Cryptanalysis of a White Box AES Implementation

Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi[⋆]

France Télécom R&D
38–40, rue du Général Leclerc
92794 Issy les Moulineaux Cedex 9 — France
{olivier.billet, henri.gilbert}@francetelecom.com
charaf_echchatbi@yahoo.fr

Abstract. The white box attack context as described in [1, 2] is the common setting where cryptographic software is executed in an untrusted environment—i.e. an attacker has gained access to the implementation of cryptographic algorithms, and can observe or manipulate the dynamic execution of whole or part of the algorithms. In this paper, we present an efficient practical attack against the obfuscated AES implementation [1] proposed at SAC 2002 as a means to protect AES software operated in the white box context against key exposure. We explain in details how to extract the whole AES secret key embedded in such a white box AES implementation, with negligible memory and worst time complexity 2^{30}.

Keywords: white box, AES, block ciphers, tamper resistance, software piracy, implementation.

1 Introduction

One of the consequences of the ever spreading use of cryptology within mass applications—e.g. email, web servers access, digital content distribution, and so on—implemented in software on standard terminals, like PCs, PDAs, or mobile phones, is that cryptologic algorithms are quite often executed in an untrusted environment. The usual "black box" model, where keys and cryptographic algorithms are confined and executed in a logically protected and tamper resistant cryptographic module, like a smart card, is no longer applicable. This situation motivated the introduction of a new setting, coined "white box" context of execution: the software representing cryptographic algorithms, cryptographic keys when separate from the cryptographic software, and dynamic data produced during the execution of all or part of the cryptographic algorithms, are exposed to being accessed or even manipulated by malicious processes hosted by the same machine—which may be controlled either by an outsider or by the legitimate user of the host terminal. Cryptographic applications running in the white box context of execution are highly vulnerable to the most severe form

[⋆] Work performed at France Télécom R&D

of attack, namely the leakage of the cryptographic keys. Thus, the protection cryptographic algorithms would offer in the black box model of execution vanish.

This security issue is at the origin of the introduction, in a pair of seminal articles [2, 1] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot, of a new protection technique preventing from key leakage for cryptographic software run in the white box context. It consists in implementing key-instantiated versions of an algorithm, as the composition of a series of lookup tables, each look-up table concealing some components of the algorithm. Implementations of an algorithm resulting from this protection technique are named white-box implementations. White box implementations of the DES and AES blockciphers were respectively described in [2] and [1]. Short after the publication of [2], it was shown by M. Jacob, D. Boneh and D. Felten in [3], that the obfuscation technique applied in [2] was insecure, i.e. that a low complexity attack requiring few accesses (with partly chosen input values) to lookup tables representing external DES rounds, allowed to extract the key from a white box DES implementation. However, the attack technique of [3] is not applicable to the white box implementation of AES described in [1] due to the additional protection provided by some extra features introduced by [1]. More precisely, a fundamental difference between both implementations results from the application, in the case of AES, of so-called external encodings. One of the main security consequences of this extra feature—which description is provided in Sec. 2—is that in the case of AES, and unlike DES, the protection of external rounds is not weaker than the protection of internal rounds. Since the attack strategy of [3] is essentially based upon the extra weakness of external rounds, it is not applicable to the AES implementation described in [1]. To the best of our knowledge, no realistic attack against the white box implementation of [1] has been proposed so far.

In this paper, we present a practical low complexity attack—i.e. with negligible memory, and work factor $3 \cdot 2^{28} < 2^{30}$—of the AES white box implementation proposed in [1]. The conducting idea of the attack is that though none of the lookup tables, when considered individually, leaks sensitive information related to the AES key in an obvious way, the analysis (based on the observation of related input/output values) of lookup tables composition, reveals information on the encodings embedded in those lookup tables. We show that the information provided by the analysis of such tables during three consecutive encoded rounds, allows an attacker to entirely recover the AES 128-bit secret key of an obfuscated AES implementation. The key steps of the proposed attack were successfully implemented in `C++`, and confirmed by computer experiments.

This paper is organized as follows. In Section 2, we describe the white box AES implementation as proposed by [1]. In Section 3 we show how to extract the secret key. The last section concludes the paper.

2 Description of the White Box AES Implementation

We now describe the implementation proposed in [1]. The general strategy is to merge several steps of the AES round function into table lookups, blended by input/output encodings, and mixing bijections.

Internal encodings (resp. mixing bijections) are non-linear (resp. GF(2)-linear) and introduce confusion (resp. diffusion) in the representation of the intermediate blocks of the computation. Their inclusion in the implementation must respect the fact that two consecutive tables in the data flow have matching output and input encodings, as well as matching mixing bijections, at their boundary.

Apart from the above pairwise canceling internal transformations, another obfuscation technique called external encoding is used. It consists in feeding the obfuscated implementation with AES inputs in an encoded form. At the same time, the implementation also outputs the AES encrypted values in an encoded form. Thus, the implementation does not exactly achieve an AES computation $Y = E_K(X)$, but a modified computation $Y = E'_K(X) = G \circ E_K \circ F^{-1}(X)$. The external input/output encodings G and F^{-1} have to be annihilated on the peer site—e.g. a server when the AES obfuscated implementation is embedded in a software player—in order to compute ${E'}_K^{-1}$. Though the encodings G and F^{-1} suggested in [1] are hereafter taken into account, our attack is not highly dependent upon their exact specification. One of the main consequences of using external encodings is that internal input/output encodings can be used to blend the first and last round, in addition to inner rounds' blending. This prevents attackers from exploiting specific weaknesses one would otherwise encounter against external rounds of obfuscated implementations [3].

