Cryptanalysis of the Xiao – Lai White-Box AES Implementation

Yoni De Mulder[1], Peter Roelse[2], and Bart Preneel[1]

[1] KU Leuven
Dept. Elect. Eng.-ESAT/SCD-COSIC and IBBT,
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
{yoni.demulder,bart.preneel}@esat.kuleuven.be
[2] Irdeto B.V.
Taurus Avenue 105, 2132 LS Hoofddorp, The Netherlands
peter.roelse@irdeto.com

Abstract. In the white-box attack context, i.e., the setting where an implementation of a cryptographic algorithm is executed on an untrusted platform, the adversary has full access to the implementation and its execution environment. In 2002, Chow *et al.* presented a white-box AES implementation which aims at preventing key-extraction in the white-box attack context. However, in 2004, Billet *et al.* presented an efficient practical attack on Chow *et al.*'s white-box AES implementation. In response, in 2009, Xiao and Lai proposed a new white-box AES implementation which is claimed to be resistant against Billet *et al.*'s attack. This paper presents a practical cryptanalysis of the white-box AES implementation proposed by Xiao *et al.* The linear equivalence algorithm presented by Biryukov *et al.* is used as a building block. The cryptanalysis efficiently extracts the AES key from Xiao *et al.*'s white-box AES implementation with a work factor of about 2^{32}.

Keywords: white-box cryptography, AES, cryptanalysis, linear equivalence algorithm.

1 Introduction

A white-box environment is an environment in which an adversary has complete access to an implementation of a cryptographic algorithm and its execution environment. In a white-box environment, the adversary is much more powerful than in a traditional black-box environment in which the adversary has only access to the inputs and outputs of a cryptographic algorithm. For example, in a white-box environment the adversary can: (1) trace every program instruction of the implementation, (2) view the contents of memory and cache, including secret data, (3) stop execution at any point and run an off-line process, and/or (4) alter code or memory contents at will. To this end, the adversary can make use of widely available tools such as disassemblers and debuggers.

An example of a white-box environment is a digital content protection system in which the client is implemented in software and executed on a PC, tablet,

L.R. Knudsen and H. Wu (Eds.): SAC 2012, LNCS 7707, pp. 34–49, 2013.
© Springer-Verlag Berlin Heidelberg 2013

set-top box or a mobile phone. A malicious end-user may attempt to extract a secret key used for content decryption from the software. Next, the end-user may distribute this key to non-entitled end-users, or the end-user may use this key to decrypt the content directly, circumventing content usage rules.

White-box cryptography was introduced in 2002 by Chow, Eisen, Johnson and van Oorschot in [4,5], and aims at protecting a secret key in a white-box environment. In [4], Chow *et al.* present generic techniques that can be used to design implementations of a cryptographic algorithm that resist key extraction in a white-box environment. Next, the authors apply these techniques to define an example white-box implementation of the Advanced Encryption Standard (AES).

In 2004, a cryptanalysis of the white-box AES implementation by Chow *et al.* was presented by Billet, Gilbert and Ech-Chatbi [1]. This attack is referred to as the Billet Gilbert Ech-Chatbi (BGE) attack in the following. The BGE attack is efficient in that a modern PC only requires a few minutes to extract the AES key from the white-box AES implementation. In [7], James Muir presents a tutorial on the design and cryptanalysis of white-box AES implementations.

The BGE attack motivated the design of other white-box AES implementations offering more resistance against key extraction. In [3], Bringer, Chabanne and Dottax proposed a white-box AES implementation in which perturbations are added to AES in order to hide its algebraic structure. However, the implementation in [3] has been cryptanalyzed by De Mulder, Wyseur and Preneel in [8]. Recently, two new white-box AES implementations have been proposed: one in 2010 by Karroumi based on dual ciphers of AES [6] and one in 2009 by Xiao and Lai based on large linear encodings [10].

This paper presents a cryptanalysis of the Xiao – Lai white-box AES implementation proposed in [10], efficiently extracting the AES key from the white-box AES implementation. The cryptanalysis uses the linear equivalence algorithm presented by Biryukov, De Cannière, Braeken and Preneel in [2] as a building block. In addition to this, the structure of AES and the structure of the white-box implementation are exploited in the cryptanalysis. Key steps of the cryptanalysis have been implemented in `C++` and verified by computer experiments.

Organization of This Paper. The remainder of this paper is organized as follows. Section 2 briefly describes the white-box AES implementation proposed in [10] and the linear equivalence algorithm presented in [2]. Section 3 outlines the cryptanalysis of the white-box AES implementation. Finally, concluding remarks can be found in Sect. 4.

2 Preliminaries

2.1 AES-128

In this section, aspects of AES-128 that are relevant for this paper are described. For detailed information, refer to FIPS 197 [9]. AES-128 is an iterated block cipher mapping a 16 byte plaintext to a 16 byte ciphertext using a 128 bit key. AES-128 consists of 10 rounds and has 11 round keys which are derived

from the AES-128 key using the AES key scheduling algorithm. Each round of the algorithm updates a 16 byte state; the initial state of the algorithm is the plaintext and the final state of the algorithm is the ciphertext. In the following, a state is denoted by $[\text{state}_i]_{i=0,1,\ldots,15}$. A round comprises the following operations:

- *ShiftRows* is a permutation on the indices of the bytes of the state. It is defined by the permutation $(0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11)$, i.e. the first byte of the output of `ShiftRows` is the first byte of the input, the second byte of the output is the fifth byte of the input, and so on.
- *AddRoundKey* is a bitwise addition modulo two of a 128 bit round key k^r $(1 \leq r \leq 11)$ and the state.
- *SubBytes* applies the AES S-box operation to every byte of the state. AES uses one fixed S-box, denoted by S, which is a non-linear, bijective mapping from 8 bits to 8 bits
- *MixColumns* is a linear operation over $\text{GF}\left(2^8\right)$ operating on 4 bytes of the state at a time. The `MixColumns` operation can be represented by a 4×4 matrix `MC` over $\text{GF}\left(2^8\right)$. To update the state, 4 consecutive bytes of the state are interpreted as a vector over $\text{GF}\left(2^8\right)$ and multiplied by `MC`. Using the notation and the representation of the finite field as in [9], we have:

$$\begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \\ \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix} \leftarrow \texttt{MC} \cdot \begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \\ \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix} \quad \text{with} \quad \texttt{MC} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix},$$

  for $i = 0, 1, 2, 3$.

