A Tutorial on White-box AES

James A. Muir*

Irdeto Canada
http://www.irdeto.com

Abstract. White-box cryptography concerns the design and analysis of implementations of cryptographic algorithms engineered to execute on untrusted platforms. Such implementations are said to operate in a *white-box attack context*. This is an attack model where all details of the implementation are completely visible to an attacker: not only do they see input and output, they see every intermediate computation that happens along the way. The goal of a white-box attacker when targeting an implementation of a cipher is typically to extract the cryptographic key; thus, white-box implementations have been designed to thwart this goal (i.e., to make key extraction difficult/infeasible). The academic study of white-box cryptography was initiated in 2002 in the seminal work of Chow, Eisen, Johnson and van Oorschot (SAC 2002). Here, we review the first white-box AES implementation proposed by Chow et al. and give detailed information on how to construct it. We provide a number of diagrams that summarize the flow of data through the various look-up tables in the implementation, which helps clarify the overall design. We then briefly review the impressive 2004 cryptanalysis by Billet, Gilbert and Ech-Chatbi (SAC 2004). The BGE attack can be used to extract an AES key from Chow et al.'s original white-box AES implementation with a work factor of about 2^{30}, and this fact has motivated subsequent work on improved AES implementations.

Keywords: software, cryptography, AES, white-box.

1 Introduction

Suppose cryptographic software is deployed on a host that is not fully trusted. Examples of this include software distributed to end-users as part of some digital rights management (DRM) system, or client software running in the cloud, or even a cryptographic operation being executed on a smart-card [3,11,12]. Continuing with the DRM example, suppose that after installing the software on a PC, laptop, tablet or mobile phone, the end-user is then able to purchase some type of premium content (e.g., a television show, sports feed, video game or e-book). The content arrives at the user's device encrypted, and is decrypted by the software as it is viewed.

* Version: 22 February 2013 23:19:23 EST. This is an extended and corrected version of a paper that initially appeared in Advances in Network Analysis and its Applications, *Mathematics in Industry* **18** (2013), 209-229.

A malicious end-user may attempt to extract cryptographic keys from the software and then use them to redistribute content outside the DRM system. An attacker such as this is much more powerful than a traditional cryptographic attacker who sees only the inputs and outputs of a cryptographic operation (i.e., an attacker who treats the implementation as a black-box). This attacker is targeting *software* running on their own device. They are able to examine its inputs, outputs, and, with the help of a disassembler/debugger (e.g., IDA Pro, OllyDbg), the result of every intermediate computation it carries out. Essentially, this attacker has total visibility into the cryptographic operation.

The study of cryptographic implementations in this type of attack context was introduced in the academic literature in 2002 by Chow, Eisen, Johnson and van Oorschot [4]. In their seminal work, they motivated and defined the *white-box attack context* and presented some generic techniques that can be used to help create cryptographic implementations that resist key-extraction. They also applied those techniques to produce example implementations of AES [4] and (in another work) DES [5].

The terms *white-box AES* and *white-box DES* have become synonymous with the first implementations disclosed by Chow et al., but these terms are actually more general. Any AES implementation engineered to resist key extraction in the white-box attack context could be called white-box AES. And note that there are a number of ways that the techniques proposed by Chow et al. could be applied to AES and DES to create protected implementations.

With the 10 year anniversary of the papers by Chow et al. upcoming, it seems an appropriate time to give them another look. Here, we review their original AES implementation and give detailed information on how to construct it. A fair criticism of Chow et al.'s AES paper is that it is quite dense, and extracting the complete details of their protected AES implementation from it can be challenging. Our goal here is to make that information more accessible; this may be of particular benefit to new researchers and software engineers who are beginning to learn about white-box cryptography.

We also give a brief review of the 2004 algebraic cryptanalysis by Billet, Gilbert and Ech-Chatbi [2] that shows how a white-box attacker can extract the key from Chow et al.'s original AES implementation using 2^{30} work-steps in the worst case. This impressive cryptanalysis has motivated the design of new white-box AES implementations more resistant to key extraction and a number of subsequent works in the open literature have appeared on this topic (e.g., [13], [14], [19], [10]).

Outline. We begin by discussing the definitional results on program obfuscation by Barak et al. in §2. We then start our review of Chow et al.'s public AES implementation in §3 by describing a table-based implementation that does not include any protections against white-box attacks. In §4, we explain how encodings and mixing bijections are applied to the implementation with the goal of making it more resistant to key extraction attacks. Then we review the cryptanalysis by Billet et al. in §5, and end with some remarks in §6.

Preliminary Facts and Notation. Let x and y be bit-strings of equal length. We denote the bit-wise *exclusive-or* of x and y by $x \oplus y$. When we say that a transformation, L, from bit-strings to bit-strings is *linear*, we mean that the identity $L(x \oplus y) = L(x) \oplus L(y)$ holds for all inputs x, y. In particular, any transformation that permutes the bits of x is linear. If L is linear, then it can be represented using matrix-vector multiplication over GF(2); that is, there exists a matrix representation of L. The composition of two functions f and g is denoted by $f \circ g$, where $f \circ g(x) = f(g(x))$. If v is a column vector, then we use v^T to denote its transpose. We sometimes abuse functional notation and apply it to matrices; for example, if M and N are matrices that can be multiplied together, then $M \circ N$ denotes the transformation $v \mapsto MNv$. If c is a constant bit-string, then $\oplus_c$ denotes the function $x \mapsto x \oplus c$.

2 Barak et al.'s Impossibility Theorem

In 2001, Barak et al. [1] published foundational results on *program obfuscation*. They defined a *program obfuscator* as an algorithm that takes a program description as input (e.g., C code) and transforms it into a functionally equivalent obfuscated program description that satisfies the *virtual black-box property*; that is, any information that can be efficiently learned from the obfuscated program description can also be efficiently learned by studying only inputs and outputs of the original program. Their main result is that *generic* program obfuscators cannot exist – they show that there must always be some class of programs that when run through the obfuscator leak information that is not available through black-box interaction with the original programs.

The results of Barak et al. are sometimes incorrectly cited to refute the possibility of designing cryptographic implementations that resist white-box attacks.[1] However, there is no evidence that common block ciphers and their component operations belong to the special family of programs that cannot be securely obfuscated; see the statements to this effect by Billet et al. [2, page 239] and by Wyseur [16, page 91]. The successful white-box cryptanalysis of Chow et al.'s published AES [2] and DES [8,17] implementations do not point to any fundamental flaw that must be present in all white-box implementations of block ciphers; parts of those attacks exploit details particular to the AES and DES algorithms. It is possible that some block ciphers can be securely obfuscated (in a strict definitional sense); and it is also possible that, with the introduction of new techniques, strong white-box implementations of AES and DES could be created.

Barak et al. suggest that the virtual black-box property used in their definition of secure obfuscation may be too strong (i.e., perhaps obfuscated programs necessarily leak some non-black-box information, which may or may not be useful

[1] More generally, the results are also cited incorrectly in anti-DRM commentaries. Barak has published a non-technical summary of their results in an attempt to dispel some of the confusion (see http://www.cs.princeton.edu/~boaz/Papers/obf_informal.html).

to an attacker). Other definitions of secure obfuscation have been proposed, and using those definitions a number of positive results have been derived (cf. [9]).

3 Table-based Implementation

One completely impractical way to create a white-box implementation of a block cipher that does not leak any more information than a black-box implementation is to create a massive look-up table that maps, say, plaintext to ciphertext under some fixed key. If the block length of the cipher is ℓ bits, then the look-up table consists of 2^ℓ entries with each entry being an ℓ-bit string. Since ℓ is typically 64 or 128, the amount of memory required to store this table is beyond the capabilities of any real-world device. However, using a number of smaller look-up tables can lead to a practical solution.

We begin our description of Chow et al.'s white-box AES implementation by first presenting an implementation that does not offer any resistance to white-box attacks. This implementation makes extensive use of look-up tables, and the cipher key can be easily recovered from some of them. Techniques for resisting key extraction are covered in §4.

