SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2016; **9**:996–1006
Published online 16 December 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1394

RESEARCH ARTICLE

A secure white-box SM4 implementation

Kunpeng Bai[1,2] and Chuankun Wu[1]*

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] University of Chinese Academy of Sciences, Beijing 100049, China

ABSTRACT

White-box cryptography aims at implementing a cipher to protect its key from being extracted in a white-box attack context, where an attacker has full control over dynamic execution of the cryptographic software. So far, most white-box implementations exploit lookup-table-based techniques and have been broken because of a weakness that the embedded large linear encodings are cancelled out by compositions of lookup tables. In this paper, we propose a new lookup-table-based white-box implementation for the Chinese block cipher standard SM4 that can protect the large linear encodings from being cancelled out. Our implementation, which can resist a series of white-box attacks, requires 32.5 MB of memory to store the lookup tables and is about nine times as fast as the previous Xiao–Lai white-box SM4 implementation. Copyright © 2015 John Wiley & Sons, Ltd

KEYWORDS
white-box cryptography; SM4; secure implementation; lookup tables; obfuscation

***Correspondence**
Chuankun Wu, State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, The B2 Building, 89#, Minzhuang Road, Haidian District, Beijing 100093, China.
E-mail: ckwu@iie.ac.cn

1. INTRODUCTION

Nowadays, large amounts of softwares, for example, e-book readers and video players, run in terminals such as PCs and smart phones. However, the terminals may be under intrusion or even owned by malicious users. Cryptographic softwares in the terminals are usually executed in a so-called white-box attack context (WBAC) [1], where a malicious user can establish full control over dynamic execution of the cryptographic implementations, view all the intermediate computing results, and hence easily obtain and illegally distribute the secret key. Not surprisingly, lots of techniques emerge to attack cryptographic implementations in these terminals. Therefore, the traditional black-box cryptographic model, which assumes that the terminals are secure and focuses on communication channel security, is not appropriate in WBAC.

White-box cryptography aims at implementing a cryptographic algorithm by embedding the key in the implementation to protect the key from being extracted in WBAC. It encrypts/decrypts contents in a single terminal and can prevent illegal distribution of the key.

In this paper, we present a secure lookup-table-based white-box implementation for the Chinese block cipher standard SM4 [2]. We show that our implementation can resist a series of white-box attacks. It requires 32.5 MB of memory to store the lookup tables and increases the speed significantly compared with previous implementations.

The rest of this paper is organized as follows. Section 2 presents related work on lookup-table-based white-box implementations and corresponding white-box attacks, and some backgrounds are introduced in Section 3. Section 4 presents our new lookup-table-based white-box SM4 implementation, and Section 5 includes security analyses. Finally, Section 6 concludes the paper.

Some acronyms used in this paper are listed in Table I.

2. RELATED WORK

Most previous white-box implementations are based on lookup tables and aim at protecting advanced encryption standard (AES) [5]. Moreover, our white-box SM4 implementation exploits the Chow *et al*. white-box (AES) techniques [1]. Therefore, this section presents related work on lookup-table-based white-box AES and SM4 implementations and corresponding white-box attacks and further compare our implementation with previous ones.

Copyright © 2015 John Wiley & Sons, Ltd

Table I. Acronyms used in this paper.

Acronym	Description
WBAC	White-box attack context, where the adversary has full control over the dynamic execution of the cryptographic software.
WBAES	White-box AES implementation, a white-box implementation of the advanced encryption standard (AES) block cipher.
WBSM4	White-box SM4 implementation, a white-box implementation of the SM4 block cipher.
BGE	Billet, Gilbert, and Ech-Chatbi, the authors of an efficient white-box attack (the BGE attack) [3].
MGH	Michiels, Gorissen, and Hollmann, the authors of the MGH attack [4].
SLT	Substitution linear-transformation, a cryptographic structure [4].

Chow *et al*. proposed the original white-box AES implementation (WBAES) [1], which embeds round key bytes into its lookup tables. They protected the lookup tables by embedding 4-bit non-linear encodings, 8-bit linear encodings, and 32-bit linear encodings into the tables. Large amounts of distinct constructions of the encodings make it very difficult to locally break these tables and extract the round key bytes.

However, Billet, Gilbert, and Ech-Chatbi proposed an efficient attack (the BGE attack) [3], which combines the tables of a single round, instead of locally breaking the tables. By combining the tables, the 32-bit linear encodings are cancelled out, and hence only the 8-bit encodings are left, which makes it much easier to recover the round key bytes. Tolhuizen [6] and De Mulder *et al*. [7,8] improved the BGE attack. The improved BGE attack can extract the key of the Chow *et al*. WBAES with a work factor of about 2^{22}.

Michiels, Gorissen, and Hollmann extended the Billet *et al*. techniques [3] and presented a universal attack (the MGH attack) [4] to break a class of white-box implementations of substitution linear-transformation (SLT) ciphers implemented with the Chow *et al*. techniques [1].

To resist the BGE attack, Xiao and Lai [9] replaced the 8-bit linear encodings of the Chow *et al*. WBAES with 16-bit ones. Moreover, to obfuscate the relations between the outputs and the inputs of two consecutive rounds, they protected the ShiftRows operation with linear encodings. The Xiao–Lai WBAES also abandoned all the non-linear encodings. De Mulder, Roelse, and Preneel [10] exploited the linearity of the Xiao *et al*. encodings and broke the obfuscation of the ShiftRows operation. Then they recovered the key by detecting linear equivalences between 16-bit S-boxes. The work factor is about 2^{32}.

Karroumi [11] applied the idea of dual AES [12–14] to the Chow *et al*. WBAES to resist the BGE attack. However, as analyzed by Lepoint *et al*. [7,15], there is indeed no difference between Karroumi's WBAES and the Chow *et al*. WBAES. They are both vulnerable to the same attacks. Lepoint *et al*. also presented a new attack on the Chow *et al*. WBAES with a work factor of 2^{22} [7,15], which exploits the collisions of the outputs of a single round.