There are several equivalent ways to describe AES-128. The following description of AES-128 is the one used in this paper, where $\hat{k}^r$ for $1 \leq r \leq 10$ is the result of applying `ShiftRows` to k^r:

---

state $\leftarrow$ plaintext
for $r = 1$ **to** 9 **do**
  state $\leftarrow$ `ShiftRows` (state)
  state $\leftarrow$ `AddRoundKey`(state,$\hat{k}^r$)
  state $\leftarrow$ `SubBytes`(state)
  state $\leftarrow$ `MixColumns` (state)
end for
state $\leftarrow$ `ShiftRows` (state)
state $\leftarrow$ `AddRoundKey`(state,$\hat{k}^{10}$)
state $\leftarrow$ `SubBytes`(state)
state $\leftarrow$ `AddRoundKey`(state,k^{11})
ciphertext $\leftarrow$ state

2.2 The White-Box AES Implementation

This section describes the white-box AES implementation proposed in [10]. As the `MixColumns` operation is omitted in the final AES-128 round, the white-box implementation of the final round differs from the white-box implementation of the other rounds. However, as the final round is not relevant for the cryptanalysis presented in this paper, the description of its implementation is omitted below.

First, the `AddRoundKey` and `SubBytes` operations of AES round r $(1 \leq r \leq 9)$ are composed, resulting in 16 8-bit bijective lookup tables for each round. In the following, such a table is referred to as a T-box. If the 16 bytes of a 128 bit round key are denoted by $\hat{k}_i^r$ $(i = 0, 1, \ldots, 15)$, then the T-boxes are defined as follows:

$$T_i^r(x) = S(x \oplus \hat{k}_i^r) \quad \text{for } 1 \leq r \leq 9 \text{ and } 0 \leq i \leq 15 \ .$$

Second, the 4×4 matrix $\mathtt{MC}$ is split into two 4×2 submatrices: $\mathtt{MC}_0$ is defined as the first 2 columns of $\mathtt{MC}$ and $\mathtt{MC}_1$ is defined as the remaining 2 columns of $\mathtt{MC}$. Using this notation, the `MixColumns` matrix multiplication is given by:

$$\begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \\ \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix} \leftarrow \mathtt{MC}_0 \cdot \begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \end{pmatrix} \oplus \mathtt{MC}_1 \cdot \begin{pmatrix} \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix}$$

for $i = 0, 1, 2, 3$.

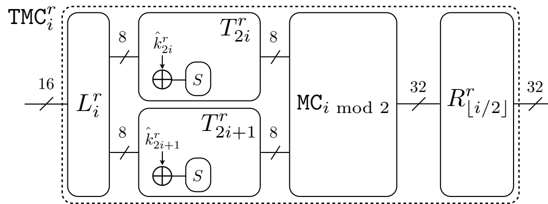


Fig. 1. Composition of T-boxes (T_{2i}^r and T_{2i+1}^r) and `MixColumns` operation ($\mathtt{MC}_{i \bmod 2}$) resulting in 16-to-32 bit lookup table $\mathtt{TMC}_i^r$

For $1 \leq r \leq 9$, the T-boxes and the `MixColumns` operations are composed as depicted in Fig. 1. Observe that this results in 8 lookup-tables per round, each table mapping 16 bits to 32 bits. To prevent an adversary from extracting the AES round keys from these tables, each table is composed with two secret white-box encodings L_i^r and $R_{\lfloor i/2 \rfloor}^r$ as depicted in Fig. 1. Each white-box encoding L_i^r is a bijective linear mapping from 16 bits to 16 bits, i.e., it can be represented by a non-singular 16×16 matrix over GF(2). Each white-box encoding $R_{\lfloor i/2 \rfloor}^r$ is a bijective linear mapping from 32 bits to 32 bits, i.e., it can be represented by a non-singular 32×32 matrix over GF(2). The resulting tables from 16 to 32 bits are referred to as $\mathtt{TMC}_i^r$ $(i = 0, 1, \ldots, 7)$ in the following.

Third, a 128×128 non-singular matrix M^r over GF (2) is associated with each round r ($1 \leq r \leq 9$). If SR denotes the 128×128 non-singular matrix over GF (2) representing the ShiftRows operation, then the matrix M^r is defined as follows:

$$M^r = \mathrm{diag}\Big((L_0^r)^{-1}, \ldots, (L_7^r)^{-1}\Big) \circ \mathtt{SR} \circ \mathrm{diag}\Big((R_0^{r-1})^{-1}, \ldots, (R_3^{r-1})^{-1}\Big) \ , \quad (1)$$

for $r = 2, 3, \ldots, 9$, where '$\circ$' denotes the function composition symbol. The matrix M^1 associated with the first round has a slightly different structure and is defined below.

Fourth, an additional secret white-box encoding is defined, denoted by IN. This encoding is represented by a non-singular 128×128 matrix over GF (2), and is applied to an AES-128 plaintext. Next, the non-singular 128×128 matrix M^1 over GF (2) is defined as follows:

$$M^1 = \mathrm{diag}\Big((L_0^1)^{-1}, \ldots, (L_7^1)^{-1}\Big) \circ \mathtt{SR} \circ \mathtt{IN}^{-1} \ . \quad (2)$$

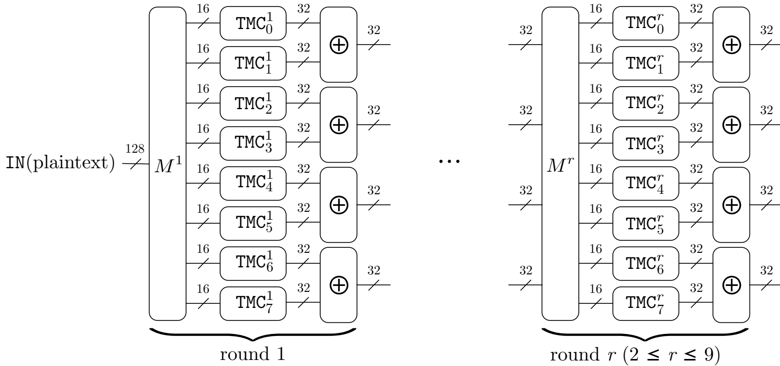


Fig. 2. White-box AES-128 implementation [10] of rounds $r = 1, 2, \ldots, 9$

Using these notations and definitions, the structure of the first 9 rounds of the white-box AES-128 implementation is depicted in Fig. 2. In the white-box implementation, an operation M^r is implemented as a matrix vector multiplication over GF (2) and an operation $\mathtt{TMC}_i^r$ is implemented as a look-up table. Notice that the output of two tables, which corresponds to the linearly encoded output of $\mathtt{MC}_0$ and $\mathtt{MC}_1$, is added modulo two in the white-box implementation. After the final AES round, a secret white-box encoding OUT is applied to the AES-128 ciphertext. OUT is represented by a non-singular 128×128 matrix over GF (2). Observe that, with the exception of the encodings IN and OUT, the white-box implementation of AES-128 is functionally equivalent to AES-128.