Let us hereafter denote AES-128 the AES version operating on 128 bits blocks. Recall [4, 5] that the AES-128 round function is made of the four steps described in Fig. 1 operating on the 16 bytes of a 4×4 state array. The AES-128

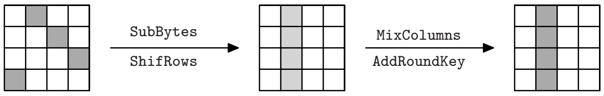

Fig. 1. tracking four bytes during an AES round

considered in [1] consists of 10 such rounds; a preliminary `AddRoundKey` step is performed before the first round, and `MixColumns` is omitted in the final round. Let us index the state bytes by their row and column numbers (i, j) in the state array. If the S-box function operating on bytes during the `SubBytes` step is denoted by S, define for any round r and any byte (i, j) with indexes taken modulo 4:

$$1 \le r \le 9 \qquad T^r_{i,j}(x) := S\left(x \oplus k^r_{i,j}\right) \ ,$$
$$T^{10}_{i,j}(x) := S\left(x \oplus k^{10}_{i,j}\right) \oplus k^{11}_{i,j-i} \ .$$

(Note that we shifted the round index of the original AES-128 by 1, and that the post-whitening key $k^{11}_{i,j}$ occuring in the last round is absorbed by the definition of the last function $T^{10}_{i,j}$.) Now each 4-byte column of the output of the `SubByte` plus `ShiftRows` steps will contribute to the 4-byte column of the state array after `MixColumns`, and those four bytes are related to the former by a 32×8

submatrix MC_i of the 32×32 matrix MC representing `MixColumns`. Now the entire function can be described by a lookup table. However, it is necessary to obfuscate this table, which leads to encode its 4-bit input and output nibbles—using concatenated non-linear permutations $\boxed{\text{in}}$ and $\boxed{\text{out}}$ respectively.

To add to the diffusion, 8×8 affine "mixing" bijection is inserted before $T^r_{i,j}$ and a 32×32 affine bijection MB is inserted after the `MixColumn` part. The resulting lookup table is depicted in Fig. 2 as the `sub` table. The 32×8 linear mapping of Fig. 2 is associated with $\mathrm{MB} \times \mathrm{MC}_i$.

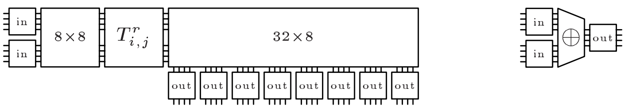

Fig. 2. `sub` table (type II) and `xor` table (type IV)

To cancel the effect of MB, a lookup table takes care of the inversion. However, instead of constructing a huge table for the entire 32×32 matrix, the mapping MB^{-1} is split into four submatrices $\left(\mathrm{MB}^{-1}\right)_i$, just like with the `MixColumns` matrix MC. This results in the lookup table depicted in Fig. 3.

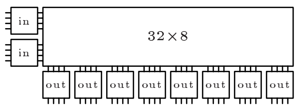

Fig. 3. `untwist` table (type III)

Finally, external input and output encodings are implemented, using two sets of sixteen 8-bit to 128-bit lookup tables depicted in Fig. 4. Each external input encoding table represents the linear mapping associated with one 128×8 vertical stripe of a 128×128 matrix—the composition of M_F and the concatenation of the input mixing bijections for $T^1_{i,j}$'s inverses—surrounded by 4-bit to 4-bit non-linear encodings. Each external output encoding table represents one 128×8 vertical stripe of a 128×128 parasitic matrix—the composition of one round 10's output mixing bijection's inverse, one of the mappings $T^{10}_{i,j}$, and 128×8 vertical stripe of a 128×128 parasitic matrix M_G—surrounded by 4-bit to 4-bit non linear encodings. The outputs of the 16 `external input encoding` tables have to be decoded, xored together and reencoded to complete the implementation. This is done by using 15×32 additional `xor` tables per 128-block. The same number of `xor` tables is needed to support the 16 `extern_encode` tables.

Thus, in order to implement a white box instance of AES-128 associated with a key K, $9 \cdot 4 \cdot 4$ `sub` tables, $9 \cdot 4 \cdot 4$ `untwist` tables, $9 \cdot 4 \cdot 3 \cdot 8$ `xor` tables supporting `sub` tables, $9 \cdot 4 \cdot 3 \cdot 8$ `xor` tables supporting `untwist` tables, $2 \cdot 16$ `extern_encode` tables, and $2 \cdot 15 \cdot 32$ `xor` tables supporting `extern_encode` tables are needed. Therefore, the total size of lookup tables in an AES-128 white box implementation is 770 048 bytes.