In 2014, Luo, Lai, and You improved the Xiao–Lai WBAES with non-linear encodings [16] and claimed that their implementation is secure against the Billet *et al*. attack [3] and the De Mulder *et al*. attack [10]. However, although non-linear encodings are applied into their WBAES, the obfuscation of the ShiftRows operation still can be broken by an extension of the De Mulder *et al*. techniques [10], and then the structure of their implementation meets the definition of the white-box implementation of a SLT cipher [4], which is vulnerable to the MGH attack.

SM4, initially called SMS4 and renamed in 2012, is a Chinese block cipher standard issued in 2006 for WLAN products [2]. Its specification has an English description [17]. SM4 is a commercial block cipher standard, so it is very important to protect SM4 in WBAC and prevent malicious users illegally benefiting from distributingits key.

In 2009, Xiao and Lai made the first and so far the only attempt to design a white-box SM4 implementation (WBSM4) [18,19]. They exploited the Chow *et al*. techniques [1] but abandoned all the non-linear encodings and replaced the transcoding tables (e.g., Type III tables of the Chow *et al*. WBAES) with matrices. Because the Chow *et al*. WBAES were badly broken, the Xiao–Lai WBSM4 is also vulnerable. Lin and Lai [20] extended the Billet *et al*. techniques [3] and broke the Xiao–Lai WBSM4.

All the introduced white-box implementations are extensions of the Chow *et al*. techniques [1] and have been broken because the embedded large linear encodings (e.g., the 32-bit linear encodings) are cancelled out by composition of lookup tables, which decreases the difficulty of recovering the encodings from 32-bit level to 8-bit level (or 16-bit level). The white-box SM4 implementation proposed in this paper can protect the 32-bit linear encodings from being cancelled out. As shown in Table II, compared with previous white-box implementations, our WBSM4 has the following properties:

- Security property: All the previous implementations except the Luo-Lai-You WBAES [16] are broken. In fact, as analyzed earlier, although there is no previous work that targets the Luo-Lai-You WBAES, it is indeed vulnerable to the combination of the MGH attack [4] and the De Mulder *et al*. techniques [10]. In contrast, our WBSM4 shows good resistance against existing white-box attacks.
- Speed: Our WBSM4 is faster than all the other white-box implementations. Specially, it is about 9 times as fast as the Xiao–Lai WBSM4.
- Memory: The large memory requirement (32.5 MB) may be a major drawback of our WBSM4. However,

Table II. Comparison between our white-box SM4 and other implementations.

Implementation	Security property	Speed (Figure 2)	Memory (MB)
The Chow *et al.* WBAES	Broken	Medium	About 0.73
Karroumi's WBAES	Broken	Medium	About 0.73
The Xiao–Lai WBAES	Broken	Medium	About 20.02
The Luo-Lai-You WBAES	Not broken	Slow	About 27.78
The Xiao–Lai WBSM4	Broken	Medium	About 0.15
Our WBSM4	Not broken	Fast	32.5

all the KB-class implementations have been broken, while all the currently unbroken implementations are in 10-MB-class [16] and even GB-class [21], which shows that to achieve good security properties 10-MB-class implementations are still acceptable. Moreover, this drawback can be reduced by the rapid development of hardware techniques.

3. BACKGROUND

3.1. Encodings and decodings

Most white-box implementations, including our WBSM4, are based on lookup tables. A lookup table is a function implemented by table lookups. It is constructed with and protected by randomly chosen encodings and decodings, as defined in Definition 1.

Definition 1 (Encoding/decoding)**.** *Given two value sets S_1 and S_2, which have the same size, a bijective mapping $\mathbf{E} : S_1 \ni \mathbf{x} \mapsto \mathbf{y} \in S_2$ is called an encoding. Its inverse mapping is called its decoding. An encoding can cancel out its decoding, and vice versa.*

The encodings and decodings make the construction of the lookup tables more diverse and protect the embedded secret information (e.g., the round key bytes). Therefore, It is very difficult to locally break the lookup tables.

Specially, all the lookup-table-based white-box implementations exploit linear/affine encodings, as defined in Definition 2, where $\oplus$ denotes a bitwise XOR operation.

Definition 2 (Linear/affine encoding)**.** *Given a nonsingular $n \times n$-bit matrix $\mathbf{M}$ and an n-bit vector $\mathbf{b}$, the function $\mathbf{LE}(\mathbf{x}) = \mathbf{M} \cdot \mathbf{x}$ (also denoted as $\mathbf{M}(\mathbf{x})$) is called a linear encoding (linear transformation), and the function $\mathbf{AE}(\mathbf{x}) = \mathbf{M} \cdot \mathbf{x} \oplus \mathbf{b}$ is called an affine encoding (affine transformation).*

These white-box implementations exploit three types of linear/affine encodings for the round transformations: 8-bit ones, 16-bit ones, and 32-bit ones. We regard the 8-bit ones as small linear/affine encodings, the 16-bit ones as medium linear/affine encodings, and the 32-bit ones as large linear/affine encodings. All the white-box implementations are badly broken because the embedded large linear/affine encodings can be cancelled out by compositions of lookup tables. Therefore, to resist existing white-box attacks, it is necessary to protect the large encodings.

3.2. Description of SM4

The SM4 algorithm maps 128-bit inputs to 128-bit outputs with 128-bit keys. There are several equivalent ways to describe it, while the one used in this paper is introduced in this section.

The SM4 block cipher takes $\mathbb{x} = (\mathbb{x}_0, \mathbb{x}_1, \mathbb{x}_2, \mathbb{x}_3)$ as the input and consists of 32 rounds, where $\mathbb{x}_i$ $(0 \leqslant i \leqslant 3)$ is a 32-bit value. Each round computes a 32-bit value as defined later:

$$\begin{aligned} \mathbb{x}_{r+3} &= \mathbb{F}(\mathbb{x}_{r-1}, \mathbb{x}_r, \mathbb{x}_{r+1}, \mathbb{x}_{r+2}, \mathbb{k}_r) \\ &= \mathbb{x}_{r-1} \oplus \mathbb{T}(\mathbb{x}_r \oplus \mathbb{x}_{r+1} \oplus \mathbb{x}_{r+2} \oplus \mathbb{k}_r) \\ &= \mathbb{x}_{r-1} \oplus \mathbb{M} \circ \mathbb{S}(\mathbb{x}_r \oplus \mathbb{x}_{r+1} \oplus \mathbb{x}_{r+2} \oplus \mathbb{k}_r) \end{aligned} \tag{1}$$