The main differences with the white-box AES-128 implementation presented in [4] are the following: (i) all secret white-box encodings are linear over GF (2),

and (ii) the secret white-box encodings operate on at least 2 bytes simultaneously instead of at least 4 bits (in case of a non-linear encoding) or at least a byte (in case of a linear encoding) in [4]. In [10], the authors argue that their white-box AES-128 implementation is resistant against the BGE attack [1].

2.3 The Linear Equivalence Algorithm

Definition 1. *Two permutations on n bits (or S-boxes) S_1 and S_2 are called linearly equivalent if a pair of linear mappings (A, B) from n to n bits exists such that $S_2 = B \circ S_1 \circ A$.*

A pair (A, B) as in this definition is referred to as a linear equivalence. Notice that both linear mappings A and B of a linear equivalence are bijective. If $S_1 = S_2$, then the linear equivalences are referred to as linear self-equivalences.

The linear equivalence problem is: given two n-bit bijective S-boxes S_1 and S_2, determine if S_1 and S_2 are linearly equivalent. An algorithm for solving the linear equivalence problem is presented in [2]. The inputs to the algorithm are S_1 and S_2, and the output is either a linear equivalence (A, B) in case S_1 and S_2 are linearly equivalent, or a message that such a linear equivalence does not exist. The algorithm is referred to as the linear equivalence algorithm (LE), and exploits the linearity of the mappings A and B. For an in depth description LE, refer to [2]. Below we give a brief description of a variant of LE where it is assumed that both given S-boxes map 0 to itself, i.e., $S_1(0) = S_2(0) = 0$. This variant of LE will be used as a building block for the cryptanalysis in this paper.

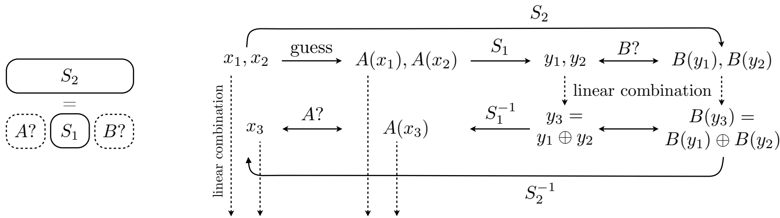


Fig. 3. Illustration how LE works

In case $S_1(0) = S_2(0) = 0$, at least two guesses for two points of A are necessary in order to start LE; select two distinct input points $x_1 \neq 0$ and $x_2 \neq 0$ and guess the values of $A(x_1)$ and $A(x_2)$. Based on these two initial guesses and the linearity of A and B, we incrementally build the linear mappings A and B as far as possible. The initial guesses $A(x_i)$ for the points x_i $(i = 1, 2)$ provide us with knowledge about B by computing $y_i = S_1(A(x_i))$ and $B(y_i) = S_2(x_i)$, which in turn gives us possibly new information about A by computing the images of the linear combinations of y_i and $B(y_i)$ through respectively S_1^{-1} and S_2^{-1}. This process is applied iteratively, where in each step of the process the linearity of the partially determined mappings A and B is verified by a Gaussian

elimination. Figure 3 illustrates the process. In case neither for A nor for B a set of n linearly independent inputs and outputs is obtained, the algorithm requires an additional guess for a new point x of A (or B) in order to continue.

If n linearly independent inputs and n linearly independent outputs to A are obtained, then a candidate for A can be computed. Similar reasoning applies to B. If the candidate linear equivalence is denoted by (A^*, B^*), then the correctness of this pair can be tested by verifying the relation $S_2 = B^* \circ S_1 \circ A^*$ for all possible inputs. If no candidate linear equivalence is found (due to linear inconsistencies occurred during the process), or if the candidate linear equivalence is incorrect, then the process is repeated with a different guess for $A(x_1)$ or for $A(x_2)$, or for any of the possibly additional guesses made during the execution of LE.

The original linear equivalence algorithm LE exits after finding one single linear equivalence which already proves that both given S-boxes S_1 and S_2 are linearly equivalent. However, by running LE over all possible guesses, i.e., both initial guesses as well as the possibly additional guesses made during the execution of LE, also other linear equivalences (A, B) can be found. The work factor of this variant is at least $n^3 \cdot 2^{2n}$, i.e., a Gaussian elimination (n^3) for each possible pair of initial guesses (2^{2n}).

3 Cryptanalysis of the White-Box AES Implementation

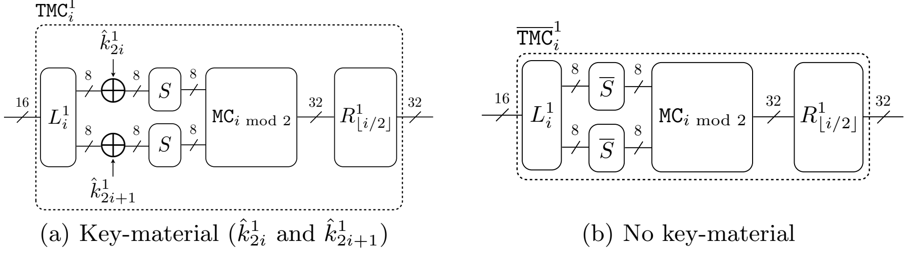
In this section, we elaborate on the cryptanalysis of the white-box AES-128 implementation proposed in [10] and described in Sect. 2.2. The goal of the cryptanalysis is the recovery of the full 128-bit AES key, together with the external input and output encodings, `IN` and `OUT` respectively.

The cryptanalysis focusses on extracting the first round key $\hat{k}^1$ contained within the 8 key-dependent 16-to-32 bit lookup tables $\mathtt{TMC}_i^1$ ($i = 0, \ldots, 7$) of the first round. Each table $\mathtt{TMC}_i^1$, depicted in Fig. 4(a), is defined as follows:

$$\mathtt{TMC}_i^1 = R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2} \circ S \| S \circ \oplus_{(\hat{k}^1_{2i} \| \hat{k}^1_{2i+1})} \circ L_i^1 \ , \qquad (3)$$

where $\|$ denotes the concatenation symbol, $\oplus_c$ denotes the function $\oplus_c(x) = x \oplus c$, and $S\|S$ denotes the 16-bit bijective S-box comprising two AES S-boxes in parallel. Given (3), the adversary knows that both S-boxes $S_1 = S\|S$ and $S_2 = \mathtt{TMC}_i^1$ are affine equivalent by the affine equivalence $(A, B) = (\oplus_{(\hat{k}^1_{2i} \| \hat{k}^1_{2i+1})} \circ L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2})$ such that $S_2 = B \circ S_1 \circ A$. As one can notice, only A is affine where the constant part equals the key-material contained within $\mathtt{TMC}_i^1$. Hence by making $\mathtt{TMC}_i^1$ key-independent (see Lemma 1 below), we can reduce the problem to finding linear instead of affine equivalences, for which we apply the linear equivalence algorithm (LE).