3.1 AES-128

AES-128 is specified in FIPS 197 [7]. It is an iterated block cipher that maps a 16-byte input to a 16-byte output using a 16-byte key. It has 10 rounds. Each round updates a 16-byte state variable, which we treat as a one-dimensional array[2], by applying a combination of four basic transformations:

- `AddRoundKey` takes a 16-byte round key, k_r, and uses exclusive-or to add it to the 16-byte state (i.e., $\texttt{state}[i] \leftarrow \texttt{state}[i] \oplus k_r[i]$ for $i = 0 \ldots 15$).
- `SubBytes` utilizes a substitution table, S, that maps bytes to bytes. Each byte of the state is updated by applying S to it (i.e., $\texttt{state}[i] \leftarrow S(\texttt{state}[i])$ for $i = 0 \ldots 15$).
- `ShiftRows` rearranges the bytes of the state using the following permutation:

0 5 10 15	4 9 14 3	8 13 2 7	12 1 6 11

  that is, $\texttt{state}[0], \texttt{state}[5], \texttt{state}[10], \texttt{state}[15]$ form the first four bytes of the updated state, and so on.
- `MixColumns` updates the state four bytes at a time. An invertible 4×4 matrix, MC, with entries from $\mathrm{GF}(2^8)$, is multiplied by a 4×1 column vector formed

[2] The state variable is usually described as a two-dimensional array of bytes (i.e., a 4×4 array). However, the four columns can be concatenated end-to-end to form a one-dimensional array. Using a one-dimensional array simplifies some of our notation and diagrams.

from four state bytes. The state bytes are interpreted as elements of GF(2^8). More precisely, the transformation is

$$\begin{bmatrix} \texttt{state}[i] \\ \texttt{state}[i+1] \\ \texttt{state}[i+2] \\ \texttt{state}[i+3] \end{bmatrix} \leftarrow MC \cdot \begin{bmatrix} \texttt{state}[i] \\ \texttt{state}[i+1] \\ \texttt{state}[i+2] \\ \texttt{state}[i+3] \end{bmatrix}$$

for $i = 0, 4, 8, 12$. The matrix MC is defined to be

$$\begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix}.$$

Let k denote an AES-128 key. The AES specification explains how to expand k into eleven round keys $k_0, k_1, \ldots, k_{10}$ (one additional round key, k_0, is required for an initial `AddRoundKey` operation that takes place before round one). We do not require the exact details of key expansion here; note, however, that k_0 is equal to k.

The conventional way to describe AES-128 encryption is as follows:

$\texttt{state} \leftarrow \textit{plaintext}$
$\texttt{AddRoundKey}(\texttt{state}, k_0)$
for $r = 1 \ldots 9$
    $\texttt{SubBytes}(\texttt{state})$
    $\texttt{ShiftRows}(\texttt{state})$
    $\texttt{MixColumns}(\texttt{state})$
    $\texttt{AddRoundKey}(\texttt{state}, k_r)$
$\texttt{SubBytes}(\texttt{state})$
$\texttt{ShiftRows}(\texttt{state})$
$\texttt{AddRoundKey}(\texttt{state}, k_{10})$
$\textit{ciphertext} \leftarrow \texttt{state}$

However, there are many other valid descriptions. Consider the following two observations:

1. The for-loop can be redefined to bring the transformation $\texttt{AddRoundKey}(\texttt{state}, k_0)$ inside it while pushing $\texttt{AddRoundKey}(\texttt{state}, k_9)$ out.
2. Since `SubBytes` applies the same S-box to each byte of the state, `SubBytes` followed by `ShiftRows` gives the same result as `ShiftRows` followed by `SubBytes`.

From these observations, we can generate the following description:

$\texttt{state} \leftarrow \textit{plaintext}$
for $r = 1 \ldots 9$
    $\texttt{AddRoundKey}(\texttt{state}, k_{r-1})$
    $\texttt{ShiftRows}(\texttt{state})$
    $\texttt{SubBytes}(\texttt{state})$
    $\texttt{MixColumns}(\texttt{state})$
$\texttt{AddRoundKey}(\texttt{state}, k_9)$
$\texttt{ShiftRows}(\texttt{state})$
$\texttt{SubBytes}(\texttt{state})$
$\texttt{AddRoundKey}(\texttt{state}, k_{10})$
$\textit{ciphertext} \leftarrow \texttt{state}$

Here is another observation:

3. Since `ShiftRows` is a linear transformation (recall that it is a permutation), $\texttt{AddRoundKey}(\texttt{state}, k_{r-1})$ followed by $\texttt{ShiftRows}(\texttt{state})$ gives the same result as $\texttt{ShiftRows}(\texttt{state})$ followed by $\texttt{AddRoundKey}(\texttt{state}, \widehat{k}_{r-1})$; here, $\widehat{k}_{r-1}$ is the result of applying `ShiftRows` to the round key k_{r-1}.

This gives us

$\texttt{state} \leftarrow \textit{plaintext}$
for $r = 1 \ldots 9$
    $\texttt{ShiftRows}(\texttt{state})$
    $\texttt{AddRoundKey}(\texttt{state}, \widehat{k}_{r-1})$
    $\texttt{SubBytes}(\texttt{state})$
    $\texttt{MixColumns}(\texttt{state})$
$\texttt{ShiftRows}(\texttt{state})$
$\texttt{AddRoundKey}(\texttt{state}, \widehat{k}_9)$
$\texttt{SubBytes}(\texttt{state})$
$\texttt{AddRoundKey}(\texttt{state}, k_{10})$
$\textit{ciphertext} \leftarrow \texttt{state}$

With AES written in this way, we are able to combine `AddRoundKey`, `SubBytes`, and part of `MixColumns` into a series of table look-ups. This technique is similar to one used by Daemen and Rijmen in their AES proposal document [6, see §5.2.1]. However, in the implementation we are about to review below, we will see that bytes of round keys are embedded into some of the tables, and a number of redundant tables are included; this differs from the implementation by Daeman and Rijmen. Essentially, the for-loop above is unrolled and a collection of tables is created for each of the ten rounds with no regard as to whether or not an identical table might exist elsewhere in the implementation.

3.2 T-boxes

In each round, the `AddRoundKey` and `SubBytes` transformations can be combined into a series of sixteen look-up tables that map bytes to bytes (i.e., 8-bits to 8-

bits). These so-called *T-boxes* are defined as follows:

$$\begin{aligned} T_i^r(x) &= S(x \oplus \widehat{k}_{r-1}[i]), && \text{for } i = 0 \ldots 15 \text{ and } r = 1 \ldots 9, \\ T_i^{10}(x) &= S(x \oplus \widehat{k}_9[i]) \oplus k_{10}[i], && \text{for } i = 0 \ldots 15. \end{aligned}$$

Note that the T-boxes for round 10 incorporate the bytes of two round keys ($\widehat{k}_9$ and k_{10}). There are 160 T-boxes in total.

3.3 Ty_i tables

In rounds 1 to 9, after a byte is mapped through a T-box, it is then input into a `MixColumns` transformation. In particular, in round 1, the outputs of $T_0^1, T_1^1, T_2^1, T_3^1$ are interpreted as a column vector and then multiplied with the matrix *MC*. This computation can also be implemented using tables.

Let x_0, x_1, x_2, x_3 be four bytes that are to be multiplied with *MC*. The multiplication can be decomposed into an exclusive-or of four 32-bit values like so:

$$\begin{bmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = x_0 \begin{bmatrix} \texttt{02} \\ \texttt{01} \\ \texttt{01} \\ \texttt{03} \end{bmatrix} \oplus x_1 \begin{bmatrix} \texttt{03} \\ \texttt{02} \\ \texttt{01} \\ \texttt{01} \end{bmatrix} \oplus x_2 \begin{bmatrix} \texttt{01} \\ \texttt{03} \\ \texttt{02} \\ \texttt{01} \end{bmatrix} \oplus x_3 \begin{bmatrix} \texttt{01} \\ \texttt{01} \\ \texttt{03} \\ \texttt{02} \end{bmatrix}.$$

The terms of the sum on the right (denote them by y_0, y_1, y_2, y_3) are each a function of one byte of input. Thus, each y_i can take on only 256 possible values.

The so-called Ty_i *tables* map 8-bits to 32-bits and are defined as follows:

$$\begin{aligned} Ty_0(x) &= x \cdot [\texttt{02 01 01 03}]^\mathsf{T} \\ Ty_1(x) &= x \cdot [\texttt{03 02 01 01}]^\mathsf{T} \\ Ty_2(x) &= x \cdot [\texttt{01 03 02 01}]^\mathsf{T} \\ Ty_3(x) &= x \cdot [\texttt{01 01 03 02}]^\mathsf{T}. \end{aligned}$$

Using these tables, we see that the 32-bits that result from applying `MixColumns` to the four bytes x_0, x_1, x_2, x_3 can be computed via four table look-ups and three exclusive-ors:

$$Ty_0(x_0) \oplus Ty_1(x_1) \oplus Ty_2(x_2) \oplus Ty_3(x_3).$$

We create 144 Ty_i tables (36 copies of each of Ty_0, Ty_1, Ty_2, Ty_3) to accept the outputs of the T-boxes in rounds 1 to 9 (recall that `MixColumns` is not applied in round 10).