where $1 \leqslant r \leqslant 32$, $\mathbb{k}_r$ is the rth 32-bit round key. $\mathbb{S} = S||S||S||S$, where S is an 8-bit S-box, and $||$ denotes a concatenation operation. $\mathbb{M}$ is a nonsingular 32×32-bit matrix defined as

$$\mathbb{M} = \begin{pmatrix} B_1 & B_2 & B_2 & B_3 \\ B_3 & B_1 & B_2 & B_2 \\ B_2 & B_3 & B_1 & B_2 \\ B_2 & B_2 & B_3 & B_1 \end{pmatrix} \tag{2}$$

where B_1, B_2, and B_3 are 8×8-bit blocks as defined later:

$$B_1 = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

$$B_2 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4}$$

$$B_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (5)$$

$\mathbb{y} = (\mathbb{x}_{35}, \mathbb{x}_{34}, \mathbb{x}_{33}, \mathbb{x}_{32})$ is taken as the output.

The encryption and decryption of SM4 has the same structure. The only difference between them is the sequence that the round keys are used: if $\mathbb{k}_r$ is the rth round key of SM4 encryption, it is the $(33 - r)$th round key of SM4 decryption.

4. A NEW WHITE-BOX SM4 IMPLEMENTATION

Lookup tables of existing white-box implementations of standard block ciphers all provide sufficient so-called white-box diversities and white-box ambiguities [1], which makes it very difficult to locally analyze these tables obfuscated with large linear encodings. However, existing attacks [3,4,7,10] analyze compositions of the lookup tables instead, which cancel out the embedded large linear encodings in the tables, decrease the difficulty of recovering the encodings from 32-bit level to 8-bit level (or 16-bit level) and make it easier to extract the round key bytes.

This section presents a new white-box SM4 implementation that prevents the embedded large encodings from being cancelled out by table compositions. The workflow of our white-box scheme is shown in Figure 1. The white-box SM4 table generator takes an SM4 key as the input and makes two types of lookup tables: TD and TR, which are applied to the white-box SM4 encryption/decryption algorithm. $\mathbb{s}_{r,0}$, $\mathbb{s}_{r,1}$, and $\mathbb{x}'_i$ are all 32-bit values. The white-box SM4 algorithm takes the lookup tables and a 128-bit value $(\mathbb{x}'_0, \mathbb{x}'_1, \mathbb{x}'_2, \mathbb{x}'_3)$ as the inputs and outputs a 128-bit value $(\mathbb{x}'_{35}, \mathbb{x}'_{34}, \mathbb{x}'_{33}, \mathbb{x}'_{32})$. It consists of 32 rounds. Each round (the rth round, where $1 \leqslant r \leqslant 32$) takes a 128-bit value $(\mathbb{x}'_{r-1}, \mathbb{x}'_r, \mathbb{x}'_{r+1}, \mathbb{x}'_{r+2})$ as the input and proceeds in the following steps:

- Initialize $\mathbb{s}_{r,0}$, $\mathbb{s}_{r,1}$, and $\mathbb{x}'_{r+3}$ to be zero.
- Take $\mathbb{s}_{r,0}$, $\mathbb{s}_{r,1}$, $\mathbb{x}'_r$, $\mathbb{x}'_{r+1}$, and $\mathbb{x}'_{r+2}$ as the inputs and update the values of $\mathbb{s}_{r,0}$ and $\mathbb{s}_{r,1}$ using 12 TD tables and 12 XOR operations.
- Take $\mathbb{x}'_{r-1}$ and $\mathbb{x}'_{r+3}$ as the inputs and update the value of $\mathbb{x}'_{r+3}$ using 4 TD tables and 4 XOR operations.
- Take $\mathbb{s}_{r,0}$, $\mathbb{s}_{r,1}$, and $\mathbb{x}'_{r+3}$ as the inputs and update the value of $\mathbb{x}'_{r+3}$ using 4 TR tables and 4 XOR operations.
- Output $\mathbb{x}'_{r+3}$.

4.1. The design of the lookup tables

Unlike some previous white-box implementations [1,16], to derive a fast implementation, we abandon all the non-linear encodings as well as the XOR-performing lookup tables. Our implementation consists of two types of lookup tables: TD and TR.

TD is a table type which maps an 8-bit input to a 32-bit output. It is used to decode the previous 32-bit affine encodings and perform a new affine encoding, as defined in Equation (6) ($1 \leqslant r \leqslant 32$, $0 \leqslant j \leqslant 3$):

$$\mathrm{TD}_{r,i,j} = \begin{cases} \mathbb{A}_{r+3,0,j} \circ D_{\mathrm{TD}_{r,i,j}}, & \text{if } i = 0 \\ \mathcal{A}_{r,k,l} \circ D_{\mathrm{TD}_{r,i,j}}, & \text{if } 1 \leqslant i \leqslant 3 \end{cases} \quad (6)$$