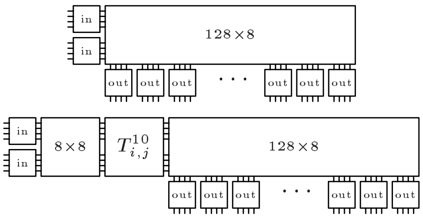

Fig. 4. `extern_encode` tables for input and output respectively (type I)

3 Cryptanalysis of the White Box AES Implementation

We now describe a very efficient attack against the white box AES implementation of [1]. The leading idea is that, though recovering information about the key by a local inspection of the lookup tables seems difficult—lookup tables were designed to satisfy so-called diversity and ambiguity criteria—recovering information by analyzing compositions of lookup tables corresponding to one encoded AES round is easier. More precisely, it is convenient to analyze each of the four mappings between four bytes of the input state array, and the four corresponding bytes of the output state array, which together form an encoded AES round. Each such mapping can be conceptualized by the box in Fig. 5, where we can choose inputs and observe outputs, whereas intermediate values remain concealed. Let us denote this box by R^r_j. Each R^r_j box is made of four 8-bit to 8-bit parasitic input permutations $P^r_{i,j}$ (resp. output permutations $Q^r_{i,j}$) constructed as the composition of two concatenated 4-bit to 4-bit input (resp. output) encodings, and one 8-bit to 8-bit linear mixing bijection. Due to the fact that internal input encodings plus linear mixing bijections and linear mixing bijections plus output encodings mutually cancel out at the boundary between two rounds r and $r+1$, each $Q^r_{i,j}$ is the inverse of $P^{r+1}_{i,j}$.

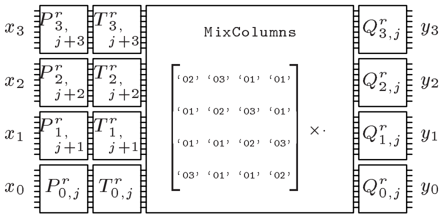

Fig. 5. One of the four R^r_j mappings, $j = 0, \ldots, 3$

The attack proceeds in three steps. First of all, we recover the non-affine part of the parasites Q^r_i in round $r = 1, \ldots, 9$, i.e. we determine Q^r_i up to unknown affine bijections, and thus get at the same time the non-affine part of the inverse P^{r+1}_i of round $r+1$, $r = 1, \ldots, 9$. At this stage we are in the setting depicted in Fig. 5, but this time the permutations P_i and Q_i are now GF(2)-affine, except for the permutation $P^1_{i,j}$ whose non-affine part has not been determined. In a second step, we recover those GF(2)-affine mappings (but $P^1_{i,j}$ and $Q^1_{i,j}$), first up to an unknown GF(2^8)-affine bijection, and then entirely. Eventually combining all this information in a third step, we extract the AES-128 key.

3.1 Recovering Non-linear Parts

Consider the mapping R^r_j. We are trying to remove the non-linearity in the parasites $(Q^r_i)_{i=0,\dots,3}$. To this end consider y_0 as a function of (x_0, x_1, x_2, x_3), and fix the values of x_1, x_2, and x_3 to some constants, say c_1, c_2, and c_3. One easily checks that there exists two constants in GF(2^8), namely α independent of c_1, c_2, c_3, and β_{c_1,c_2,c_3}, such that

$$y_0(x, c_1, c_2, c_3) = Q^r_{0,j}\left(\alpha T^r_{0,j}\left(P^r_{0,j}(x)\right) \oplus \beta_{c_1,c_2,c_3}\right) \ .$$

Since x only takes 256 values, those mappings are known by input/output, as well as their inverses. Also, varying one constant (say c_3) into the whole GF(2^8), and keeping the other one fixed, has the effect that β_{c_1,c_2,c'_3} takes all the values in GF(2^8). We are thus able to produce—as lookup tables, of course—all the functions

$$y_0(x, c_1, c_2, c_3) \circ y_0(x, c_1, c_2, c'_3)^{-1} = Q_0\left(Q_0^{-1}(x) \oplus \beta\right) \ , \qquad (1)$$

where $\beta = \beta_{c_1,c_2,c'_3} \oplus \beta_{c_1,c_2,c_3}$ takes all the values in GF(2^8). This leads to the problem of recovering Q_0, or at least its non-linear part from the set of all those lookup tables. Note that since functions are given as lookup tables, we are not provided with the underlying translations: we only know the unordered set of functions corresponding to the 256 translations. As this problem is of independent interest, we state it, along with a solution, in a standalone context.

Theorem 1. *Given a set of functions $\mathcal{S} = \{Q \circ \oplus_\beta \circ Q^{-1}\}_{\beta \in \mathrm{GF}(2^8)}$ given by values, where Q is a permutation of* GF(2^8) *and $\oplus_\beta$ is the translation by β in* GF(2^8)*, one can construct a particular solution $\widetilde{Q}$ such that there exists an affine mapping A so that $\widetilde{Q} = Q \circ A$.*

Proof. There is an isomorphism between the commutative groups $\left(\mathrm{GF}(2)^8, \oplus\right)$ and $(\mathcal{S}, \circ)$, given by

$$\begin{array}{rccc} \varphi : & \mathcal{S} & \longrightarrow & \mathrm{GF}(2)^8 \\ & Q \circ \oplus_\beta \circ Q^{-1} & \longmapsto & [\beta] \ , \end{array}$$

where $[\beta]$ denotes the embedding of the element β into the vector space GF$(2)^8$ with canonical base $([e_i])_{i=1,\dots,8}$. The issue is we do not know this isomorphism. The general idea of the proof is to recover this isomorphism up to an unknown linear bijection, i.e. to recover a known isomorphism ψ equal to φ up to an unknown linear bijection. To this end, first select from $\mathcal{S}$ a tuple $(f_1, \dots, f_8)$ of 8 functions such that their images through φ constitute a base of GF$(2)^8$. Although we do not know φ—and thus the underlying translations $[\beta_i] = \varphi(f_i)$

for each $f_i = Q \circ \oplus_{\beta_i} \circ Q^{-1}$—this can easily be done by gradually selecting f_1 to f_8 so that they span the whole set $\mathcal{S}$ through composition, that is

$$\forall f \in \mathcal{S}, \quad \exists!(\varepsilon_1, \ldots, \varepsilon_8) \in \{0,1\}^8, \qquad f = f_8^{\varepsilon_8} \circ f_7^{\varepsilon_7} \circ \cdots \circ f_1^{\varepsilon_1} \ , \tag{2}$$

where $f_i^1 = f_i$ and f_i^0 denotes the identity function. An efficient algorithm to compute such a tuple of functions $(f_1, \ldots, f_8)$ is described at the end of this paragraph.