3.4 XOR tables

The exclusive-or operations, which combine 32-bit values from the Ty_i tables, can also be implemented using tables. Define a look-up table, XOR, that takes two nibbles (i.e., two 4-bit values) as input and maps them to their exclusive-or:

$$\text{XOR}(x, y) = x \oplus y.$$

Note that XOR maps 8-bits to 4-bits. The exclusive-or of two 32-bit values can be computed using 8 copies of the XOR look-up table.

In each of rounds 1 to 9, twelve 32-bit exclusive-ors are required to determine the result of `MixColumns`. To carry out this computation, we create 96 copies of the XOR table in each of these rounds (i.e., 864 copies of the XOR table in total). Although it seems that we could make do with only one XOR table, the protections introduced in §4 do not permit this.

3.5 Table Composition

Wherever a T-box feeds directly into a Ty_i table (i.e., in rounds 1 to 9), we can replace the two separate tables with their composition. For example, in round one, T_0^1 and Ty_0 could be replaced with the new look-up table $Ty_0 \circ T_0^1$ where

$$Ty_0 \circ T_0^1(x) = Ty_0(T_0^1(x)).$$

Composing look-up tables reduces the number of individual table accesses required to carry out an encryption. Throughout rounds 1 to 9, the T-boxes and Ty_i tables are composed.

3.6 Summary

We now have all the tables (144 composed T-boxes/Ty_i tables, 864 XOR tables, 16 T-boxes) we need for our implementation, which can be summarized as follows:

```
state ← plaintext
for r = 1...9
    ShiftRows
    TBoxesTyiTables
    XORTables
ShiftRows
TBoxes
ciphertext ← state
```

The flow of four bytes of state through round 1 is illustrated in Figure 1. Note that there are a number of different ways that the XOR tables could be utilized to determine the value of the state variable at the end of rounds 1 to 9; the flow in Figure 1 is only an example. A zoomed in look at the XOR computation is given in Figure 2.

4 Protected Implementation

We consider now how to protect the table-based implementation of the previous section in the white-box attack context. Recall that this means that the software implementing AES-128 encryption for a particular key executes in an environment that is under the control of an attacker. By using a disassembler/debugger, it is easy for the attacker to learn the contents of the various look-up tables, including the composed T-boxes/Ty_i tables that incorporate bytes of round keys.

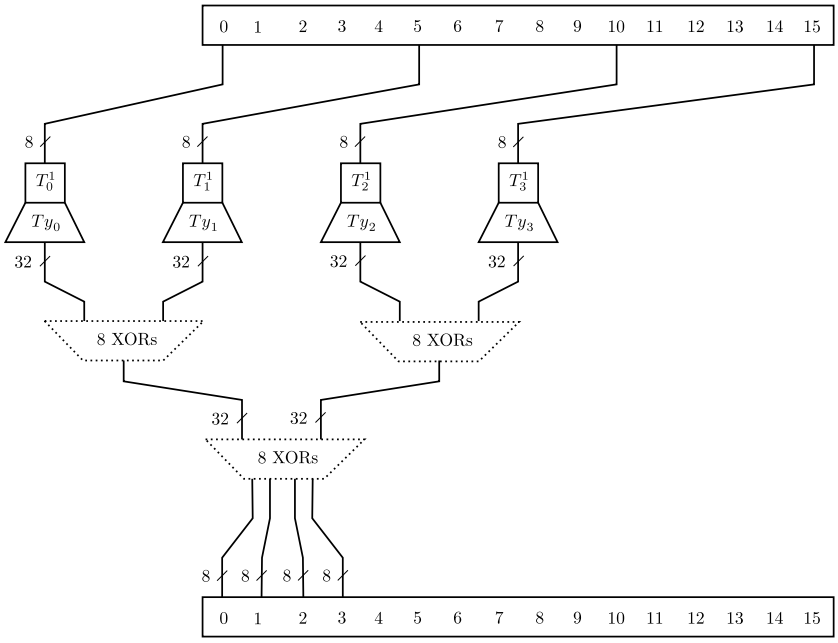

Fig. 1. The data flow for round one of AES with respect to bytes 0,5,10,15 of the input state (i.e., the plaintext). The data flow for the other bytes is similar. Note that the input state is at the top of the diagram and the output state is at the bottom.

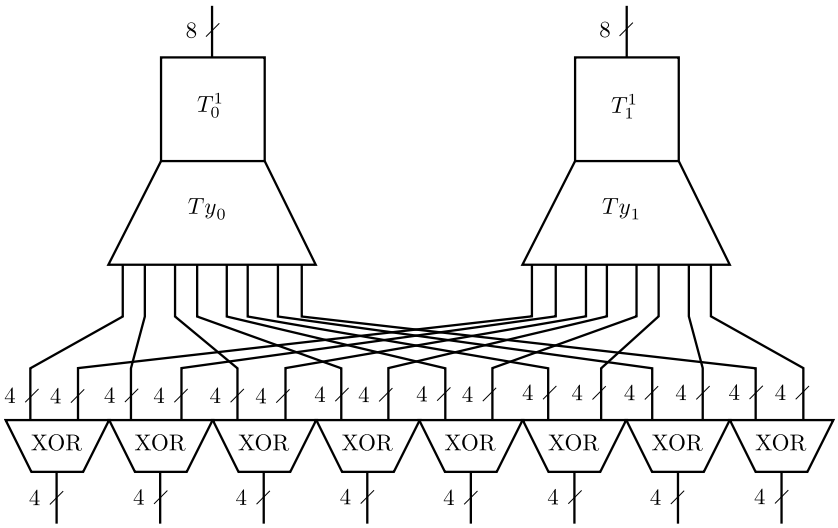

Fig. 2. Computing an exclusive-or of two 32-bit values utilizes eight XOR tables. The inputs enter at the top of the diagram and the outputs appear at the bottom.

4.1 Encodings

If the composed T-box/Ty_i tables from round 1 are known to an attacker, then they can easily recover the AES key. Consider table $Ty_0 \circ T_0^1$, and let a denote the byte of round key k_0 used to build T_0^1. There are only 256 different constructions of $Ty_0 \circ T_0^1$, and thus the attacker can enumerate them and simply look-up the value of a from that list.[3]

Something must be done to protect the contents of the composed T-box/Ty_i tables and the T-boxes in round 10 if the implementation is going to resist key extraction. The technique proposed by Chow et al. [4] is to use *input and output encodings*.

An encoding is simply a bijection. To protect a table, T, we choose bijections f and g and form the new table T' where

$$T' = g \circ T \circ f^{-1}.$$

f is called the *input encoding* and g is called the *output encoding*. This new table maps encoded inputs to encoded outputs and can still be used to compute $T(x)$. To retrieve the value of $T(x)$, we map $f(x)$ through T' and then apply g^{-1} to the result.

If the output of table T feeds into another table R, then encodings are applied to those two tables in a so-called *networked* fashion; that is, the output encoding of T and the input encoding of R are chosen so that they cancel each other out. For example, T and R would be protected as follows,

$$T' = g \circ T \circ f^{-1} \quad \text{and} \quad R' = h \circ R \circ g^{-1},$$

from which we see that

$$R' \circ T' = (h \circ R \circ g^{-1}) \circ (g \circ T \circ f^{-1}) = h \circ (R \circ T) \circ f^{-1}.$$

Encodings are used to obfuscate the contents of all look-up tables in Chow et al.'s AES implementation. They are selected uniformly at random and independently wherever possible. Because of the design of the XOR tables, which consume four bits from one look-up table and four bits from another, almost all the encodings used in the protected implementation are *concatenated encodings*. These are bijections formed from smaller bijections. For example, we can build an 8-bit encoding, f, from two 4-bit encodings, f_0 and f_1, like so:

$$f(x_0 \| x_1) = f_0(x_0) \| f_1(x_1).$$

Here, the symbol $\|$ denotes the concatenation of bit-strings, and x_0, x_1 are 4-bit strings. Similarly, we can build a 32-bit encoding, g, using eight 4-bit encodings:

$$g(x_0 \| x_1 \| \cdots \| x_7) = g_0(x_0) \| g_1(x_1) \| \cdots \| g_7(x_7).$$

Concatenated 4-bit input and output encodings are individually selected and applied to all look-up tables, with the following exceptions:

[3] The attacker can also compute the key byte directly: $a = S^{-1} \circ Ty_0^{-1} \circ (Ty_0 \circ T_0^1)(0)$.