where

- $\circ$ denotes the function composition. Given two functions f and g, $f \circ g(x) = f(g(x))$.
- $D_{\mathrm{TD}_{r,i,j}} = \oplus_{\mathbb{L}^{-1}_{r+i-1}(\mathbb{b}_{r+i-1,j})} \circ (\mathbb{L}^{-1}_{r+i-1})_j$ is the input decoding, in which $\oplus_c(x) = x \oplus c$ (where x and c are two values), and $(\mathbb{L}^{-1}_{r+i-1})_j$ is a 32×8-bit strip of $\mathbb{L}^{-1}_{r+i-1} = ((\mathbb{L}^{-1}_{r+i-1})_0, \cdots, (\mathbb{L}^{-1}_{r+i-1})_3)$, which is the inverse of the 32-bit linear encoding, $\mathbb{L}_{r+i-1}$. $\mathbb{b}_{r+i-1,j}$ is a randomly chosen 32-bit value, and $\mathbb{b}_{r+i-1} = \bigoplus_{j=0}^{3} \mathbb{b}_{r+i-1,j}$ is the constant part of the affine encoding $\mathbb{A}_{r+i-1} = \oplus_{\mathbb{b}_{r+i-1}} \circ \mathbb{L}_{r+i-1}$.
- $\mathbb{A}_{r+3,0,j} = \oplus_{\mathbb{b}_{r+3,0,j}} \circ \mathbb{L}_{r+3}$, where $\mathbb{L}_{r+3}$ is a randomly chosen 32-bit linear transformation and $\mathbb{b}_{r+3,0,j}$ is a randomly chosen 32-bit value.
- $\mathcal{A}_{r,k,l} = \|_{m=0}^{3} A_{r,k,l,m} = \|_{m=0}^{3} (\oplus_{b_{r,k,l,m}} \circ L_{r,k,m})$, in which $k = \lfloor j/2 \rfloor$, and $l = (i-1) \times 2 + (j \bmod 2)$. $L_{r,k,m}$ is a randomly chosen 8-bit linear encoding, and $b_{r,k,l,m}$ is a randomly chosen 8-bit value. $b_{r,k,m} = \bigoplus_{l=0}^{5} b_{r,k,l,m}$ is the constant part of the 8-bit affine encoding $A_{r,k,m} = \oplus_{b_{r,k,m}} \circ L_{r,k,m}$.

TR is a table type, which maps a 16-bit input to a 32-bit output. It is used to perform the $\mathbb{T}$ transformation defined in Equation (1), as defined in Equation (7) ($1 \leqslant r \leqslant 32$, $0 \leqslant j \leqslant 3$):

$$\mathrm{TR}_{r,j} = \mathbb{A}_{r+3,1,j} \circ \mathbb{M}_j \circ S \circ \oplus_{\mathbb{k}_{r,j}} \circ D_{\mathrm{TR}_{r,j}} \quad (7)$$

where $D_{\mathrm{TR}_{r,j}} = (L^{-1}_{r,0,j}, L^{-1}_{r,1,j}) \circ \oplus_{(b_{r,0,j}, b_{r,1,j})}$, $\mathbb{k}_{r,j}$ is the jth byte of $\mathbb{k}_r$, $\mathbb{M}_j$ is a 32×8-bit strip of $\mathbb{M} = (\mathbb{M}_0, \cdots, \mathbb{M}_3)$ ($\mathbb{M}$ is also used to denote its corresponding linear transformation here), and $\mathbb{A}_{r+3,1,j} = \oplus_{\mathbb{b}_{r+3,1,j}} \circ \mathbb{L}_{r+3}$. $\mathbb{b}_{r+3} = \bigoplus_{j=0}^{3} (\mathbb{b}_{r+3,0,j} \oplus \mathbb{b}_{r+3,1,j})$ is the constant part of the affine encoding $\mathbb{A}_{r+3} = \oplus_{\mathbb{b}_{r+3}} \circ \mathbb{L}_{r+3}$.

4.2. The encryption/decryption process

The encryption and decryption process of our white-box SM4 implementation are the same, using lookup tables derived from encryption round keys and decryption round keys, respectively. The transformation process is shown in Algorithm 1.

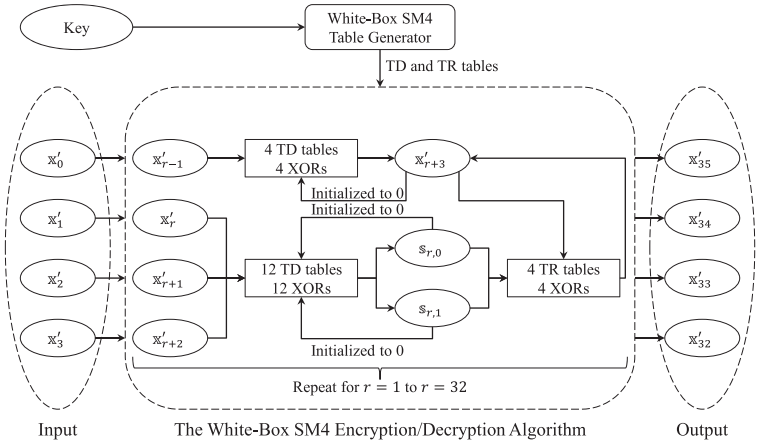

Figure 1. The workflow of our scheme.

Algorithm 1 The transformation process of our WBSM4

Input:
$\mathbb{x}' = (\mathbb{x}'_0, \mathbb{x}'_1, \mathbb{x}'_2, \mathbb{x}'_3)$, the encoded input;
Output:
$\mathbb{y}' = (\mathbb{x}'_{35}, \mathbb{x}'_{34}, \mathbb{x}'_{33}, \mathbb{x}'_{32})$, the encoded output;
for $r \leftarrow 1 \ldots 32$ **do**
  $\mathbb{x}'_{r+3} \leftarrow 0$;
  $\mathbb{s}_{r,0} \leftarrow 0$;
  $\mathbb{s}_{r,1} \leftarrow 0$;
  for $i \leftarrow 1 \ldots 3$ **do**
    $\mathbb{s}_{r,0} \leftarrow \mathbb{s}_{r,0} \oplus \mathrm{TD}_{r,i,0}(\mathbb{x}'_{r+i-1,0})$;
    $\mathbb{s}_{r,0} \leftarrow \mathbb{s}_{r,0} \oplus \mathrm{TD}_{r,i,1}(\mathbb{x}'_{r+i-1,1})$;
    $\mathbb{s}_{r,1} \leftarrow \mathbb{s}_{r,1} \oplus \mathrm{TD}_{r,i,2}(\mathbb{x}'_{r+i-1,2})$;
    $\mathbb{s}_{r,1} \leftarrow \mathbb{s}_{r,1} \oplus \mathrm{TD}_{r,i,3}(\mathbb{x}'_{r+i-1,3})$;
  end for
  for $j \leftarrow 0 \ldots 3$ **do**
    $\mathbb{x}'_{r+3} \leftarrow \mathbb{x}'_{r+3} \oplus \mathrm{TR}_{r,j}((\mathbb{s}_{r,0,j}, \mathbb{s}_{r,1,j}))$;
    $\mathbb{x}'_{r+3} \leftarrow \mathbb{x}'_{r+3} \oplus \mathrm{TD}_{r,0,j}(\mathbb{x}'_{r-1,j})$;
  end for
end for
return $\mathbb{y}'$;