Now since $([\beta_i])_{i=1\ldots8}$ is a base of $\mathrm{GF}(2)^8$, there exists a unique one-to-one linear change of base L mapping $[e_i]$ onto $[\beta_i]$ for all $i = 1, \ldots, 8$. Also define the isomorphism $\psi \overset{\text{def}}{=} L^{-1} \circ \varphi$ between $(\mathcal{S}, \circ)$ and $\big(\mathrm{GF}(2)^8, \oplus\big)$. One checks that ψ can be efficiently recovered, by using the unique decomposition given by Eq. 2. Indeed, for any $f \in \mathcal{S}$ the unique tuple of binary values $(\varepsilon_1, \ldots, \varepsilon_8)$ verifying Eq. 2 is easily computed—an exhaustive search would be quick enough, but we give a better algorithm at the end of this paragraph. By successively applying φ

$$\begin{tikzcd}
\cdot \arrow[r, "Q^{-1}"] \arrow[d, "f"'] & \cdot \arrow[r, "L^{-1}"] \arrow[d, "\oplus_{\varphi(f)}"'] & \cdot \arrow[r, "\oplus_{(Q\circ L)^{-1}([\text{'00'}])}"] \arrow[d, "\oplus_{\psi(f)}"'] & \cdot \arrow[d, "\oplus_{\psi(f)}"'] \\
\cdot & \cdot \arrow[l, "Q"'] & \cdot \arrow[l, "L"'] & \cdot \arrow[l, "\oplus_{(Q\circ L)^{-1}([\text{'00'}])}"'] \arrow[ll, bend left, "A"] \arrow[lll, bend left=40, "\widetilde{Q}"]
\end{tikzcd}$$

Fig. 6. Relating f, $\psi(f)$, and $\tilde{Q}$

and L^{-1} to f, one obtains

$$\psi(f) = L^{-1}\big(\varphi(f)\big) = L^{-1}\left(\bigoplus_{i=1\ldots8} \varepsilon_i[\beta_i]\right) = \bigoplus_{i=1\ldots8} \varepsilon_i[e_i] \ .$$

Thus the isomorphism ψ is entirely determined.

Let us explain how to recover Q from the knowledge of ψ, up to an unknown affine transformation A. For that purpose, consider the commutative diagram of Fig. 6, and define the GF(2)-affine one-to-one mapping A by

$$A(x) \overset{\text{def}}{=} L\big(x \oplus (Q \circ L)^{-1}([\text{'00'}])\big) = L(x) \oplus Q^{-1}([\text{'00'}]) \ ,$$

and let us set

$$\widetilde{Q} \overset{\text{def}}{=} Q \circ A \ .$$

One verifies that $\widetilde{Q}^{-1}(\text{'00'}) = [\text{'00'}]$. By applying the above definition of $\widetilde{Q}$, or equivalently by inspecting the commutative diagram of Fig.6, one checks that $f = \widetilde{Q} \circ \oplus_{\psi(f)} \circ \widetilde{Q}^{-1}$. Hence,

$$f(\text{'00'}) = \widetilde{Q}(\psi(f)) \ .$$

From our knowledge of ψ and f, we can therefore compute $\widetilde{Q} = Q \circ A$. $\square$

Now, we propose an efficient algorithm—time complexity is at most 2^{24}—that chooses a tuple $(f_1, \ldots, f_8)$ on the fly, and computes the corresponding mapping ψ. It was successfully implemented in C++.

```
INPUT :      $\mathcal{S}$
OUTPUT :     $\mathcal{R} \subset \mathcal{S} \times \mathrm{GF}(2^8)$ such that $\forall (f, \beta) \in \mathcal{R},\ \psi(f) = [\beta]$
ALGORITHM :  $\mathcal{R} \leftarrow \{(id, \text{‘00’})\}$
             $\psi(id) = [\text{‘00’}]$
             $e \leftarrow \text{‘01’}$
             while $\#\mathcal{R} < 2^8$ do
                $\mathcal{S} \leftarrow \mathcal{S} \setminus \{f\}$
                if $(f, \cdot) \notin \mathcal{R}$ then
                   $e \leftarrow \text{‘02’} \times e$
                   $\psi(f) = [e]$
                   foreach $(g, \eta) \in \mathcal{R}$ do
                      $\mathcal{R} \leftarrow \mathcal{R} \cup \{(f \circ g, [e] \oplus [\eta])\}$
                      $\psi(f \circ g) = [e] \oplus [\eta]$
                   enddo
                endif
             endwhile
```

Going back to our initial motivation, Theorem 1 enables us to recover for any round $r = 1, \ldots, 9$, the non-linear part $\widetilde{Q}^r_{i,j}$ of $Q^r_{i,j}$, i.e. such that $\widetilde{Q}^r_{i,j}{}^{-1} \circ Q^r_{i,j}$ is an affine mapping $A^r_{i,j}{}^{-1}$. Given the fact that for the next round, the input encoding $P^{r+1}_{i,j}$ must match the output encoding $Q^r_{i,j}$ of the previous round—that is $P^{r+1}_{i,j} \circ Q^r_{i,j}$ must be the identity—we have that $P^{r+1}_{i,j} \circ \widetilde{Q}^r_{i,j}$ is exactly the mapping $A^r_{i,j}$. Thus, we have reduced the original problem depicted in Fig. 5 where all P and Q are non-linear and matching, to one where they are affine and still matching. The next step is to recover those affine mappings, which is the subject of next sections.