– the output encodings applied to the XOR tables (these are just 4-bit encodings, not concatenated encodings),
– the input encodings applied to the composed T-box/Ty_i tables in round 1,
– the output encodings applied to the T-boxes in round 10.

The encodings mentioned in the latter two items do not have to be networked with XOR tables, so we have more freedom in their selection. We will discuss this further when we consider external encodings in §4.3.

Local security. When the composed T-box/Ty_i tables are protected with encodings, there are now too many table constructions for an attacker to enumerate. Consider a composed T-box/Ty_i table from round 2. There are $(16!)^2 \cdot (16!)^8$ ways of choosing input and output encodings for this table. If the input encoding is fixed, then it can be shown that all of the $(16!)^8$ possible output encodings produce distinct look-up tables. Thus, the number of table constructions is at least $(16!)^8 \approx 2^{354}$ (and is at most $(16!)^{10} \approx 2^{442}$).

An attacker might hope to deduce the key byte from an encoded T-box/Ty_i table by studying the lists of table constructions (i.e., the table constructions for each possible value of the key byte). However, this approach will not yield any information about the key byte. It can be shown, by manipulating input encodings, that all 256 lists of table constructions are the same.

From the previous fact, we can conclude that the protected tables are information theoretically secure. It is not possible for the attacker to extract the key byte from the encoded version of, say, $Ty_0 \circ T_0^2$, if he or she studies only that table; Chow et al. [5] refer to this property as *local security*. However, even though no information about the key leaks from this protected table, other information may. For example, the definition of the output encodings may leak, which could be useful for extracting key bytes from T-box/Ty_i tables in the next round.

Although the use of encodings is actually what motivates the table-based implementation of AES in the first place, encodings are the very last form of protection applied. Note that encodings are selected uniformly at random and will be non-linear with very high probability.

4.2 Mixing Bijections

The look-up tables that incorporate bytes of round keys can be considered miniature block ciphers. The application of concatenated input and output encodings help these components achieve *confusion*, as defined by Shannon [15]. To help them achieve *diffusion*, linear transformations are also composed at their input and output (these compositions are done before the application of the concatenated input and output encodings). An invertible linear transformation is referred to as a *mixing bijection*.

Mixing bijections are applied to all the key-dependent look-up tables in the implementation. In general, each mixing bijection is selected uniformly at random. Their usage in each of the interior rounds (i.e., rounds 2 to 9) is the same;

for the exterior rounds (i.e., round 1 and 10), there are a few differences, which we will explain.

We begin by selecting all the required mixing bijections:

- for each of rounds 2 to 10, select 16 8-bit to 8-bit mixing bijections (i.e., 144 mixing bijections in total). These will be composed at the input of each T-box in rounds 2 to 10.
- for each of rounds 1 to 9, select 4 32-bit to 32-bit mixing bijections (i.e., one mixing bijection for each of the four matrix multiplication steps in each of those rounds). These will be composed at the output of each Ty_i table in rounds 1 to 9.

Note that mixing bijections can be selected uniformly at random by constructing invertible matrices over GF(2) (i.e., build a random matrix, test it for invertibility, and repeat if necessary). Now consider, for example, the first four key-dependent look-up tables in round 2:

$$Ty_0 \circ T_0^2,$$
$$Ty_1 \circ T_1^2,$$
$$Ty_2 \circ T_2^2,$$
$$Ty_3 \circ T_3^2.$$

Let $L_0^2, L_1^2, L_2^2, L_3^2$ be the four 8-bit to 8-bit mixing bijections selected for these tables. The inverses of these transformations are composed at their input. Let MB be the 32-bit to 32-bit mixing bijection chosen for these four tables.[4] MB is composed at the output of each table. This produces the following tables:

$$MB \circ Ty_0 \circ T_0^2 \circ {L_0^2}^{-1},$$
$$MB \circ Ty_1 \circ T_1^2 \circ {L_1^2}^{-1},$$
$$MB \circ Ty_2 \circ T_2^2 \circ {L_2^2}^{-1},$$
$$MB \circ Ty_3 \circ T_3^2 \circ {L_3^2}^{-1}.$$

The four bytes entering these tables are computed in round 1 and will have the appropriate mixing bijections applied to them there (i.e., the four inputs are respectively encoded using $L_0^2, L_1^2, L_2^2, L_3^2$ in round 1).

The outputs of these tables feed into the XOR tables, as in Figure 1. However, at the end of the third stage of XOR tables, the resulting 32-bit value is now

$$MB \circ MC\,[z_0\ z_1\ z_2\ z_3]^{\mathsf{T}}.$$

We need to remove the transformation MB, and also apply the 8-bit mixing bijections required for the next round.

[4] Chow et al. recommend that the MB matrices be selected with some additional properties; see Appendix B for a commentary on this.

Four new 8-bit to 32-bit tables are introduced to remove the effect of MB. These tables are generated using the familiar technique of decomposing a matrix multiplication into an exclusive-or of four 32-bit vectors:

$$MB^{-1}\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = MB^{-1}\begin{bmatrix} z_0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ z_1 \\ 0 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ 0 \\ z_2 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ 0 \\ 0 \\ z_3 \end{bmatrix}.$$

Let $MB_0^{-1}, MB_1^{-1}, MB_2^{-1}, MB_3^{-1}$ denote the four 8-bit to 32-bit tables corresponding to the terms of the sum on the right. Let L^3 denote a 32-bit to 32-bit mixing bijection constructed by concatenating four 8-bit to 8-bit mixing bijections from round 3. L^3 is used to put the proper encodings on bytes 0,1,2,3 of the state array that enters round 3. Accounting for the `ShiftRows` transformation at the beginning of round 3, L^3 is defined as follows:

$$L^3 = L_0^3 \| L_{13}^3 \| L_{10}^3 \| L_7^3.$$

L^3 is composed at the output of each of $MB_0^{-1}, MB_1^{-1}, MB_2^{-1}, MB_3^{-1}$ resulting in the tables

$$L^3 \circ MB_0^{-1},$$
$$L^3 \circ MB_1^{-1},$$
$$L^3 \circ MB_2^{-1},$$
$$L^3 \circ MB_3^{-1}.$$

The outputs of these tables are combined using three new stages of XOR tables. A summary of this process is presented in Figure 3. In comparison with Figure 1, we see that the number of tables has doubled.

The application of mixing bijections in round 1 is very similar to Figure 3. The only difference is that there are no input mixing bijections applied to the T-boxes. In round 10, there are input mixing bijections applied to the T-boxes, but output mixing bijections are not applied. The reason for these differences is related to the external encodings, which we discuss next.

4.3 External encodings

One question that often arises when considering the white-box attack context is this: why would an attacker want to extract the cipher key when they already have software that will decrypt ciphertext for them? The answer, given by Chow et al., is to design the implementation so that it does not map raw ciphertext to raw plaintext, but rather *encoded ciphertext* to *encoded plaintext*. Encodings that affect the input and output of the cipher are referred to as *external encodings*.