Like other white-box implementations, our implementation also maps encoded inputs $\mathbb{x}' = (\mathbb{x}'_0, \mathbb{x}'_1, \mathbb{x}'_2, \mathbb{x}'_3)$ to encoded outputs $\mathbb{y}' = (\mathbb{x}'_{35}, \mathbb{x}'_{34}, \mathbb{x}'_{33}, \mathbb{x}'_{32})$. In Algorithm 1, $\mathbb{x}'_{i'} = \mathbb{A}_{i'}(\mathbb{x}_{i'})$ $(0 \leqslant i' \leqslant 35)$. Each round computes a 32-bit value $\mathbb{x}'_{r+3}$. $\mathbb{s}_{r,0}$ and $\mathbb{s}_{r,1}$ are both intermediate 32-bit values, and $\mathbb{s}_{r,0,j}$ and $\mathbb{s}_{r,1,j}$ are the jth byte of $\mathbb{s}_{r,0}$ and $\mathbb{s}_{r,1}$ respectively.

$\mathbb{s}_{r,0}$ and $\mathbb{s}_{r,1}$ are protected by different concatenated 8-bit affine encodings and different 32×16-bit strips of the decodings of the previous 32-bit affine encodings. In the internal of TR_r tables, $\mathbb{s}_{r,0}$ and $\mathbb{s}_{r,1}$ are decoded and XORed, and then the previous 32-bit affine encodings are cancelled out. However, the outputs of TR_r tables are protected by new 32-bit affine encodings. That is to say, all the intermediate results are protected by 32-bit affine encodings/decodings. Therefore, the 32-bit affine encodings cannot be cancelled out by table compositions, and the adversary cannot obtain intermediate results that are protected only by 8-bit encodings. So the difficulty of recovering the encodings cannot be decreased from 32-bit level to 8-bit level.

4.3. Size and performance

4.3.1. Performance metrics.

Chow *et al*. [1] proposed two metrics to evaluate the performance of their WBAES: number of table lookups and number of XOR operations that XORs the outputs of the lookup tables. Besides these two metrics, Xiao and Lai [9] also exploited number of matrix multiplications to evaluate the performance of their implementation. These metrics can help to evaluate the performance of a lookup-table-based white-box implementation but are not sufficient.

In fact, once the inputs are given, table lookups are very fast. However, before table lookups, the inputs need to be obtained, the process of which is usually very time-consuming. For example, Type IV tables of the Chow *et al*. WBAES map two 4-bit inputs to a 4-bit output. To use a Type IV table, two 4-bit values need to be got from two previous computing results (which may be 32-bit values), which requires many additional operations such as bitwise AND, bitwise XOR, and bitwise shift. Therefore, the number of inputs of the lookup tables, which has a large influence over the performance, is proposed as a measure in this paper.

Moreover, there are a type of lookup tables, named bitwise tables as defined in Definition 3, which require many operations to store the outputs after lookups of the tables.

Definition 3 (Bitwise table)**.** *A lookup table is called a bitwise table if the byte count of its output is not an integer.*

DOI: 10.1002/sec

For example, Type IV tables of the Chow *et al*. WBAES, which have 4-bit (0.5-byte) outputs, are bitwise tables. In byte-operation-based software implementations, lookups of such tables usually require many additional operations to store their outputs, which are also very time-consuming. Therefore, to reasonably evaluate the performance of our WBSM4, we also propose another metric: the number of outputs of bitwise tables.

4.3.2. Comparison of size and performance.

Our implementation exploits 16 TD and 4 TR tables in each round, and hence the total number of table lookups of our implementation is $32 \times (16 + 4) = 640$. Each TD table maps an 8-bit input to a 32-bit output, and each TR table maps two 8-bit inputs to a 32-bit output. So the total number of inputs of our lookup tables is $32 \times (16 + 4 \times 2) = 768$. Moreover, in each round, it needs $3 \times 4 + 4 \times 2 = 20$ XOR operations to support TD and TR tables, so the total number of XOR operations is $32 \times 20 = 640$. The size of lookup tables of our implementation is $32 \times (16 \times 2^8 \times 4 \times 2^{-20} + 4 \times 2^{16} \times 4 \times 2^{-20}) = 32.5$ (MB).

The Xiao–Lai WBSM4 requires $32 \times 4 = 128$ table lookups, 160 32-bit affine transformations, and $32 \times (2 + 4) = 192$ XOR operations to support them. It needs 148.625 KB to store its lookup tables and affine transformations.

Comparison of size and performance between our WBSM4 and other white-box implementations is shown in Table III. The number of table lookups, number of inputs of the tables, and number of outputs of bitwise tables of the Chow *et al*. WBAES, Karroumi's WBAES, and the Luo-Lai-You WBAES are all very large, which make their speeds very slow. XOR operations are very fast, but matrix multiplications are very time-consuming. Therefore, our WBSM4 is faster than the Xiao–Lai WBAES and the Xiao–Lai WBSM4.

Compared with the Xiao–Lai WBSM4, our implementation requires much more memories to store the lookup tables, which may be a major drawback of our WBSM4. However, all the KB-class white-box implementations [1,9,11,18] were badly broken, and currently unbroken white-box implementations are all in 10-MB-class [16] and even GB-class [21]. Therefore, the large memory requirement of our implementation is acceptable for applications which aim at achieving good white-box security properties, while it is very important to choose platforms which can provide sufficient memories to apply our implementation. On the other hand, rapid development of hardware techniques can reduce this drawback.

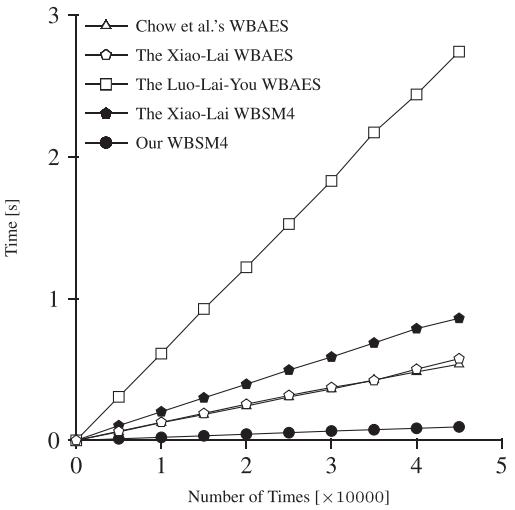

Figure 2. Comparison of speeds shown in a graph: the x axis represents how many times a block is encrypted, and the y axis represents the time used to encrypt the block for x times.