3.2 Relations Between Affine Parasites

So let us start again with the setting depicted in Fig.5, except for the fact that all parasitic mappings $P^r_{i,j}$ and $Q^r_{i,j}$ are now affine. Since the problem is identical for each round, we drop the subscripts r and j without loss of generality. We have access to the following functions as lookup tables

$$\begin{cases} y_0(x_0,x_1,x_2,x_3) = Q_0\left(\text{‘02’} \cdot T'_0(x_0) \oplus \text{‘03’} \cdot T'_1(x_1) \oplus \text{‘01’} \cdot T'_2(x_2) \oplus \text{‘01’} \cdot T'_3(x_3)\right) \\ y_1(x_0,x_1,x_2,x_3) = Q_1\left(\text{‘01’} \cdot T'_0(x_0) \oplus \text{‘02’} \cdot T'_1(x_1) \oplus \text{‘03’} \cdot T'_2(x_2) \oplus \text{‘01’} \cdot T'_3(x_3)\right) \\ y_2(x_0,x_1,x_2,x_3) = Q_2\left(\text{‘01’} \cdot T'_0(x_0) \oplus \text{‘01’} \cdot T'_1(x_1) \oplus \text{‘02’} \cdot T'_2(x_2) \oplus \text{‘03’} \cdot T'_3(x_3)\right) \\ y_3(x_0,x_1,x_2,x_3) = Q_3\left(\text{‘03’} \cdot T'_0(x_0) \oplus \text{‘01’} \cdot T'_1(x_1) \oplus \text{‘01’} \cdot T'_2(x_2) \oplus \text{‘02’} \cdot T'_3(x_3)\right) \end{cases}$$

with the shortcut $T'_i = T_i \circ P_i$. Actually there is one more issue, which is that we do know the set $\{y_i\}_{i=0,\ldots,3}$ but we do not know the labels to put on each function. Put in another way, we know those functions have the general form

$$y_i(x_0,x_1,x_2,x_3) = Q\left(\alpha_{i,0} \cdot T'_0(x_0) \oplus \alpha_{i,1} \cdot T'_1(x_1) \oplus \alpha_{i,2} \cdot T'_2(x_2) \oplus \alpha_{i,3} \cdot T'_3(x_3)\right)$$

but we do not know what the underlying coefficients $\alpha_{i,j}$ occurring from the `MixColumn` step are. Let us hereafter denote by Λ_α the matrix over $\mathrm{GF}(2)^8$ of the multiplication by α.

Before going any further, let us state a very useful property. Though simple, it is a corner stone in the strategy we designed for the affine parasites' recovery, as well as in resolving the above mentioned renaming issue.

Proposition 1. *For any pair (y_i, y_j) as introduced above, there exists a unique linear mapping L and a unique constant c such that,*

$$\forall x_0 \in \mathrm{GF}(2^8), \quad y_i(x_0, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'}) = L\left(y_j(x_0, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'})\right) \oplus c \,. \tag{3}$$

Proof. Decompose the affine maps $Q_i(x) = A_i(x) \oplus q_i$ and $Q_j(x) = A_j(x) \oplus q_j$, where A_i and A_j are linear, q_i and q_j constants. Hence,

$$\begin{aligned} y_i(x, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'}) &= A_i(\alpha_{i,0} \cdot T_0'(x) \oplus c_i) \oplus q_i \,, \\ y_j(x, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'}, \text{`}\mathtt{00}\text{'}) &= A_j(\alpha_{j,0} \cdot T_0'(x) \oplus c_j) \oplus q_j \,. \end{aligned}$$

Thus, by taking $L = A_i \circ \Lambda_{\alpha_{i,0}/\alpha_{j,0}} \circ A_j^{-1}$ and $c = q_i \oplus A_i(c_i) \oplus L\left[q_j \oplus A_j(c_j)\right]$, Eq. 3 holds, which shows the existence of a solution.

The other way round, assuming there is a linear mapping L and a constant c such that Eq. 3 holds, amounts to saying that $(A_i \circ \Lambda_{\alpha_{i,0}} \oplus L \circ A_j \circ \Lambda_{\alpha_{j,0}}) \circ T_0'$ is a constant mapping. Since $T_0' = T_0 \circ P_0$ is one-to-one, and $(A_i \circ \Lambda_{\alpha_{i,0}} \oplus L \circ A_j \circ \Lambda_{\alpha_{j,0}})$ is a linear mapping, this constant must be '`00`'. Thus $L = A_i \circ \Lambda_{\alpha_{i,0}/\alpha_{j,0}} \circ A_j^{-1}$, which uniquely defines L. Then $\alpha_{i,0} \cdot y_i \oplus L \circ \alpha_{j,0} \cdot y_j$ is constant, and this constant uniquely defines c. □

Obviously, there are analogous statements where one varies the second, third, or fourth variable and keep the other one constant. Also note that given two functions y_i and y_j, there is a straightforward practical algorithm to get the corresponding affine mapping (L, c) connecting their affine parts together. Indeed, considering the 64 entries of the matrix L as well as the 8 entries of the constant vector of c as unknowns over $\mathrm{GF}(2)$, and using our knowledge of the functions y_i and y_j by values, one can form a highly overdefined linear system of $2^8 \times 8$ equations involving the 72 unknowns and solve it with time complexity much lower than 2^{16}.