Lemma 1. *Given the key-dependent 16-to-32 bit lookup table* $\mathtt{TMC}_i^1$ *(defined by* (3) *and depicted in Fig. 4(a)), let* x_0^i *be the 16-bit value such that* $\mathtt{TMC}_i^1(x_0^i) = 0$*, and let* $\overline{\mathtt{TMC}}_i^1$ *be defined as* $\overline{\mathtt{TMC}}_i^1 = \mathtt{TMC}_i^1 \circ \oplus_{x_0^i}$*. If* $\overline{S}$ *is defined as the 8-bit bijective S-box* $\overline{S} = S \circ \oplus_{\text{'52'}}$ *where* S *denotes the AES S-box, then:*



(a) Key-material ($\hat{k}^1_{2i}$ and $\hat{k}^1_{2i+1}$) (b) No key-material

Fig. 4. Key-dependent table $\texttt{TMC}^1_i$ versus key-independent table $\overline{\texttt{TMC}}^1_i$

$$\overline{TMC}_i^{\ 1} = R^1_{\lfloor i/2 \rfloor} \circ MC_{i\ mod\ 2} \circ \overline{S} \| \overline{S} \circ L^1_i \ , \tag{4}$$

where $\overline{S}\|\overline{S}$ denotes the 16-bit bijective S-box comprising two S-boxes $\overline{S}$ in parallel. The key-independent 16-to-32 bit lookup table $\overline{TMC}_i^{\ 1}$ is depicted in Fig. 4(b).

Proof. Given the fact that $\texttt{TMC}^1_i$ is encoded merely by linear input and output encodings L^1_i and $R^1_{\lfloor i/2 \rfloor}$ (see (3)) and that $S(\text{‘}\texttt{52}\text{’}) = 0$, the 16-bit value x^i_0, for which $\texttt{TMC}_i{}^1(x^i_0) = 0$, has the following form:

$$x^i_0 = (L^1_i)^{-1}\Big((\hat{k}^1_{2i} \oplus \text{‘}\texttt{52}\text{’}) \parallel (\hat{k}^1_{2i+1} \oplus \text{‘}\texttt{52}\text{’})\Big) \ , \tag{5}$$

In the rare case that $x^i_0 = 0$, it immediately follows that both first round key bytes $\hat{k}^1_{2i}$ and $\hat{k}^1_{2i+1}$ are equal to ‘$\texttt{52}$’. Now, based on x^i_0, one can construct the key-independent 16-to-32 bit lookup table $\overline{\texttt{TMC}}^1_i$ as follows:

$$\begin{aligned}\overline{\texttt{TMC}}^1_i = \texttt{TMC}^1_i \circ \oplus_{x^i_0} &= R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2} \circ S\|S \circ \oplus_{(\hat{k}^1_{2i} \| \hat{k}^1_{2i+1})} \circ \oplus_{L^1_i(x^i_0)} \circ L^1_i \\ &= R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2} \circ S\|S \circ \oplus_{(\text{‘}\texttt{52}\text{’}\|\text{‘}\texttt{52}\text{’})} \circ L^1_i \\ &= R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2} \circ \overline{S}\|\overline{S} \circ L^1_i \ .\end{aligned}$$ □

The 8-bit bijective S-box $\overline{S}$ maps 0 to itself, i.e. $\overline{S}(\text{‘}\texttt{00}\text{’}) = \text{‘}\texttt{00}\text{’}$, since $S(\text{‘}\texttt{52}\text{’}) = \text{‘}\texttt{00}\text{’}$. Given (4), it also follows that $\overline{\texttt{TMC}}^1_i(0) = 0$.

Linear Equivalence Algorithm (LE). Given (4), the adversary knows that the 16-bit bijective S-box $S_1 = \overline{S}\|\overline{S}$ and the key-independent 16-to-32 bit lookup table $S_2 = \overline{\texttt{TMC}}^1_i$ (which is a bijective mapping from GF (2^{16}) to a 16-dimensional subspace of GF (2^{32})) obtained through Lemma 1, are linearly equivalent by the linear equivalence $(A, B) = (L^1_i, R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2})$. His goal is to recover this linear equivalence which contains the secret linear input encoding L^1_i he needs in order to extract both first round key bytes $\hat{k}^1_{2i}$ and $\hat{k}^1_{2i+1}$ out of the 16-bit value x^i_0 given by (5). The described problem is exactly what the linear equivalence algorithm (LE) tries to solve.

Since in this case both S-boxes $S_1 = \overline{S}\|\overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$ map 0 to itself, at least two initial 16-bit guesses $A(x_n)$ for two distinct points $x_n \neq 0$ $(n = 1, 2)$ of A are necessary to execute LE, and hence the work factor becomes at least 2^{44}, i.e., $n^3 \cdot 2^{2n}$ for $n = 16$. Furthermore, 128 linear equivalences $(A, B) = (A^s \circ L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2} \circ B^s)$ can be found, where (A^s, B^s) denotes the 128 linear self-equivalences of $\overline{S}\|\overline{S}$ (see Appendix A):

$$\overline{S}\|\overline{S} = B^s \circ \overline{S}\|\overline{S} \circ A^s \quad \rightarrow$$
$$\overline{\mathtt{TMC}}_i^1 = R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2} \circ B^s \circ \overline{S}\|\overline{S} \circ A^s \circ L_i^1 = B \circ \overline{S}\|\overline{S} \circ A \ .$$

The desired linear equivalence, i.e., the linear equivalence that the adversary wants to obtain, is denoted by $(A, B)_d = (L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2})$ and corresponds to the one with the linear self-equivalence $(A^s, B^s) = (I_{16}, I_{16})$, where I_{16} denotes the 16-bit identity matrix over GF (2).

Our Goal. In the following sections, we present a way how to modify the linear equivalence algorithm when applied to $S_1 = \overline{S}\|\overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$, such that only the single desired linear equivalence $(A, B)_d = (L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2})$ is given as output. At the same time, the work factor decreases as well. This modification exploits both the structure of AES as well as the structure of the white-box implementation.