The Xiao–Lai WBSM4 is the only previous white-box SM4 implementation. To compare the actual speeds of the Xiao–Lai WBSM4 and our implementation, we implemented them with C++ on a PC, which has an Intel Core i7 processor and 4GB of RAM, and exploited them to encrypt a given 16-byte encoded plaintext for several times. The experimental results are shown in Figure 2, in which the x axis represents the number of times a block (16-byte input) is encrypted, and the y axis represents the time used to encrypt the block for x times. For example, the figure shows that it takes about 0.5 second for the Xiao–Lai WBSM4 to encrypt a given input for 25 000 times. As shown in Figure 2, the actual speed of our implementation is about nine times as fast as that of the Xiao–Lai WBSM4.

In addition, the performance of existing white-box AES implementations are also shown in Figure 2. Karroumi's WBAES is indeed the same as the Chow *et al*. WBAES, so we only show the latter's performance curve in the figure. The Chow *et al*. WBAES exploits many table lookups,

Table III. Comparison of size and performance.

Implementation	NTL	NIT	NOBT	NXOR	NMM	Memory (MB)
The Chow *et al*. WBAES	3104	5696	2688	0	0	About 0.73
Karroumi's WBAES	3104	5696	2688	0	0	About 0.73
The Xiao–Lai WBAES	80	80	0	40	11	About 20.02
The Luo-Lai-You WBAES	7776	18496	7360	0	0	About 27.78
The Xiao–Lai WBSM4	128	128	0	192	160	About 0.15
Our WBSM4	640	768	0	640	0	32.5

NTL, Number of Table Lookups; NIT, Number of Inputs of the lookup Tables; NOBT, Number of Outputs of Bitwise Tables; NXOR, Number of XORs; NMM, Number of Matrix Multiplications.

while the Xiao–Lai WBAES exploits many matrix multiplications. So their performance curves are approximately the same. The Xiao–Lai WBAES exploits 128 × 128-bit matrix multiplications, while the Xiao–Lai WBSM4 exploits 32 × 32-bit ones. Therefore, although the number of matrix multiplications of the latter is about 14.5 times as large as that of the former, the actual speed of the former is only about 1.5 times as fast as that of the latter. The Luo-Lai-You WBAES has very large number of table lookups as well as number of inputs of the tables and number of outputs of bitwise tables, so it is very slow as shown in the figure.

5. SECURITY ANALYSIS

This section includes security comments on our white-box SM4 implementation. First, a table that compares security property of our WBSM4 with other white-box implementations is shown in Table IV, where Y means that the implementation can resist the attack, and N means that the implementation cannot resist the attack. Empty cells mean that no previous work shows whether the implementation can resist the attack or not. For example, the Luo-Lai-You WBAES can resist the BGE attack and the De Mulder *et al*. attack, but no evidence shows that it can resist other attacks. Indeed, our analyses in Section 2 show the vulnerability of the Luo-Lai-You WBAES.

Table IV shows good resistance of our implementation against existing white-box attacks. In the following, we will present detailed analyses on security properties of our WBSM4.

5.1. Preliminary security comments

Chow *et al*. proposed two metrics to evaluate security properties of lookup tables: white-box diversity and white-box ambiguity [1]. White-box diversity measures the number of distinct constructions of a table type, while white-box ambiguity measures the number of distinct constructions that produce exactly the same table.

The number of distinct constructions of a 32×8-bit matrix is $2^{32\times 8} = 2^{256}$; the number of distinct constructions of a nonsingular 8×8-bit matrix is $\sum_{i=0}^{7}(2^8 - 2^i) \approx 2^{62.2}$, and the number of distinct constructions of a nonsingular 32×32-bit matrix is $\sum_{i=0}^{31}(2^{32}-2^i) \approx 2^{1022.2}$. There are 2^8 distinct 8-bit values and 2^{32} distinct 32-bit values.

Therefore, the white-box diversity of a $\mathrm{TD}_{r,0,j}$ table is about $2^{256} \times 2^{32} \times 2^{1022.2} \times 2^{32} = 2^{1342.2}$ and that of a $\mathrm{TD}_{r,i,j}$ $(1 \leqslant i \leqslant 3)$ table is about $2^{256} \times 2^{32} \times (2^{62.2} \times 2^8)^4 = 2^{568.8}$. The white-box diversity of a TR table is about $(2^8 \times 2^{62.2})^2 \times 2^8 \times 2^{1022.2} \times 2^{32} = 2^{1202.6}$.

Given a $\mathrm{TD}_{r,0,j}$ table, the 32×8-bit matrix and the corresponding 32-bit value determine the 32-bit affine encoding. Therefore, the white-box ambiguity of a $\mathrm{TD}_{r,0,j}$ table is $2^{256} \times 2^{32} = 2^{288}$.

Given a $\mathrm{TD}_{r,i,j}$ $(1 \leqslant i \leqslant 3)$ table, the four 8-bit affine encodings determine the 32×8-bit matrix and the corresponding 32-bit value. Therefore, the white-box ambiguity of a $\mathrm{TD}_{r,i,j}$ table is about $(2^{62.2} \times 2^8)^4 = 2^{280.8}$.

Given a TR table, the two 8-bit affine decodings and the round key byte together determine the 32-bit affine encoding. Therefore, the white-box ambiguity of a TR table is about $(2^8 \times 2^{62.2})^2 \times 2^8 = 2^{148.4}$.

5.2. Compositions of lookup tables

Large white-box diversities and white-box ambiguities of lookup tables make it very difficult to locally analyze these tables. Therefore, existing attacks all analyze compositions of lookup tables instead. For our implementation, there are two types of table compositions, which will be introduced later.