3.3 Recovering the Affine Parasites

We note that Prop. 1 of the previous section enables us to directly compute the linear parts of Q_1, Q_2, and Q_3 from the knowledge of Q_0's linear part. We will therefore focus on Q_0's determination. This section is organized in two steps. First, we show how to recover the linear part of Q_0 up to Λ_γ, for some non-zero γ in $\mathrm{GF}(2^8)$. Then we show how this information can be used to recover both γ and the constant part q_0 of Q_0.

About Q_0's Linear Part. Let us recall that we decompose each affine transformation Q_i into its linear and constant parts: $Q_i(x) = A_i(x) + q_i$. Applying Prop. 1 with $i = 0$ and $j = 1$, we get $L_0 = A_0 \circ \Lambda_{\alpha_{0,0}/\alpha_{1,0}} \circ A_1^{-1}$. Then, using the variant of Prop. 1 with $i = 0$ and $j = 1$, but where one varies x_1 instead of x_0, we obtain $L_1 = A_0 \circ \Lambda_{\alpha_{0,1}/\alpha_{1,1}} \circ A_1^{-1}$. We are thus able to compute $L = L_0 \circ L_1^{-1}$, that is $L = A_0 \circ \Lambda_\beta \circ A_0^{-1}$ where $\beta = \alpha_{0,0}\alpha_{1,1}/\alpha_{0,1}\alpha_{1,0}$. Remembering that values α are standing for the `MixColumn` coefficients—i.e., taking their values in the set $\{$'`01`', '`02`', '`03`'$\}$—only 16 values for β remain possible, which are collected in the following set

$$B = \{\text{'02'}, \text{'d8'}, \text{'03'}, \text{'6f'}, \text{'04'}, \text{'bc'}, \text{'06'}, \text{'b7'}, \text{'05'}, \text{'25'}, \text{'4a'}, \text{'f8'}, \text{'7f'}, \text{'c8'}, \text{'64'}, \text{'5f'}\} .$$

(One checks that no element of B is contained in any subfield of $\mathrm{GF}(2^8)$.)

Thus, the new starting point is a matrix L, with the form $A_0 \circ \Lambda_\beta \circ A_0^{-1}$, and we want to retrieve both β and A_0. Given that β is chosen from B, computing the characteristic polynomial of L reduces the number of possibilities for β to at most 2; actually, either β is already determined, or $\beta \in \{b, b^2\} \subset B$. To ease the exposition, we assume that β is known, for instance by testing the two possibilities, and using Prop. 3 of the next section to determine the correct one.

Proposition 2. *Given an element β of $\mathrm{GF}(2^8)$ not in any subfields of $\mathrm{GF}(2^8)$ and its corresponding matrix $L = A_0 \circ \Lambda_\beta \circ A_0^{-1}$, we can compute with time complexity lower than 2^{16}, a matrix $\widetilde{A}_0$ such that there exists a unique non-zero constant γ in $\mathrm{GF}(2^8)$, so that $\widetilde{A}_0 = A_0 \circ \Lambda_\gamma$.*

Proof. We seek for $\widetilde{A}_0$ such that $L \circ \widetilde{A}_0 = \widetilde{A}_0 \circ \Lambda_\beta$. Considering $\widetilde{A}_0$'s entries as unknowns, this equation gives 64 equations in the 64 unknowns. Some non-trivial solution can be computed in time complexity $64^\omega < 2^{16}$, which we hereafter denote by $\widetilde{A}_0$. Then, define $A = A_0^{-1} \circ \widetilde{A}_0$. The equation $L \circ \widetilde{A}_0 = \widetilde{A}_0 \circ \Lambda_\beta$ also reads $\Lambda_\beta \circ A = A \circ \Lambda_\beta$. The only $\mathrm{GF}(2)$-affine mappings that commutes with the multiplication by β, are the multiplications by a $\mathrm{GF}(2^8)$ element. (To see this, write $A(x) = \sum_{i=0}^{7} \gamma_i \cdot x^{2^i}$. The commutativity constraint is then expressed by $\sum_{i=0}^{7} \gamma_i \beta^{2^i} \cdot x^{2^i} = \sum_{i=0}^{7} \beta\gamma_i \cdot x^{2^i}$ for all $x \in \mathrm{GF}(2^8)$. Since β is not contained in any subfield of $\mathrm{GF}(2^8)$, this in turn implies $\gamma_i =$ '`00`' for all i but $i = 0$. Therefore, as announced, $A(x) = \gamma_0\, x$.) Thus, there exists a unique $\gamma \in \mathrm{GF}(2^8)$ such that $A = \Lambda_\gamma$, and remembering that $A = A_0^{-1} \circ \widetilde{A}_0$, we have computed $\widetilde{A}_0 = A_0 \circ \Lambda_\gamma$. □

Now we only have to recover γ of Prop. 2 in order to fully determine A_0, the linear part of Q_0. In the following paragraph we explain how to compute it, as well as the constant part q_0 of Q_0, that is to recover Q_0 entirely.