3.1 Obtain Leaked Information about the Linear Input Encoding L_i^1

Due to the inherent structure of the white-box implementation, partial information about the linear input encoding L_i^1 of the key-independent tables $\overline{\mathtt{TMC}}_i^1$ of the first round is leaked. For each L_i^1, this leaked information comprises four sets of 16-bit encoded values for which the underlying unencoded bytes share a known bijective function. In the next section, we show how to modify the linear equivalence algorithm based on this leaked information. Here, we elaborate on how this information is extracted for $L_{i^*}^1$ of a given table $\overline{\mathtt{TMC}}_{i^*}^1$ for some $i^* \in \{0, 1, \ldots, 7\}$. Below, the following description is used: given an AES state by $[\text{state}_n]_{n=0,1,\ldots,15}$, then each set of 4 consecutive bytes $[\text{state}_{4j}, \text{state}_{4j+1}, \text{state}_{4j+2}, \text{state}_{4j+3}]$ for $j = 0, \ldots, 3$ is referred to as column j.

First, one builds an implementation which only consists of the single key-independent table $\overline{\mathtt{TMC}}_{i^*}^1$ followed by the matrix multiplication over GF (2) with M^2 given by (1) for $r = 2$. This implementation is in detail depicted in Fig. 5 for $i^* = 4$, where the internal states U, V and Y are indicated as well: the 2-byte state $U = (u_0\|u_1)$ corresponds to the 2-byte input to $\overline{S}\|\overline{S}$ and the 4-byte state $V = (v_0\|v_1\|v_2\|v_3)$ corresponds to the 4-byte output of $\mathtt{MC}_z$ where $z = i^* \bmod 2$. Since each byte v_l $(l = 0, \ldots, 3)$ of V is an output byte of $\mathtt{MC}_z$, the relation between U and V is given by $mc^z_{l,0} \otimes \overline{S}(u_0) \oplus mc^z_{l,1} \otimes \overline{S}(u_1) = v_l$ for $l = 0, \ldots, 3$, where $\otimes$ denotes the multiplication over the Rijndael finite field GF $\left(2^8\right)$ and

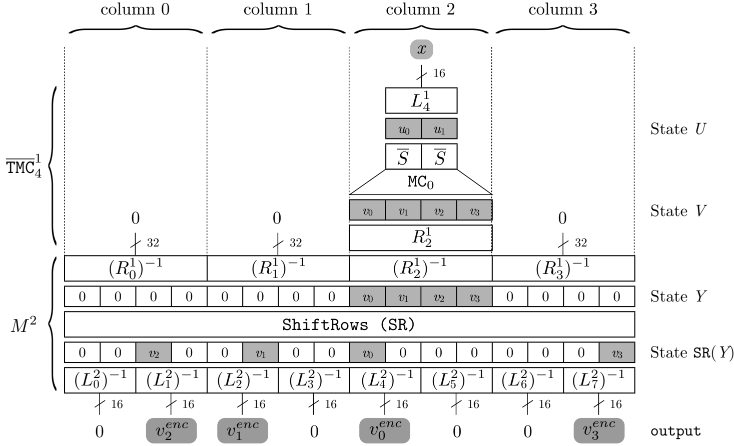


Fig. 5. Example of the implementation associated to $\overline{\mathtt{TMC}}_{i^*}^1$ with $i^* = 4$: identifying the four values v_l^{enc} for $l = 0, \ldots, 3$ in order to build the corresponding sets $\mathcal{S}_l^{i^*}$

the pair $(mc_{l,0}^z, mc_{l,1}^z)$ corresponds to the `MixColumns` coefficients on row l of the 4×2 submatrix $\mathtt{MC}_z$ over GF (2^8), i.e. $(mc_{l,0}^z, mc_{l,1}^z) \in \mathcal{S}_{\mathtt{MC}}$ with:

$$\mathcal{S}_{\mathtt{MC}} = \{(\text{`}\mathtt{02}\text{'}, \text{`}\mathtt{03}\text{'}), (\text{`}\mathtt{01}\text{'}, \text{`}\mathtt{02}\text{'}), (\text{`}\mathtt{01}\text{'}, \text{`}\mathtt{01}\text{'}), (\text{`}\mathtt{03}\text{'}, \text{`}\mathtt{01}\text{'})\} \ . \tag{6}$$

Then, the 16-byte input to M^2 is provided as follows: three 4-byte 0-values for columns $j \neq \lfloor i^*/2 \rfloor$ (in our example: $j = 0, 1, 3$) and the 4-byte output of $\overline{\mathtt{TMC}}_{i^*}^1$ for column $j = \lfloor i^*/2 \rfloor$ (in our example: $j = 2$). This ensures that the three columns j with $j \neq \lfloor i^*/2 \rfloor$ of the state Y remain zero, whereas column $j = \lfloor i^*/2 \rfloor$ equals the 4-byte state V. The `ShiftRows` operation ensures that the four bytes v_l $(l = 0, \ldots, 3)$ of V are spread over all four columns of the internal state $\mathtt{SR}(Y)$, which are then each encoded by a different linear encoding $(L^2_{2((\lfloor i^*/2 \rfloor - l) \bmod 4) + \lfloor l/2 \rfloor})^{-1}$. Hence the `output` state contains four 16-bit 0-values, whereas the other four 16-bit output values v_l^{enc} $(l = 0, \ldots, 3)$ each correspond to one of the 4 bytes v_l of V in a linearly encoded form.

If $v_l^{enc} = 0$, then the associated byte $v_l = \text{`}\mathtt{00}\text{'}$ as well, such that we have a known bijective function $f_l^{i^*}$ between the bytes u_0, u_1 of U, which is defined as:

$$u_1 = f_l^{i^*}(u_0) \text{ with } f_l^{i^*} = \left(\overline{S}\right)^{-1} \circ \otimes_{(mc_{l,1}^z)^{-1}} \circ \otimes_{mc_{l,0}^z} \circ \overline{S} \ , \tag{7}$$

where $z = i^* \bmod 2$. This function follows out of the equation $mc_{l,0}^z \otimes \overline{S}(u_0) \oplus mc_{l,1}^z \otimes \overline{S}(u_1) = \text{`}\mathtt{00}\text{'}$ and is depicted in Fig. 6.

Now, for the linear input encoding $L_{i^*}^1$ of $\overline{\mathtt{TMC}}_{i^*}^1$, four sets $\mathcal{S}_l^{i^*}$ $(l = 0, \ldots, 3)$ are built as follows. First associate one of the four values v_l^{enc} with each set $\mathcal{S}_l^{i^*}$. Then, for each set $\mathcal{S}_l^{i^*}$, store the 16-bit value x, given as input to $\overline{\mathtt{TMC}}_{i^*}^1$, for which the associated output value $v_l^{enc} = 0$. Do this for all $x \in \text{GF}\left(2^{16}\right)$. This results

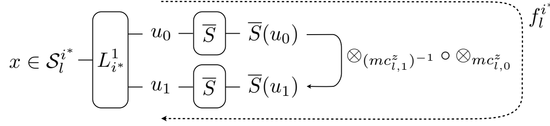


Fig. 6. How the known bijective function $f_l^{i^*}$ between u_0 and u_1 is defined

in that each set $\mathcal{S}_l^{i^*}$ is composed of 2^8 16-bit encoded values x for which the underlying unencoded bytes u_0, u_1 share the known bijective function $f_l^{i^*}$ given by (7) and depicted in Fig. 6:

$$\mathcal{S}_l^{i^*} = \{x \in \mathrm{GF}\left(2^{16}\right) \mid L_{i^*}^1(x) = u_0 \| u_1 \wedge u_1 = f_l^{i^*}(u_0)\} \quad \text{with } |\mathcal{S}_l^{i^*}| = 2^8 \ . \quad (8)$$

So with each set $\mathcal{S}_l^{i^*}$ $(l = 0, \ldots, 3)$, a known bijective function $f_l^{i^*}$ is associated.

