidelberg, August 2017.

LN05. H. E. Link and W. D. Neumann. Clarifying obfuscation: improving the security of white-box des. In *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, volume 1, pages 679–684 Vol. 1, April 2005.

LR13. Tancrède Lepoint and Matthieu Rivain. Another nail in the coffin of white-box AES implementations. Cryptology ePrint Archive, Report 2013/455, 2013. `http://eprint.iacr.org/2013/455`.

LRM+14. Tancrède Lepoint, Matthieu Rivain, Yoni De Mulder, Peter Roelse, and Bart Preneel. Two attacks on a white-box AES implementation. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 265–285. Springer, Heidelberg, August 2014.

LT17. Huijia Lin and Stefano Tessaro. Indistinguishability obfuscation from trilinear maps and block-wise local PRGs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 630–660. Springer, Heidelberg, August 2017.

MRP13a. Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In Lars R. Knudsen and Huapeng Wu, editors, *SAC 2012*, volume 7707 of *LNCS*, pages 34–49. Springer, Heidelberg, August 2013.

MRP13b. Yoni De Mulder, Peter Roelse, and Bart Preneel. Revisiting the BGE attack on a white-box AES implementation. Cryptology ePrint Archive, Report 2013/450, 2013. `http://eprint.iacr.org/2013/450`.

MWP10. Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a perturbated white-box AES implementation. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 292–310. Springer, Heidelberg, December 2010.

New04. M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev. E*, 69:066133, Jun 2004.

Rol09. Rolf Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

RP10. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, Heidelberg, August 2010.

SMH15. Eloi Sanfelix, Cristofaro Mune, and Job de Haas. Unboxing the White-Box - Practical attacks against Obfuscated Ciphers . `https://www.blackhat.com/docs/eu-15/materials/eu-15-Sanfelix-Unboxing-The-White-Box-Practical-Attacks-Against-Obfuscated-Ciphers-wp.pdf`, 2015. Accessed: October 2017.

Str69. Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, August 1969.

SW14. Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014.

SWP09. Amitabh Saxena, Brecht Wyseur, and Bart Preneel. Towards security notions for white-box cryptography. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC 2009*, volume 5735 of *LNCS*, pages 49–58. Springer, Heidelberg, September 2009.

Whia. CHES 2017 Capture the Flag Challenge - The WhibOx Contest, An ECRYPT White-Box Cryptography Competition. `https://whibox.cr.yp.to/`. Accessed: October 2017.

whib. WhibOx 2016 - White-Box Cryptography and Obfuscation. `https://www.cryptoexperts.com/whibox2016/`. Accessed: October 2017.

WMGP07. Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In Carlisle M. Adams, Ali Miri, and Michael J. Wiener, editors, *SAC 2007*, volume 4876 of *LNCS*, pages 264–277. Springer, Heidelberg, August 2007.

XL09. Yaying Xiao and Xuejia Lai. A secure implementation of white-box aes. In *Computer Science and its Applications, 2009. CSA'09. 2nd International Conference on*, pages 1–6. IEEE, 2009.

YJWD15. Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691. IEEE Computer Society Press, May 2015.

A Code Listings

A.1 Swapping in Overlapping Loops

Here is a code segment to show swapping implementation in two different ways by using bitwise operations. The operands indicates the address in table T. The first operand is for the result, while the remaining ones are for the inputs.

```
// swapping values in T[248329] and T[178697] where 248329 = 178697 mod 2^12
not(225586, 248329);
not( 99382, 178697);
not(125856,  99382);
xor( 13816, 225586,  99382);
xor( 33114,  99382, 225586);
not( 20933,  13816);
not(188758, 225586);
not(180239,  33114);
 or(261865, 180239, 133397);
 or( 94096,  20933, 133397);
xor(201945, 261865, 125856);
xor(  3792,  94096, 188758);
not(248329,   3792);
not(178697, 201945);

// swapping values in T[92413] and T[22781] where 92413 = 22781 mod 2^12
not( 24583,  92413);
not(146257,  22781);
xor( 67653, 146257, 133397);
xor(234702,  24583, 133397);
 or(181444,  24583, 133397);
and(172013, 234702,  24583);
 or(110852, 172013, 146257);
and(248606, 110852, 181444);
 or( 79222, 146257, 133397);
and(146881,  67653, 146257);
 or( 86050, 146881,  24583);
and( 44767,  86050,  79222);
not( 92413,  44767);
not( 22781, 248606);
```