Recovering P_i up to the Key, and Q_0. Let us return to the function we originally studied, namely

$$y_0(x_0, x_1, x_2, x_3) = Q_0\left(\bigoplus_{i=0}^{3} \alpha_{0,i} \cdot T_i \circ P_i(x_i)\right) . \qquad (4)$$

Remember that T_i stands for the key addition, followed by the AES-128's S-box application, that is $T_i(z) = S(z \oplus k_i)$. Hence, the mapping $x \mapsto S^{-1} \circ T_i \circ P_i(x) = P_i(x) \oplus k_i$ is affine. Now, from Prop. 2 we get some matrix $\widetilde{A}_0 = A_0 \circ \Lambda_{1/\gamma}$. We have the following:

Proposition 3. *There exists unique pairs* $(\delta_i, c_i)_{i=0,\ldots,3}$ *of elements in* $\mathrm{GF}(2^8)$*,* δ_i *being non-zero, such that*

$$\begin{aligned}
\widetilde{P}_0 &: x \longmapsto (S^{-1} \circ \Lambda_{\delta_0} \circ \widetilde{A}_0^{-1})\ \big(y_0(x, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}) \oplus c_0\big)\ ,\\
\widetilde{P}_1 &: x \longmapsto (S^{-1} \circ \Lambda_{\delta_1} \circ \widetilde{A}_0^{-1})\ \big(y_0(\text{‘}\mathtt{00}\text{’}, x, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}) \oplus c_1\big)\ ,\\
\widetilde{P}_2 &: x \longmapsto (S^{-1} \circ \Lambda_{\delta_2} \circ \widetilde{A}_0^{-1})\ \big(y_0(\text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, x, \text{‘}\mathtt{00}\text{’}) \oplus c_2\big)\ ,\\
\widetilde{P}_3 &: x \longmapsto (S^{-1} \circ \Lambda_{\delta_3} \circ \widetilde{A}_0^{-1})\ \big(y_0(\text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, x) \oplus c_3\big)\ ,
\end{aligned}$$

are affine mappings. Any pair (δ_i, c_i) *can be computed with time complexity* 2^{24}*. Moreover, those mappings are exactly* $\widetilde{P}_i = P_i(x) \oplus k_i$*.*

Proof. The proposition amounts to saying that $x \to S^{-1}(\delta \cdot S(x) \oplus c)$ is affine and non-constant. Since S represent the AES-128 S-box, and δ in non-zero, this is only possible if $(\delta, c) = (\text{‘}\mathtt{01}\text{’}, \text{‘}\mathtt{00}\text{’})$, hence the existence and uniqueness of (δ_i, c_i). (This is also very easy to verify by an exhaustive search, which we have done.)

Since c is ‘00’, we have $c_0 = y_0(x, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}) \oplus \alpha_{0,0} \cdot T_0(P_0(x))$, and since $\widetilde{A}_0 = A_0 \circ \Lambda_{1/\gamma}$, we get $\widetilde{P}_0(x) = S^{-1} \circ \Lambda_{\delta_0 \cdot \gamma \cdot \alpha_{0,0}} \circ S(P_0(x) \oplus k_0)$, where k_0 is a byte of the corresponding round key. As shown above, $\delta_0 \cdot \gamma \cdot \alpha_{0,0}$ must be ‘01’, hence $\widetilde{P}_0(x) = P_0(x) \oplus k_0$. The proof goes the same for $\widetilde{P}_1$, $\widetilde{P}_2$, and $\widetilde{P}_3$.

For every possible values for the pairs (δ_i, c_i)—there are 2^{16} possible pairs—we test if the corresponding mapping is affine. The lookup table has to be evaluated 2^8 times, and then 8 systems of 9 unknowns over $\mathrm{GF}(2)$, or equivalently one system of 72 unknowns which can be precomputed, has to be solved. Since the mapping evaluation through the lookup table dominates, the total time complexity is bounded by 2^{24}.

□

Since $\delta_i^{-1} = \gamma \cdot \alpha_{0,i}$, and given the fact that two of those $\alpha_{0,i}$ are ‘01’, another is ‘02’ and the last one is ‘03’, exactly two of the δ_i^{-1} are equal and share the common value γ. Therefore we know Λ_γ, and thus the matrix $A_0 = \widetilde{A}_0 \circ \Lambda_\gamma$, as well as the underlying `MixColumn` coefficients $\alpha_{0,i}$.

Also note that we recover at the same time the constant q_0 of the affine mapping Q_0. Indeed, let us define $c_4 = y_0(\text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’})$. Considering Eq. 4, it can also be written as

$$c_4 = \left(\bigoplus_{i=0}^{3} \alpha_{0,i} \cdot T_i \circ P_i(\text{‘}\mathtt{00}\text{’}) \right) \oplus q_0\ .$$

Then, remembering that

$$\begin{aligned} c_0 &= y_0(x, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}) \oplus \alpha_{0,0} \cdot T_0(P_0(x)) , \\ c_1 &= y_0(\text{‘}\mathtt{00}\text{’}, x, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}) \oplus \alpha_{0,1} \cdot T_1(P_1(x)) , \\ c_2 &= y_0(\text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, x, \text{‘}\mathtt{00}\text{’}) \oplus \alpha_{0,2} \cdot T_2(P_2(x)) , \\ c_3 &= y_0(\text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, \text{‘}\mathtt{00}\text{’}, x) \oplus \alpha_{0,3} \cdot T_3(P_3(x)) , \end{aligned}$$

which holds for every x and thus in particular for ‘00’, we easily check that the constant part of Q_0 is given by $q_0 = c_0 \oplus c_1 \oplus c_2 \oplus c_3 \oplus c_4$, which achieves to fully recover the mapping Q_0.