3.2 Finding the Desired Linear Equivalence $(\boldsymbol{A}, \boldsymbol{B})_{\boldsymbol{d}}$: Obtain the Full Linear Input Encoding $\boldsymbol{L_i^1}$

So far, for the secret linear input encoding L_i^1 of each table $\overline{\mathtt{TMC}}_i^1$ $(i = 0, 1, \ldots, 7)$ of the first round, four sets $\mathcal{S}_l^i$ $(l = 0, \ldots, 3)$ defined by (8) are obtained. For each element $x \in \mathcal{S}_l^i$, the underlying unencoded bytes u_0, u_1 share a specific known bijective function f_l^i given by (7) and depicted in Fig. 6. Now, by exploiting this leaked information about L_i^1, we present an efficient algorithm for computing the desired linear equivalence $(A, B)_d = (L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \mathtt{MC}_{i \bmod 2})$. This enables the adversary to obtain the secret linear input encoding $A = L_i^1$ of $\overline{\mathtt{TMC}}_i^1$, which also corresponds to the linear input encoding of $\mathtt{TMC}_i^1$.

Algorithm for Finding $(\boldsymbol{A}, \boldsymbol{B})_{\boldsymbol{d}}$. Since $A = L_i^1$ in the desired linear equivalence, we exploit the leaked information about L_i^1 in order to make the two initial guesses $A(x_n)$ for two distinct points $x_n \neq 0$ $(n = 1, 2)$ of A. Only two out of four sets $\mathcal{S}_l^i$ are considered, i.e., those where the pair of `MixColumns` coefficients $(mc_{l,0}^z, mc_{l,1}^z)$ of the associated function f_l^i equals (‘`01`’, ‘`02`’) or (‘`02`’, ‘`03`’). We choose one of both sets and simply denote it by $\mathcal{S}^i$.

Now, select two distinct points $x_n \neq 0$ $(n = 1, 2)$ out of the chosen set, i.e., $x_n \in \mathcal{S}^i$. Based on definition (8) of $\mathcal{S}^i$, these points are defined as $x_n = (L_i^1)^{-1}\big(u_n \| f^i(u_n)\big)$ for some unknown distinct 8-bit values $u_n \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$, where f^i denotes the known function associated with $\mathcal{S}^i$. Now, based on this knowledge and the fact that we want to find $A = L_i^1$, the two initial guesses $A(x_n)$ are made as follows: $A(x_n) = a_n \| f^i(a_n)$ for all $a_n \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$ $(n = 1, 2)$. Hence although $A(x_n)$ is a 16-bit value, we only need to guess the 8-bit value a_n such that the total number of guesses becomes 2^{16} (i.e. $2^{2\frac{n}{2}}$ with $n = 16$). For each possible pair of initial guesses $\big(A(x_n) = a_n \| f^i(a_n)\big)_{n=1,2}$, LE is executed on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$. All found linear equivalences are stored in the set $\mathcal{S}_{\mathrm{LE}}$.

It is assumed that at least $(A,B)_d = (L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \mathtt{MC}_{i \bmod 2}) \in \mathcal{S}_{\mathrm{LE}}$, which occurs when $a_n = u_n$ for $n = 1,2$. It is possible that one or more linear equivalences $(A,B) = (A^s \circ L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \mathtt{MC}_{i \bmod 2} \circ B^s)$ with $A^s \neq I_{16}$ (see the introducing part of Sect. 3) can be found as well such that $|\mathcal{S}_{\mathrm{LE}}| > 1$. In that case, the procedure needs to be repeated for two new distinct points $x_n^* \neq 0$ $(n = 1,2)$ out of the chosen set $\mathcal{S}^i$, which are also distinct from the original chosen points $x_n \neq 0$ $(n = 1,2)$. This results in a second set $\mathcal{S}_{\mathrm{LE}}^*$. Assuming that all possible linear equivalences between $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$ are given by $(A,B) = (A^s \circ L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \mathtt{MC}_{i \bmod 2} \circ B^s)$, it can be shown that for both considered sets, it is impossible that a linear equivalence with $A^s \neq I_{16}$ is given as output during both executions of the procedure. Hence taking the intersection of both sets $\mathcal{S}_{\mathrm{LE}}$ and $\mathcal{S}_{\mathrm{LE}}^*$ results in the desired linear equivalence $(A,B)_d$.

Algorithm 1 gives a detailed description of the whole procedure. It has an average case work factor of 2^{29}, i.e., $2 \cdot n^3 \cdot 2^{2\frac{n}{2}}$ for $n = 16$.

Algorithm 1. Finding the desired linear equivalence $(A,B)_d$

Input: $S_1 = \overline{S} \| \overline{S}$, $S_2 = \overline{\mathtt{TMC}}_i^1$, $\mathcal{S}^i$, f^i
Output: $(A,B)_d = (L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \mathtt{MC}_{i \bmod 2})$

1: select two distinct points $x_1, x_2 \in \mathcal{S}^i$ with $x_n \neq 0$ $(n = 1,2)$
2: **call** search-LE$(x_1, x_2) \rightarrow \mathcal{S}_{\mathrm{LE}}$
3: **if** $|\mathcal{S}_{\mathrm{LE}}| > 1$ **then**
4: select two distinct points $x_1^*, x_2^* \in \mathcal{S}^i$ with $x_n^* \neq 0$, $x_n^* \neq x_m$ $(n = 1,2\ ;\ m = 1,2)$
5: **call** search-LE$(x_1^*, x_2^*) \rightarrow \mathcal{S}_{\mathrm{LE}}^*$
6: $\mathcal{S}_{\mathrm{LE}} \leftarrow \mathcal{S}_{\mathrm{LE}} \cap \mathcal{S}_{\mathrm{LE}}^*$
7: **end if**
8: **return** $\mathcal{S}_{\mathrm{LE}}$

where
Procedure search-LE (**Input:** x_1, x_2 – **Output:** $\mathcal{S}_{\mathrm{LE}}$)