Denote an AES-128 decryption by D_k. Select 128-bit to 128-bit bijections F and G. Chow et al. recommend that the implementation computes

$$D'_k = G \circ D_k \circ F^{-1}.$$

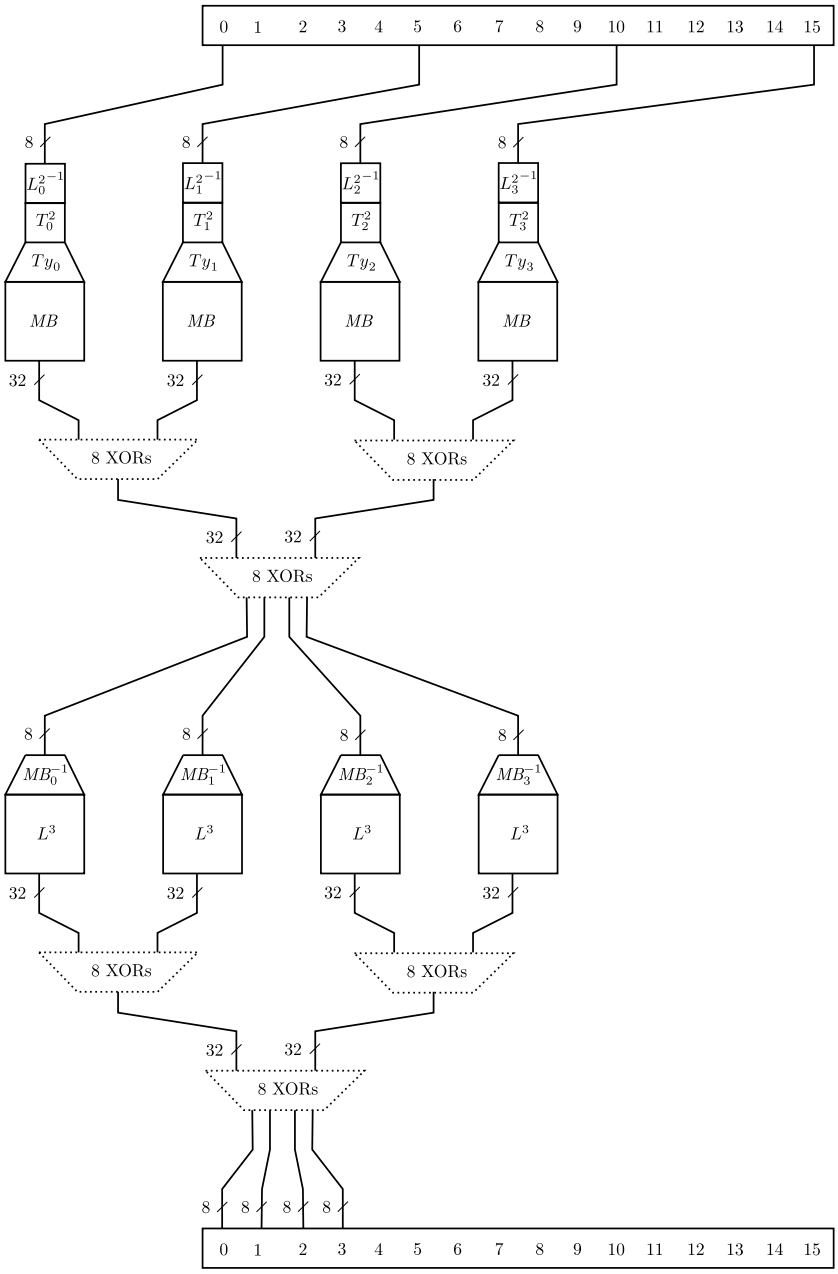

Fig. 3. The application of mixing bijections in round 2. The picture for rounds 3 to 9 is the same. For round 1, the only difference is the absence of the input mixing bijections on the T-boxes. For round 10, mixing bijections are applied at the input of the T-boxes, but not the output. At the bottom of the diagram, note that bytes 0,1,2,3 of the state array that enters round 3 feed T-boxes $T_0^3, T_{13}^3, T_{10}^3, T_7^3$, respectively.

As with the encodings discussed in §4.1, a ciphertext, x, must be encoded with F, which gives $F(x)$, before it is passed as input to the implementation. In our DRM example, this external encoding could be applied on the server that supplies premium content for downloading (i.e., on a host separate from the one where the client software runs). The DRM client software, running on the user's device, receives $F(x)$ and applies D'_k to give the result $G(D_k(x))$. The remaining encoding could be removed by the content viewer, perhaps as the content is played.

Use of external encodings is mainly to ensure that there is no point during the execution of the client software where raw ciphertext and raw plaintext appear. Chow et al. suggest that the external encodings be 128-bit to 128-bit mixing bijections. Using external encodings such as these requires that a number of new tables be added to the protected implementation: 16 8-bit to 128-bit tables along with supporting XOR tables (480) to compute the matrix multiplication for F^{-1} prior to round 1, and similarly for the matrix multiplication for G after round 10 (note that the 8-bit to 128-bit tables for G can be composed with the round 10 T-boxes).

Here, to simplify our exposition, we will not use mixing bijections for the external encodings. Instead, we will use concatenated 8-bit encodings; that is,

$$F = F_0 \| F_1 \| \cdots \| F_{15} \quad \text{and} \quad G = G_0 \| G_1 \| \cdots \| G_{15},$$

where each F_i and G_i is an 8-bit to 8-bit bijection (selected uniformly at random). For each T-box in round 1, the corresponding ${F_i}^{-1}$ is composed at its input. And for each T-box in round 10, the corresponding G_i is composed at its output. External encodings of this type do not require the addition of any new tables to the implementation.

4.4 Summary

To protect the tabled-based AES implementation from §3, we apply mixing bijections, then (internal) encodings, and then external encodings. After the application of mixing bijections, the number of look-up tables in rounds 1 to 9 doubles. The table counts are as follows:

288	8-bit to 32-bit tables (1024 bytes each),
1728	8-bit to 4-bit tables (128 bytes each),
16	8-bit to 8-bit tables (256 bytes each).

The total storage requirement for the tables is 508 KB. A summary of the various tables, with all protections applied to them, is given in Figure 4.

5 Cryptanalysis

Chow et al. [4] present some interesting attacks on weakened variants of their protected implementation, which justify some of their design choices. In particular, they show that if external encodings are not used, then a linear relation

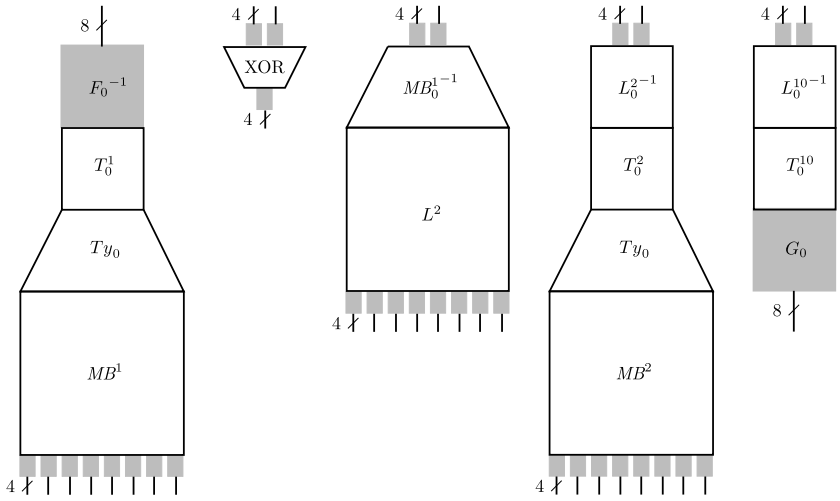

Fig. 4. Representatives from the five classes of protected look-up tables. Grey boxes are used to denote input and output encodings (big grey boxes are external encodings). For each representative (going from left to right), we list the rounds where tables like it can be found: round 1 only, rounds 1 to 9, rounds 1 to 9, rounds 2 to 9, round 10 only.

amongst round key bytes can be found that makes a key search feasible (the search space is reduced from 2^{128} to 2^{32}). They also show that if mixing bijections are not applied (recall that the inclusion of mixing bijections roughly doubles the number of look-up tables), then it is possible to deduce the output encodings for any key-dependent look-up table in rounds 2-9 (see Appendix A for details on this). Knowledge of those output encodings in, say, round 2 leads to discovery of the input encodings in round 3 because the encodings on the XOR tables at the end of round 2 can be deduced. Now, with knowledge of the input and output encodings in round 3, round key bytes can be easily extracted.

In 2004, Billet, Gilbert and Ech-Chatbi [2] published an algebraic attack against Chow et al.'s first AES implementation. They showed that the cipher key can be extracted using at most 2^{30} work-steps and negligible memory. We give a brief review of their method here.

5.1 The BGE attack

As is illustrated in Figure 3 in the previous section, an AES round can be interpreted as the parallel application of four 32-bit to 32-bit transformations to the state array. Although the mixing bijection MB is present in the protected implementation, it has no influence on the four bytes output at the bottom of Figure 3. The effect of MB and any other internal encodings are canceled out

(this is by design), and the 32-bit to 32-bit transformation has the form displayed in the lefthand diagram of Figure 5.