5.2.1. The first type of table compositions.

Existing white-box attacks, including the Lin–Lai attack, mainly analyze the first type of table compositions, which combines $\mathrm{TR}_{r,j}$, $\mathrm{TD}_{r,0,j}$, and $\mathrm{TD}_{r+3,1,j}$ $(\forall r \in [1, 29])$ tables, as shown in Figure 3, where $\mathbb{v}_{r+3,1,0}$ and $\mathbb{v}_{r+3,1,1}$ are both 32-bit values.

In Figure 3, the input of $\mathbb{A}_{r+3}$ is a 32-bit value defined as

$$\mathbb{s}'_{r+3} = \mathbb{A}_{r-1}^{-1}\left(\mathbb{x}'_{r-1}\right) \oplus \mathbb{M} \circ \mathbb{S}\left(\mathbb{k}_r \oplus \Big\|_{j=0}^{3}\left(A_{r,0,j}^{-1}\left(\mathbb{s}_{r,0,j}\right) \oplus A_{r,1,j}^{-1}\left(\mathbb{s}_{r,1,j}\right)\right)\right) \tag{8}$$

where $\mathbb{s}_{r,0,j}$ and $\mathbb{s}_{r,1,j}$ are the jth byte of $\mathbb{s}_{r,0}$ and $\mathbb{s}_{r,1}$ of Algorithm 1, respectively. The output of $\mathbb{A}_{r+3}$ is $\mathbb{x}'_{r+3} = \mathbb{A}_{r+3}(\mathbb{s}'_{r+3}) = \mathbb{L}_{r+3}(\mathbb{s}'_{r+3}) \oplus \mathbb{b}_{r+3}$. From Equation (6), the expressions of $\mathbb{v}_{r+3,1,0}$ and $\mathbb{v}_{r+3,1,1}$ are derived as follows:

Table IV. Comparison of resistance against existing attacks.

Implementation	The BGE attack [3]	The MGH attack [4]	The De Mulder *et al.* attack [10]	The Lepoint–Rivain attack [7,15]	The Lin–Lai attack [20]
The Chow *et al.* WBAES	N	N		N	
Karroumi's WBAES	N	N		N	
The Xiao–Lai WBAES	Y		N		
The Luo-Lai-You WBAES	Y		Y		
The Xiao–Lai WBSM4					N
Our WBSM4	Y	Y	Y	Y	Y

DOI: 10.1002/sec

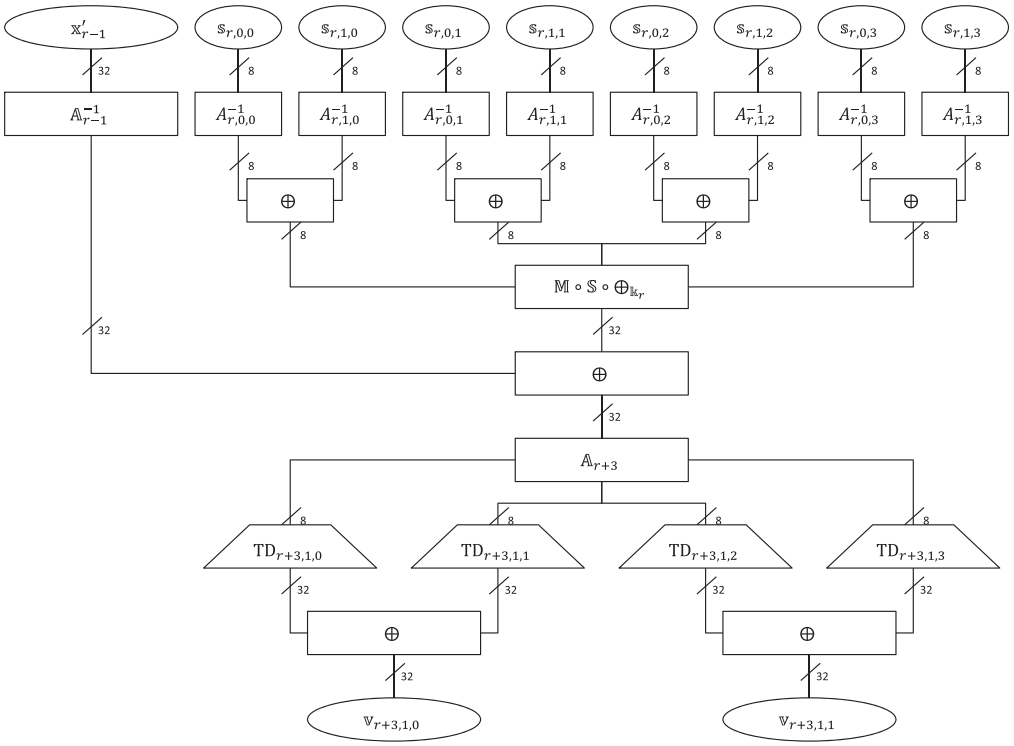

Figure 3. The first type of table compositions.

$$\begin{aligned}
\mathbb{v}_{r+3,1,0} =& \mathcal{A}_{r+3,0,0}\left(\left(\mathbb{L}_{r+3}^{-1}\right)_0\left(\mathbb{x}'_{r+3,0}\right) \oplus \mathbb{L}_{r+3}^{-1}\left(\mathbb{b}_{r+3,0}\right)\right) \\
&\oplus \mathcal{A}_{r+3,0,1}\left(\left(\mathbb{L}_{r+3}^{-1}\right)_1\left(\mathbb{x}'_{r+3,1}\right) \oplus \mathbb{L}_{r+3}^{-1}\left(\mathbb{b}_{r+3,1}\right)\right) \\
\mathbb{v}_{r+3,1,1} =& \mathcal{A}_{r+3,1,0}\left(\left(\mathbb{L}_{r+3}^{-1}\right)_2\left(\mathbb{x}'_{r+3,2}\right) \oplus \mathbb{L}_{r+3}^{-1}\left(\mathbb{b}_{r+3,2}\right)\right) \\
&\oplus \mathcal{A}_{r+3,1,1}\left(\left(\mathbb{L}_{r+3}^{-1}\right)_3\left(\mathbb{x}'_{r+3,3}\right) \oplus \mathbb{L}_{r+3}^{-1}\left(\mathbb{b}_{r+3,3}\right)\right)
\end{aligned} \tag{9}$$