1: $\mathcal{S}_{\mathrm{LE}} \leftarrow \varnothing$
2: **for all** $a_1 \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$ **do**
3: $A(x_1) \leftarrow a_1 \| f^i(a_1)$
4: **for all** $a_2 \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$ **do**
5: $A(x_2) \leftarrow a_2 \| f^i(a_2)$
6: **call** LE on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$ with initial guesses $A(x_1), A(x_2) \rightarrow \mathcal{S}_{\mathrm{LE}}$
7: **end for**
8: **end for**

Choice of Set $\mathcal{S}^i$. To each of the four sets $\mathcal{S}_l^i$ $(l = 0, \ldots, 3)$, a pair of `MixColumns` coefficients $(mc_{l,0}^z, mc_{l,1}^z) \in \mathcal{S}_{\mathtt{MC}}$ (see (6)) of the associated function f_l^i is related. Let us denote this relation by $\mathcal{S}_l^i \leftrightarrow (mc_{l,0}^z, mc_{l,1}^z)$. Here, we elaborate on the fact that not all four sets are equally suitable to be used in Algorithm 1.

$\mathcal{S}_l^i \leftrightarrow$ (‘01’,‘01’): in this case, the associated function f_l^i becomes the identity function such that the pair of initial guesses becomes $\left(A(x_n) = a_n \| a_n\right)_{n=1,2}$ with $a_n \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$. When executing LE on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}_i^1}$ for any such pair, we only find at most 8 linearly independent inputs and output to A (or B). This is explained by the fact that linear combinations of $a_n \| a_n$ (or of $\overline{S}(a_n) \| \overline{S}(a_n)$) span at most an 8-dimensional space. In order to continue executing LE, an additional guess for a new point x of A (or B) is required which increases the work factor. Hence we avoid using this set;

$\mathcal{S}_l^i \leftrightarrow$ {(‘01’,‘02’),(‘02’,‘03’),(‘03’,‘01’)}: computer simulations show that all three remaining sets can be used in Algorithm 1 without requiring an additional guess during the execution of LE. However, in the worst case scenario, using the set $\mathcal{S}_l^i \leftrightarrow$ (‘03’, ‘01’) requires that the procedure ‘`search-LE`’ needs to be executed 4 times in total in order to find the single desired linear equivalence $(A, B)_d$, instead of at most 2 times in case of the set $\mathcal{S}_l^i \leftrightarrow$ (‘01’, ‘02’) or the set $\mathcal{S}_l^i \leftrightarrow$ (‘02’, ‘03’). Note that Algorithm 1 assumes that one of the latter sets is chosen.

Implementation. Algorithm 1 has been implemented in `C++` and tests have been conducted on an Intel Core2 Quad @ 3.00GHZ. For the conducted tests, we chose the set $\mathcal{S}_l^i$ where $(mc_{l,0}^z, mc_{l,1}^z) =$ (‘02’, ‘03’). We ran the implementation 3000 times in total, each time for a different randomly chosen L_i^1 and $R_{\lfloor i/2 \rfloor}^1$. Only 4 times the procedure ‘`search-LE`’ needed to be repeated since 2 linear equivalences were found during the first execution. The implementation always succeeded in finding only the single desired linear equivalence $(A, B)_d$, which required on average ≈ 1min. It should be noted that the implementation was not optimized for speed, hence improvements are possible. The implementation also showed that each pair of initial guesses as defined above were sufficient in order to execute LE, i.e., no additional guesses were required.

3.3 Extracting the Full 128-Bit AES Key and the External Input and Output Encodings `IN` and `OUT`

At this point in the cryptanalysis, we extracted the 16-bit secret linear input encodings L_i^1 of all 8 16-to-32 bit tables $\mathtt{TMC}_i^1$ $(i = 0, \ldots, 7)$ of the first round.

Extracting the Full 128-bit AES Key. Given the 16-bit value x_0^i of each table $\mathtt{TMC}_i^1$ defined by $\mathtt{TMC}_i^1(x_0^i) = 0$ (see (5)), the adversary can extract both first round key bytes $\hat{k}_{2i}^1$ and $\hat{k}_{2i+1}^1$ contained within each key-dependent table $\mathtt{TMC}_i^1$ as follows:

$$\hat{k}_{2i}^1 \parallel \hat{k}_{2i+1}^1 = L_i^1(x_0^i) \oplus \left(\text{‘52’} \parallel \text{‘52’}\right) .$$

By doing so for each table $\mathtt{TMC}_i^1$ $(i = 0, 1, \ldots, 7)$ and taking into account the data flow of the white-box implementation of the first round, the adversary is able to obtain the full 128-bit first round key $\hat{k}^1$, which after applying the inverse `ShiftRows` operation to it results in the actual first round key k^1. According to the AES key scheduling algorithm, k^1 corresponds to the 128-bit AES key k.

Extracting the External Input and Output Encodings $\mathtt{IN}$ *and* $\mathtt{OUT}$. By knowing all 8 linear input encodings L_i^1 $(i = 0, 1, \ldots, 7)$ of the first round, the external 128-bit linear input encoding $\mathtt{IN}$ can be extracted out of the 128×128 binary matrix M^1 given by (2) as follows: $\mathtt{IN}^{-1} = \mathtt{SR}^{-1} \circ \mathrm{diag}(L_0^1, \ldots, L_7^1) \circ M^1$.

The external 128-bit linear output encoding $\mathtt{OUT}$ can be extracted once both the AES key k and $\mathtt{IN}$ have been recovered. Let us take the canonical base $([e_i])_{i=0,\ldots,127}$ of the vector space $\mathrm{GF}\,(2)^{128}$ and calculate for each 128-bit base vector e_i the 128-bit value $y_i = \mathrm{WBAES}_k\big(\mathtt{IN}\big(\mathrm{AES}_k^{-1}(e_i)\big)\big)$, where WBAES_k denotes the given white-box AES implementation defined by $\mathrm{WBAES}_k = \mathtt{OUT} \circ \mathrm{AES}_k \circ \mathtt{IN}^{-1}$ and AES_k^{-1} denotes the inverse standard AES implementation, both instantiated with the AES key k:

$$y_i = \underbrace{\mathtt{OUT}(\mathrm{AES}_k(\mathtt{IN}^{-1}}_{\mathrm{WBAES}_k}(\mathtt{IN}(\mathrm{AES}_k^{-1}(e_i))))) = \mathtt{OUT}(e_i) \ .$$

As one can notice, y_i corresponds to the image of e_i under the external 128-bit linear output encoding $\mathtt{OUT}$. Hence $\mathtt{OUT}$ is completely defined by calculating all pairs (e_i, y_i) for $i = 0, \ldots, 127$.