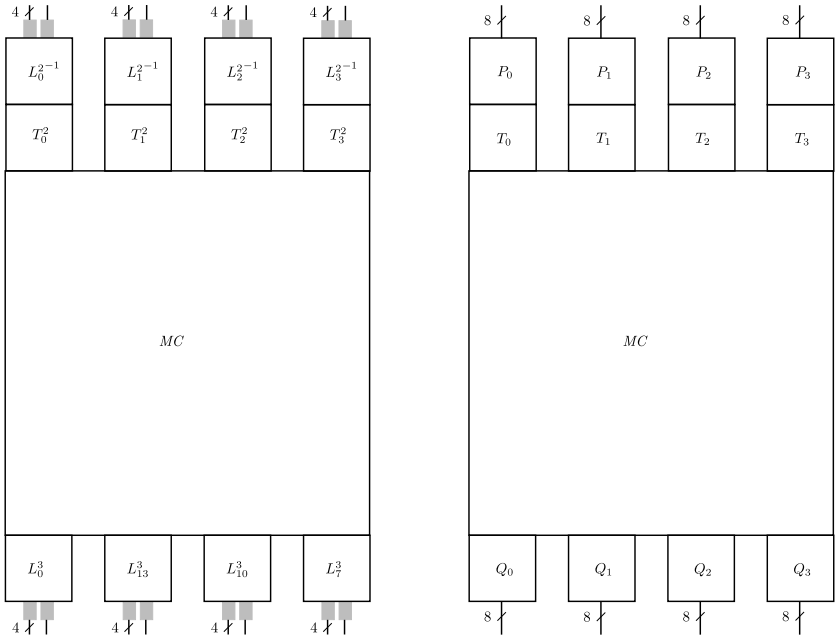

Fig. 5. The left diagram displays one of the 32-bit to 32-bit transforms applied in round 2. *MC* is the `MixColumns` matrix. The notation used matches that of the previous figures. The right diagram introduces a more general notation used by Billet et al. [2] where the mixing bijections and concatenated encodings are combined. Note that the inverses of the output encodings become input encodings in the subsequent round.

The righthand diagram of Figure 5 introduces the notation used by Billet et al. when they examine one of four 32-bit to 32-bit transforms in round 2. The P_i's are the combination of input encodings and mixing bijections; the Q_i's are the combination of mixing bijections and output encodings. Let x_0, x_1, x_2, x_3 denote four input bytes and let y_0, y_1, y_2, y_3 denote the resulting output. From the definition of the matrix *MC*, the relation between the inputs and outputs can be summarized like so:

$$y_0 = Q_0\big(\texttt{02} \cdot T'_0(x_0) \oplus \texttt{03} \cdot T'_1(x_1) \oplus \texttt{01} \cdot T'_2(x_2) \oplus \texttt{01} \cdot T'_3(x_3)\big), \tag{1}$$
$$y_1 = Q_1\left(\texttt{01} \cdot T'_0(x_0) \oplus \texttt{02} \cdot T'_1(x_1) \oplus \texttt{03} \cdot T'_2(x_2) \oplus \texttt{01} \cdot T'_3(x_3)\right), \tag{2}$$
$$y_2 = Q_2\left(\texttt{01} \cdot T'_0(x_0) \oplus \texttt{01} \cdot T'_1(x_1) \oplus \texttt{02} \cdot T'_2(x_2) \oplus \texttt{03} \cdot T'_3(x_3)\right), \tag{3}$$
$$y_3 = Q_3\left(\texttt{03} \cdot T'_0(x_0) \oplus \texttt{01} \cdot T'_1(x_1) \oplus \texttt{01} \cdot T'_2(x_2) \oplus \texttt{02} \cdot T'_3(x_3)\right); \tag{4}$$

here, T'_i is short for $T_i \circ P_i$. Note that each y_i is a function of x_0, x_1, x_2, x_3.

Billet et al. found that information about the output encodings (i.e., the Q_i's) leak from the four identities above. For each Q_i, they showed that it is possible to build an approximation, $\widetilde{Q_i}$, that differs from it by an unknown affine transformation; that is,

$$\widetilde{Q_i} = Q_i \circ A_i,$$

where A_i consists of an invertible linear transformation followed by an exclusive-or with a constant.

The approximations are built by analyzing a new set of look-up tables derived from Figure 5. As noted previously, y_0 is a function of x_0, x_1, x_2, x_3; that is, $y_0 = f(x_0, x_1, x_2, x_3)$. However, if x_2, x_3 are kept constant, then y_0 can be considered a function of only x_0 and x_1. Fix x_2, x_3 to $\mathtt{00}, \mathtt{00}$ and let $f_{x_1}(x_0)$ denote the function $f(x_0, x_1, \mathtt{00}, \mathtt{00})$. Build the look-up table for $y_0 = f_{\mathtt{00}}(x_0)$ and for $y_0 = f_{\mathtt{01}}(x_0)$. From equation (1), we see that

$$\begin{aligned} f_{\mathtt{00}}(x_0) &= Q_0(\mathtt{02} \cdot T_0'(x_0) \oplus \beta_{\mathtt{00}}), \\ f_{\mathtt{01}}(x_0) &= Q_0(\mathtt{02} \cdot T_0'(x_0) \oplus \beta_{\mathtt{01}}), \end{aligned}$$

where $\beta_{\mathtt{00}}$ and $\beta_{\mathtt{01}}$ are unknown 8-bit strings.

From the look-up tables for $f_{\mathtt{00}}$ and $f_{\mathtt{01}}$, we can construct the look-up table for $f_{\mathtt{01}} \circ {f_{\mathtt{00}}}^{-1}$, which has a very simple description. Using functional notation, we can write

$$\begin{aligned} f_{\mathtt{00}} &= Q_0 \circ \oplus_{\beta_{\mathtt{00}}} \circ \mathtt{02} \cdot T_0', \\ f_{\mathtt{01}} &= Q_0 \circ \oplus_{\beta_{\mathtt{01}}} \circ \mathtt{02} \cdot T_0'. \end{aligned}$$

Thus, we see that

$$\begin{aligned} f_{\mathtt{01}} \circ {f_{\mathtt{00}}}^{-1} &= (Q_0 \circ \oplus_{\beta_{\mathtt{01}}} \circ \mathtt{02} \cdot T_0') \circ ((\mathtt{02} \cdot T_0')^{-1} \circ \oplus_{\beta_{\mathtt{00}}} \circ {Q_0}^{-1}) \\ &= Q_0 \circ \oplus_{\beta} \circ {Q_0}^{-1}, \end{aligned}$$

where $\beta = \beta_{\mathtt{01}} \oplus \beta_{\mathtt{00}}$.

There are exactly 256 bijections of the form $Q_0 \circ \oplus_\delta \circ {Q_0}^{-1}$ where δ is an 8-bit string. This set of bijections forms a commutative group under composition, denoted by $(G, \circ)$. All the group elements (i.e., look-up tables) are generated by computing the following compositions

$$f_{\mathtt{00}} \circ {f_{\mathtt{00}}}^{-1}, f_{\mathtt{01}} \circ {f_{\mathtt{00}}}^{-1}, \ldots, f_{\mathtt{ff}} \circ {f_{\mathtt{00}}}^{-1}.$$

It is not difficult to find 8 group elements, $g_1, g_2, \ldots, g_8$, such that a subset of them can be composed to generate any group element. These elements act like a vector-space basis for $(G, \circ)$ and can be used to build an isomorphism $\psi : (G, \circ) \to (\mathrm{GF}(2)^8, \oplus)$. Without going into further detail (see [2, Theorem 1]), it is this isomorphism that is used to construct the approximation to Q_0: for any $g \in G$, we set

$$\widetilde{Q_0}(\psi(g)) = g(\mathtt{00}).$$

Thus, $\widetilde{Q_0}$ is constructed as a look-up table. An analogous process builds approximation for Q_1, Q_2, Q_3.

Returning to Figure 5, with the approximations at hand, the output encodings in the diagram can be simplified. The simplification is done by composing new output encodings with the existing ones. After each Q_i, the bijection $\widetilde{Q_i}^{-1}$ is applied. These two bijections compose to give

$$\widetilde{Q_i}^{-1} \circ Q_i = {A_i}^{-1} \circ {Q_i}^{-1} \circ Q_i = {A_i}^{-1}.$$

The inverse of an affine bijection is an affine bijection, and so the new output encodings are much simpler than the original (very likely non-linear) ones. And since the output encodings in a given round correspond to input encodings in the subsequent round, the output encoding approximations also lead to input encoding approximations. Thus, the input encodings in Figure 5 can also be simplified.