3.4 Putting Everything Together

Let us now summarize the whole process of recovering the white box AES-128 implementation's original parasites. In Sec. 3.1 we have shown how to compute, for any round $r = 1, \ldots, 9$ and any index $j = 0, \ldots, 3$, with time complexity 2^{24}, the non-linear part of any parasitic mapping $Q^r_{i,j}$, $i = 0, \ldots, 3$—and thus at the same time, the non-linear part of its inverse parasitic mapping $P^{r+1}_{i,j}$—up to some affine application $x \mapsto A^r_i(x) \oplus q^r_i$. Section 3.2 showed how to recover A_1, A_2, and A_3 from the knowledge of A_0, with time complexity lower than $3 \cdot 2^{16}$. Finally, Sec. 3.3 explained how to recover the affine mapping $x \mapsto A^r_0(x) \oplus q^r_0$, for $r = 2, \ldots, 9$, with time complexity lower than 2^{16}. At the same time, Sec. 3.3 also retrieved the missing affine part of $P^r_{i,j}$ up to the key addition, which will allow us, as explained in the next section, to extract the key embedded in the AES-128 white box implementation.

Hence the time complexity to compute the parasites for a complete obfuscated AES-128 round, is bounded by $4 \cdot 4 \cdot 2^{24} = 2^{28}$.

3.5 Key Extraction

We now give the procedure for the key extraction. The white box implementation of AES-128 key embeds round keys produced by the AES-128 key derivation algorithm. Thus the keys for two different rounds are related to each other. Using this property, one can obviously ease the recovery of the keys.

In a first step, we determine $Q^r_{i,j}$'s non-linear part for some round plus the entire parasites of two consecutive AES-128 obfuscated rounds. For instance, recover the parasitic mappings $Q^2_{i,j}$, as well as $\widetilde{P}^3_{i,j}$, $Q^3_{i,j}$, and $\widetilde{P}^4_{i,j}$, for $i = 0, \ldots, 3$ and $j = 0, \ldots, 3$ as described in Sec. 3.4. Then, since $P^{r+1}_{i,j} \circ Q^r_{i,j}$ must be the identity, we get the round key bytes as the composition of the affine mappings $\widetilde{P}$ and the affine part of Q which is denoted here by $\bar{Q}$, that is $k^3_{i,j} = \widetilde{P}^3_{i,j} \circ \bar{Q}^2_{i,j}$, and $k^4_{i,j} = \widetilde{P}^4_{i,j} \circ \bar{Q}^3_{i,j}$.

We now have the key bytes $k^3_{i,j}$ and $k^4_{i,j}$, however they are not necessarily in the right order. Still, the data flow exposed by the implementation, rules the way each round r key bytes relates to the next round $r + 1$ key bytes. If we assume—according to Sec. 3.1 of [1]—that the round keys were generated using the key derivation algorithm of AES-128, the added constraint between the 16 bytes $k^3_{i,j}$ and the 16 bytes $k^4_{i,j}$ allows us to rearrange them the right way. Thus, having correctly recovered an AES-128 round key, we are able to derive the whole set of round keys.

4 Conclusion

This paper explained how to extract, in a very efficient way, the whole secret key of a white box AES-128 implementation suggested in [1]. Some of our attack methods, for instance the technique of Sec. 3.1 used to recover the non linear parts of the encodings, are potentially applicable to other iterated blockciphers white box implementations using similar encoding and linear mixing techniques. However, parts of our attack take advantage from AES specificities. Therefore, no general conclusion can be drawn about the possibility to construct a strong white box AES implementation, or a strong white box implementations of other iterated blockciphers. Despite the general impossibility results concerning obfuscation [6], there is no evidence so far that strong white box implementation of blockciphers is unachievable; there is only some practical evidence that this is not an easy task. An interesting avenue for further research on obfuscation techniques might consist in developing a dedicated blockcipher, designed bottom-up with white box implementation in mind.

Acknowledgements

The authors thank the anonymous referees for their valuable comments.

References

1. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-Box Cryptography and an AES Implementation. In Nyberg, K., Heys, H.M., eds.: Selected Areas in Cryptography – SAC 2002. Volume 2595 of Lecture Notes in Computer Science., Springer Verlag (2003) 250–270
2. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: A White-Box DES Implementation for DRM Applications. In Feigenbaum, J., ed.: Digital Rights Management Workshop – DRM 2002. Volume 2696 of Lecture Notes in Computer Science., Springer Verlag (2003) 1–15
3. Jacob, M., Boneh, D., Felten, E.W.: Attacking an Obfuscated Cipher by Injecting Faults. In Feigenbaum, J., ed.: Digital Rights Management – DRM 2002. Volume 2696 of Lecture Notes in Computer Science., Springer Verlag (2003) 16–31
4. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer Verlag (2002)
5. National Institute of Standards and Technology: Advanced encryption standard. FIPS publication 197 (2001)
`http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

6. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (Im)possibility of Obfuscating Programs. In Kilian, J., ed.: Advances in Cryptology – CRYPTO 2001. Volume 2139 of Lecture Notes in Computer Science., Springer Verlag (2001) 1–18
7. Biryukov, A., Preneel, B., Braeken, A., de Cannire, C.: A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms. In Biham, E., ed.: Advances in Cryptology – EUROCRYPT 2003. Volume 1267 of Lecture Notes in Computer Science., Springer Verlag (2003) 33–50