3.4 Work Factor

The overall work factor of our cryptanalysis is dominated by the execution of Algorithm 1 in order to obtain the linear input encodings L_i^1 of all 8 16-to-32 bit tables $\mathtt{TMC}_i^1$ $(i = 0, \ldots, 7)$ of the first round. The algorithm has a work factor of about 2^{29}. Thus, executing the algorithm on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$ for $i = 0, 1, \ldots, 7$ leads to an overall work factor of about $8 \cdot 2^{29} = 2^{32}$. Once obtained L_i^1 for $i = 0, 1, \ldots, 7$, the AES key together with the external encodings can be extracted as explained in Sect. 3.3.

4 Conclusion

This paper described in detail a practical attack on the white-box AES implementation of Xiao and Lai [10]. The cryptanalysis exploits both the structure of AES as well as the structure of the white-box AES implementation. It uses a modified variant of the linear equivalence algorithm presented by Biryukov *et al.* [2], which is built by exploiting leaked information out of the white-box implementation. The attack efficiently extracts the AES key from Xiao *et al.*'s white-box AES implementation with a work factor of about 2^{32}. In addition to extracting the AES key, which is the main goal in cryptanalysis of white-box implementations, our cryptanalysis is also able to recover the external input and output encodings. Crucial parts of the cryptanalysis have been implemented in `C++` and verified by computer experiments. The implementation furthermore shows that both the 128-bit AES key as well as the external input and output encodings can be extracted from the white-box implementation in just a few minutes on a modern PC.

Acknowledgements. This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the European Commission through the ICT Programme under contract ICT-2007-216676 ECRYPT II. In addition, this work was supported by the Flemish Government, FWO WET G.0213.11N and IWT GBO SEC SODA. Yoni De Mulder was supported in part by IBBT (Interdisciplinary institute for BroadBand Technology) of the Flemish Government.

References

1. Billet, O., Gilbert, H., Ech-Chatbi, C.: Cryptanalysis of a White Box AES Implementation. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 227–240. Springer, Heidelberg (2004)
2. Biryukov, A., De Cannière, C., Braeken, A., Preneel, B.: A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 33–50. Springer, Heidelberg (2003)
3. Bringer, J., Chabanne, H., Dottax, E.: White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468 (2006), `http://eprint.iacr.org/2006/468.pdf`
4. 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.) SAC 2002. LNCS, vol. 2595, pp. 250–270. Springer, Heidelberg (2003)
5. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.C.: A White-Box DES Implementation for DRM Applications. In: Feigenbaum, J. (ed.) DRM 2002. LNCS, vol. 2696, pp. 1–15. Springer, Heidelberg (2003)
6. Karroumi, M.: Protecting White-Box AES with Dual Ciphers. In: Rhee, K.-H., Nyang, D. (eds.) ICISC 2010. LNCS, vol. 6829, pp. 278–291. Springer, Heidelberg (2011)
7. Muir, J.A.: A tutorial on white-box AES. Mathematics in Industry (2012), `http://www.ccsl.carleton.ca/~jamuir/papers/wb-aes-tutorial.pdf`
8. De Mulder, Y., Wyseur, B., Preneel, B.: Cryptanalysis of a Perturbated White-Box AES Implementation. In: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 292–310. Springer, Heidelberg (2010)
9. National Institute of Standards and Technology. Advanced encryption standard. FIPS publication 197 (2001), `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`
10. Xiao, Y., Lai, X.: A secure implementation of white-box AES. In: 2nd International Conference on Computer Science and its Applications (CSA 2009), pp. 1–6. IEEE (2009)

A Linear Self-equivalences of 16-bit Bijective S-box $\overline{S}\|\overline{S}$

Let the AES S-box S be defined as $S = A(x^{-1})$ where A is a 8-bit bijective affine mapping over GF (2) and x^{-1} denotes the inverse of x in the Rijndael finite field GF (2^8) with '`00`' $^{-1}$ = '`00`', and let the 8-bit bijective S-box $\overline{S}$ be defined as

$\overline{S} = S \circ \oplus_{\text{‘52’}}$. If Φ_l denotes the set of exactly $\#_l = 8$ linear self-equivalences (α, β) of $\overline{S}$ such that $\overline{S} = \beta \circ \overline{S} \circ \alpha$, and is defined as:

$$\Phi_l = \left\{(\alpha = [c] \circ Q^i, \beta = A \circ Q^{-i} \circ [c] \circ A^{-1}) \mid (i, c) \in \mathcal{S}_l\right\} \quad \text{with}$$
$$\mathcal{S}_l = \{(0, \text{‘01’}), (1, \text{‘05’}), (2, \text{‘13’}), (3, \text{‘60’}), (4, \text{‘55’}), (5, \text{‘f6’}), (6, \text{‘b2’}), (7, \text{‘66’})\} \ ,$$

where $[c]$ denotes the 8×8 binary matrix representing a multiplication by c in $\mathrm{GF}\left(2^8\right)$ and Q denotes the 8×8 binary matrix that performs the squaring operation in $\mathrm{GF}\left(2^8\right)$ (both considered the Rijndael finite field), then the 16-bit bijective S-box comprising two identical S-boxes $\overline{S}$ in parallel, i.e. $\overline{S} \| \overline{S}$, has $2 \cdot \#_l^2 = 128$ trivial linear self-equivalences denoted by the pair of 16-bit bijective linear mappings (A^s, B^s) such that $\overline{S} \| \overline{S} = B^s \circ \overline{S} \| \overline{S} \circ A^s$, with the following diagonal structure:

$$A^s = \begin{pmatrix} \alpha_1 & 0_{8\times8} \\ 0_{8\times8} & \alpha_2 \end{pmatrix} \ , B^s = \begin{pmatrix} \beta_1 & 0_{8\times8} \\ 0_{8\times8} & \beta_2 \end{pmatrix} \text{ or}$$
$$A^s = \begin{pmatrix} 0_{8\times8} & \alpha_1 \\ \alpha_2 & 0_{8\times8} \end{pmatrix} \ , B^s = \begin{pmatrix} 0_{8\times8} & \beta_2 \\ \beta_1 & 0_{8\times8} \end{pmatrix} \ , \tag{9}$$

for any combination $[(\alpha_1, \beta_1), (\alpha_2, \beta_2)]$ where both $(\alpha_1, \beta_1), (\alpha_2, \beta_2) \in \Phi_l$. In (9), $0_{8\times8}$ denotes the 8×8 binary zero-matrix over $\mathrm{GF}(2)$.

The linear equivalence algorithm (implemented in `C++`) has been executed with $S_1 = S_2 = \overline{S} \| \overline{S}$ and found exactly these 128 linear self-equivalences (A^s, B^s) of the form (9).