From Billet et al.'s approximations, we can continue under the assumption that the P_i's and Q_i's in Figure 5 are affine bijections. Each Q_i has the form $M_i(x) \oplus q_i$, where M_i is a linear bijection and q_i is an 8-bit string. By setting x_1, x_2, x_3 all to `00` in equations (1)-(4), an explicit linear relation between M_0 and each of M_1, M_2, M_3 can be derived. Thus, if M_0 is recovered, then so too will M_1, M_2, M_3.

The next step in the attack recovers M_0 and q_0 (see the original paper for details). Thus, the value of the output encodings can be determined completely. With complete knowledge of Q_0, Q_1, Q_2, Q_3, then from Figure 5 we see that it is possible to compute the outputs of the T-boxes. From the complete knowledge of the output encodings in the previous round, we also learn P_0, P_1, P_2, P_3 completely. Now, it is easy to extract the key bytes, as discussed in §4.1.

The time complexity of Billet et al.'s key extraction attack is dominated by the work required to build the approximation to each output encoding. They estimate this to be 2^{24} work-steps. Thus, computing approximations for an entire AES round is $16 \cdot 2^{24} = 2^{28}$ work-steps. They recommend computing approximations for three consecutive AES rounds ($3 \cdot 2^{28} < 2^{30}$ work-steps), which leads to the recovery of two complete round keys, so that any ambiguity in the order of the key bytes recovered from the tables can be eliminated.

6 Remarks

White-box cryptography was introduced in the academic literature by Chow, Eisen, Johnson and van Oorschot 10 years ago and is still a relatively new area of research, with plenty of real-world applications and room for new contributions. For those interested in working in this area, a good understanding of the original white-box AES implementation [4] and the BGE attack [2] are essential, and hopefully this tutorial can help provide that. Although the BGE attack permits the key to be extracted from Chow et al.'s original white-box AES implementation, the attack has served mainly as motivation for work on stronger

white-box implementations, and this line of research has been particularly active in the last few years (e.g., [13], [19]).

7 Acknowledgements

The author thanks Phil Eisen who, over a number of conversations and presentations at Irdeto, motivated the style of exposition on AES in §3. Thanks are also extended to Michael Wiener who provided valuable comments on a preliminary draft of this work (especially with regards to the local security of the composed T-box/Ty_i tables). Also, conversations on white-box cryptography with Jeremy Clark, Alfred Menezes and Anil Somayaji were helpful in directing some of our commentary. Thanks also go to Elif Bilge Kavun who pointed out a notational error in a previous version of §4.2.

References

1. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs (Extended Abstract). In "Advances in Cryptology – CRYPTO 2001: 21st Annual International Cryptology Conference", *Lecture Notes in Computer Science* **2139** (2001), 1–18. Full version available from http://eccc.hpi-web.de/report/2001/057/.
2. O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In "Selected Areas in Cryptography: 11th International Workshop, SAC 2004", *Lecture Notes in Computer Science* **3357** (2005), 227–240.
3. D. Boneh, R. DeMillo, and R. Lipton. On the importance of checking cryptographic protocols for faults. *Journal of Cryptology* **14** (2001), 101–119.
4. S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot. White-Box Cryptography and an AES Implementation. In "Selected Areas in Cryptography: 9th Annual International Workshop, SAC 2002", *Lecture Notes in Computer Science* **2595** (2003), 250–270.
5. S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot. A White-box DES Implementation for DRM Applications. In "Digital Rights Management: ACM CCS-9 Workshop, DRM 2002", *Lecture Notes in Computer Science* **2696** (2003), 1–15.
6. J. Daemen and V. Rijmen. AES submission document on Rijndael, Version 2, September 1999. Available from http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf
7. FIPS 197. *Advanced Encryption Standard*. Federal Information Processing Standards Publication 197, U.S. Department Of Commerce / National Institute of Standards and Technology, 2001. Available from http://www.csrc.nist.gov/publications/fips/
8. L. Goubin, J.-M. Masereel, and M. Quisquater. Cryptanalysis of White-Box DES Implementations. In "Selected Areas in Cryptography: 14th International Workshop, SAC 2007", *Lecture Notes in Computer Science* **4876** (2007), 278–295.
9. S. Hohenberger, G. Rothblum, A. Shelat, and V. Vaikuntanathan. Securely Obfuscating Re-Encryption. In "Theory of Cryptography: 4th Theory of Cryptography Conference, TCC 2007", *Lecture Notes in Computer Science* **4392** (2007), 233–252.

10. M. Karroumi. Protecting White-Box AES with Dual Ciphers. In "Information Security and Cryptology – ICISC 2010", *Lecture Notes in Computer Science* **6829** (2010), 278–291.
11. P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In "Advances in Cryptology – CRYPTO '96", *Lecture Notes in Computer Science* **1109** (1996), 104–113.
12. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In "Advances in Cryptology – CRYPTO '99", *Lecture Notes in Computer Science* **1666** (1999), 388–397.
13. W. Michiels and P. Gorissen. "Cryptographic Method for a White-Box Implementation". U.S. Patent Application 2010/0080395 A1, filed November 9, 2007.
14. W. Michiels and P. Gorissen. "Cryptographic System". U.S. Patent Application 2011/0116625 A1, filed March 2, 2009.
15. C. E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal* **28** (1949), 656–715.
16. B. Wyseur. "White-Box Cryptography", PhD thesis, Katholieke Universiteit Leuven, 2009.
17. B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In "Selected Areas in Cryptography: 14th International Workshop, SAC 2007", *Lecture Notes in Computer Science* **4876** (2007), 264–277.
18. J. Xiao and Y. Zhou. Generating Large Non-Singular Matrices over an Arbitrary Field with Blocks of Full Rank. Cryptology ePrint Archive: Report 2002/096, 2002. Available from http://eprint.iacr.org/2002/096.
19. Y. Xiao and X. Lai. A Secure Implementation of White-Box AES. In "2009 2nd International Conference on Computer Science and its Applications: CSA 2009", IEEE (2009), 6 pages.

A Mixing Bijections are Necessary

Suppose that mixing bijections are not used to protect key-dependent look-up tables and they are instead protected using only concatenated 4-bit input and output encodings. In this case, the white-box implementation would look similar to what is depicted in Figure 1 (see §3.6), with the exception that the input and output encodings are not shown there. A protected T-box/Ty_i table from round 2 is illustrated in Figure 6. Chow et al. show that the output encodings can be easily extracted from this table using frequency analysis.

Consider the three functions from bytes to bytes that consist of an S-box evaluation followed by multiplication (in the AES field GF(2^8)) with one of the `MixColumns` coefficients:

$$\texttt{01} \cdot S, \quad \texttt{02} \cdot S, \quad \texttt{03} \cdot S.$$

Each of these functions can be represented as a 16-by-16 array where rows and columns are indexed using 4-bit values x, y; for each x and y, the value of the function at $x \| y$ is stored at cell (x, y). The distribution of left and right nibbles in each row and each column of an array can be used to form what Chow et al. call *frequency signatures*.

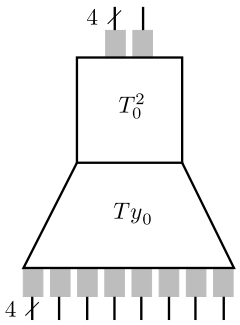

Fig. 6. A composed T-box/Ty_i table from round 2 protected using only input and output encodings.

63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75

Fig. 7. The first four rows of the AES S-box.

To compute the frequency signature for the first row of $\texttt{01} \cdot S$ consider Figure 7. The sequence of left nibbles from the first row is

6, 7, 7, 7, F, 6, 6, C, 3, 0, 6, 2, F, D, A, 7.

The number of times each possible nibble value (0-F) occurs in this sequence is summarized in the following table:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	0	1	1	0	0	4	4	0	0	1	0	1	1	0	2

The first half of the frequency signature is formed by sorting the (left) nibble-count sequence in descending order: 4421111110000000. The second half of the signature is formed similarly from the sequence of right nibbles. The complete frequency signature for the first row of $\texttt{01} \cdot S$ is

44211111100000004311111111100000.

Column signatures are similarly defined. Because of the sorting step, it is easily seen that even if $\texttt{01} \cdot S$ was obfuscated using 4-bit output encodings each of its row and column signatures would remain the same.

To learn the output encodings in Figure 6, we first build two lists R and C. For $0 \leq i \leq 15$, entry i in list R consists of the row-i frequency signatures from each of $\texttt{01} \cdot S, \texttt{02} \cdot S, \texttt{03} \cdot S$ (i.e., each entry consists of an ordered-triple of frequency signatures). And similarly, for $0 \leq i \leq 15$, entry i in list C consists of

the column-i frequency signatures from each of $\texttt{01} \cdot S, \texttt{02} \cdot S, \texttt{03} \cdot S$. In practice, R and C might simply be 16-line text files with three frequency signatures on each line. Once R and C have been constructed, it can be easily verified that their entries are all distinct[5]; moreover, if each list entry is split into 6 half-signatures and considered as a set (rather than an ordered 6-tuple), they are also distinct (i.e., they form 32 distinct sets).

Let U denote a look-up table that enumerates the outputs of Figure 6 that has been extracted from a white-box implementation. For input nibbles x, y, let $U[x][y]$ denote the resulting output, which is a sequence of 8 nibbles. There may be some ambiguity as to how the nibbles are ordered (i.e., which nibbles are affected by a particular `MixColumns` coefficient, and which are left-nibble outputs and which are right-nibble outputs), but this can be eliminated using the procedure presented below. Consider an input nibble x. In Figure 6, x passes through an input encoding, call it ${f_1}^{-1}$, and is then xor-ed with a nibble of round-key, say k_1, before it reaches the S-box. Let h_1 denote the composition of those two bijections (i.e., $h_1 = \oplus_{k_1} \circ {f_1}^{-1}$). The map h_1 is unknown; however, for any x_0 in the interval $0 \ldots 15$ we can deduce the value of $h_1(x_0)$ through the following steps:

1. collect all outputs of the form $U[x_0][y]$ where $0 \leq y \leq 15$.
2. from those 16 outputs, compute 8 half-signatures. At most 6 of those half-signatures will be distinct (recall that the `MixColumns` coefficient `01` is used twice).
3. search lists R and C for the entry i that contains all the computed half-signatures (there will be exactly one matching entry).
4. conclude that $h_1(x_0) = i$.

By repeating these steps, we learn h_1 completely. Note that the matches found in step 3 will resolve any ambiguity about the nibble-sequence of U. Note further that the matches will either all come from R or all from C — by fixing x and letting y vary, we collect outputs from either rows or columns exclusively.

For an input nibble y, let h_2 be defined analogously to h_1; i.e., $h_2 = \oplus_{k_2} \circ {f_2}^{-1}$ where ${f_2}^{-1}$ is the input encoding that affects y and k_2 is a round-key nibble. The map h_2 can be deduced similarly; for any y_0 in the interval $0 \ldots 15$ we collect the 16 outputs of the form $U[x][y_0]$, compute 8 half-signatures and then search R or C for a match. If all the matches occurred in R previously, then they will now occur in C (and vice versa).

With h_1 and h_2 known, the output encodings can be easily derived. For example, four nibbles of the output $U[{h_1}^{-1}(0)][{h_2}^{-1}(0)]$ encode the left and right nibbles of $\texttt{01} \cdot S(0\|0) = 63$. Let g_1, g_2, g_3, g_4 denote the output encodings for those four nibbles; thus we learn the values $g_1(6), g_2(3), g_3(6), g_4(3)$.

We can now derive all the output encodings on the T-box/Ty_i tables in round 2. As mentioned at the beginning of §5, these output encodings correspond to input encodings on XOR tables (see Figure 1). The output encodings on those XOR tables can therefore be derived, which also reveals the input encodings on

[5] For example, the two lists can be sorted and merged.

subsequent XOR tables. Continuing in this manner, we see that all the input and output encodings on the XOR tables in round 2 can be extracted. The final set of output encodings in round 2 reveals the input encodings on the T-box/Ty_i tables in round 3. The output encodings on those T-box/Ty_i tables can be derived using the procedure above. With knowledge of the input and output encodings on each round 3 T-box/Ty_i table, their round-key bytes can be easily extracted.

B Mixing Bijection Selection

The probability that a randomly constructed 32×32 matrix over GF(2) is invertible is

$$\frac{(2^{32}-1)(2^{32}-2)\cdots(2^{32}-2^{31})}{2^{32\cdot 32}} = \prod_{i=0}^{31}\left(1-2^{i-32}\right) \approx 0.288.$$

Thus, if we repeatedly create a random binary matrix and use row-reduction to determine if it is invertible, then we expect to get an invertible matrix after about $1/0.288 = 3.47$ iterations on average. However, for the mixing bijections, *MB*, that are composed at the output of each T-box/Ty_i table in rounds 1 to 9, Chow et al. recommend that those matrices satisfy an additional property aside from invertibility: each *MB* should be built out of invertible 4×4 blocks [4, p. 260]. We explain how to create such matrices and consider the type of information leakage they are intended to protect against.

There are $(2^4-1)(2^4-2)(2^4-4)(2^4-8) = 20160$ invertible 4×4 matrices over GF(2), and so there are 20160^{64} different block matrices of the form

$$\begin{bmatrix} B_{11} & B_{12} & \dots & B_{18} \\ B_{21} & B_{22} & \dots & B_{28} \\ \vdots & \vdots & & \vdots \\ B_{81} & B_{82} & \dots & B_{88} \end{bmatrix}$$

where each B_{ij} is an invertible 4×4 block. A matrix constructed this way may fail to be invertible (consider the case where all B_{ij} are equal). However, empirical results suggest that the density of invertible block matrices is approximately the same as the density of invertible binary matrices.[6] Thus, we can simply create random 32×32 block matrices until we find one that is invertible. As before, we expect to be successful after about 3.47 iterations on average. An alternative method for generating invertible block matrices using induction is presented by Xiao and Zhou [18].

Consider now the 8-bit to 32-bit tables from Figure 4 used to apply MB^{-1} and the mixing bijections required in the next round. If we ignore the 4-bit input

[6] Out of a sample of 10^6 randomly generated block matrices, we found that 288379 ($\approx 28.8\%$) were invertible.

and output encodings, then each table reduces to a linear transformation of the form

$$\begin{bmatrix} L_1 & 0 & 0 & 0 \\ 0 & L_2 & 0 & 0 \\ 0 & 0 & L_3 & 0 \\ 0 & 0 & 0 & L_4 \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} y = \begin{bmatrix} L_1 A_1 y \\ L_2 A_2 y \\ L_3 A_3 y \\ L_4 A_4 y \end{bmatrix};$$

here, y is the 8-bit input, L_1, L_2, L_3, L_4 are the 8×8 mixing bijections and A_1, A_2, A_3, A_4 are 8×8 blocks taken from MB^{-1}. Each product $L_i A_i$ is an 8×8 matrix. Consider the matrix $L_1 A_1$ and partition it into four 4×4 blocks. Let y_0 and y_1 be the two nibbles of y; we have

$$L_1 A_1 y = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} B_{11} \\ B_{21} \end{bmatrix} y_0 + \begin{bmatrix} B_{21} \\ B_{22} \end{bmatrix} y_1 = \begin{bmatrix} z_0 \\ z_1 \end{bmatrix}.$$

An attacker can examine the outputs that result when y_1 is held constant and y_0 varies (the 4-bit input encodings permit this). Let z_0 and z_1 denote the nibbles of the output. The base-2 logarithm of the number of different z_0 and z_1 values gives, respectively, the rank of B_{11} and the rank of B_{21}. The rank of the other two blocks can be similarly deduced.

Chow et al. note that low rank blocks can leak information about output encodings [4, p. 263]. They consider the extreme case where blocks have rank 0. Suppose this is true for B_{11} and B_{12}; then the output nibble z_0 would always be zero, and so the attacker would learn the 4-bit encoding of 0. Extracting encoding information when the ranks are nonzero seems more difficult. However, the number of blocks of rank 1 is 225, and the number of rank 2 is 7350; these small sets make attractive targets for some type of exhaustive search.

Note that the combination of L_i and A_i together determines whether the product $L_i A_i$ can be partitioned into invertible blocks. Even if L_i and A_i are each built out of invertible blocks, this is not necessarily true of their product. Similar reasoning applies to the 8-bit to 32-bit tables that incorporate MB. Even if MB is constructed out of invertible blocks (as is recommended), this is not necessarily true for the product of MB and a column of the `MixColumns` matrix, MC. Thus, it is not immediately clear what the benefit of imposing this restriction on MB is. It may be more appropriate to construct MB^{-1} of out invertible blocks (and note that if an invertible matrix A is constructed out of invertible blocks, then this is not necessarily true of A^{-1}). However, whatever structure or properties the matrices MB and MB^{-1} might have make no difference in the BGE attack.