If $\mathcal{A}_{r+3,0,j}$ and $\mathcal{A}_{r+3,1,j \bmod 2}$ have the same linear part, $\mathbb{v}_{r+3,1,0}$ and $\mathbb{v}_{r+3,1,1}$ can be XORed, and then the previous 32-bit affine encoding $\mathbb{A}_{r+3}$ can be cancelled out. However, in fact, the linear parts of $\mathcal{A}_{r+3,0,j}$ and $\mathcal{A}_{r+3,1,j \bmod 2}$ are different, which prevents $\mathbb{A}_{r+3}$ from being cancelled out by XORing $\mathbb{v}_{r+3,1,0}$ and $\mathbb{v}_{r+3,1,1}$. That is to say, this type of table compositions cannot cancel out the 32-bit affine encodings of our WBSM4.

5.2.2. The second type of table compositions.

As analyzed in Section 5.2.1, most existing white-box attacks analyze the first type of table compositions to decrease the difficulty of recovering the linear encodings from 32-bit level to 8-bit level. However, this type of table compositions cannot cancel out our 32-bit affine encodings.

The second type of table compositions, which combine all the tables of a single round (the *r*th round), may be an alternative choice, as shown in Figure 4. All the concatenated 8-bit affine encodings are cancelled out, and the 32-bit affine encodings are required to be broken to extract the round key $\mathbb{k}_r$.

In the remainder of Section 5, the resistance of our implementation against existing attacks will be analyzed by analyzing these two types of table compositions.

5.3. Resistance against the Biryukov–Shamir Attack

Biryukov and Shamir proposed a structural attack on an "SASAS" structure (S-boxes Affine-transformation S-boxes Affine-transformation S-boxes), which starts by recovering the two external S-box layers, followed by attacking the remaining "ASA" structure [22].

Although table compositions of our implementation coincidentally have an "ASA" structure, Biryukov and Shamir pointed out that their attack can only "find an equivalent representation" of all the cryptographic components in the "SASAS" scheme, which enables the adversary to encrypt/decrypt arbitrary inputs, without finding "the original definition of these elements" [22]. The sufficient white-box diversities and white-box ambiguities of our lookup tables prevent the Biryukov–Shamir attack from recovering the original construction and further extracting the key.

White-box cryptography exploits the lookup tables to encrypt/decrypt contents in a single terminal and aims at protecting the embedded key from being extracted in this terminal and from being illegally distributed to other

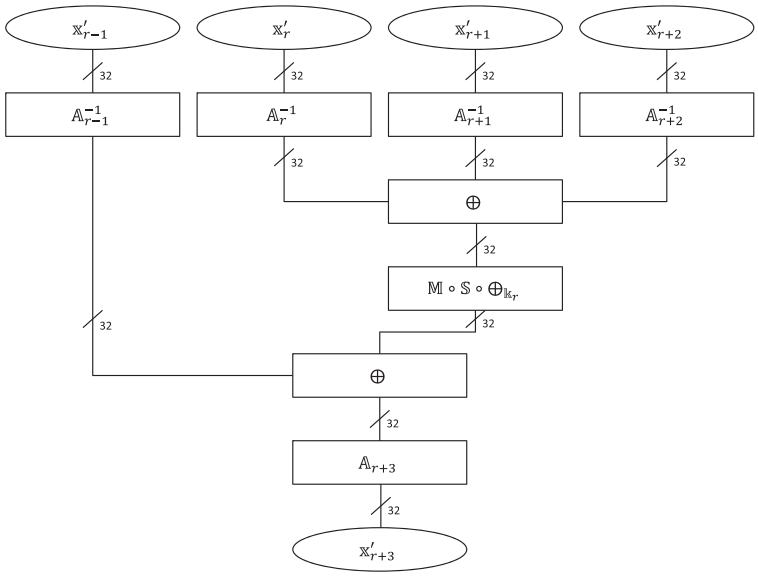

Figure 4. The second type of table compositions.

terminals. The Biryukov–Shamir attack cannot recover the key, so our white-box SM4 implementation are secure against it from the perspective of white-box cryptography.

5.4. Resistance against the Billet, Gilbert, and Ech-Chatbi attack

The BGE attack [3] recovers the round key by detecting affine equivalences, and the Lin–Lai attack [20] also relies on the Billet *et al*. techniques [3] to recover the linear encodings. Therefore, in this section, we take the BGE attack as an example to analyze the resistance of our WBSM4 against this type of attacks.

For the first type of table compositions of our implementation shown in Figure 3, the 32-bit affine encoding is not cancelled out, and the 8×8-bit blocks of the 32×8-bit matrix of $\mathrm{TD}_{r+3,1,j}$ may be singular (the rank of the 32×8-bit matrix is only 8), so the affine equivalence equation cannot be constructed, and the attack is not applicable.

For the second type of table compositions shown in Figure 4, denote the round function as $\mathbb{x}'_{r+3} = \mathbb{F}'(\mathbb{x}'_{r-1}, \mathbb{x}'_r, \mathbb{x}'_{r+1}, \mathbb{x}'_{r+2})$, and an affine equivalence equation can be constructed as

$$\psi : \mathbb{x}'_r \quad \mapsto \mathbb{S}^{-1} \circ \mathbb{M}^{-1} \circ \oplus_{\mathbb{A}^{-1}_{r-1}(0)} \circ \mathbb{A}^{-1}_{r+3} \circ \mathbb{F}'(0, \mathbb{x}'_r, 0, 0) \tag{10}$$

It needs to find the pair $(\mathbb{A}^{-1}_{r+3}, \mathbb{A}^{-1}_{r-1}(0))$ such that ψ is an affine mapping. The work factor is $\sigma \times 2^{32}$, where σ is the number of distinct constructions of the 32-bit invertible affine decoding $\mathbb{A}^{-1}_{r+3}$. So the work factor is about $2^{1022.2} \times 2^{32} = 2^{1054.2}$.

5.5. Resistance against the Michiels, Gorissen, and Hollmann attack

The MGH attack [4] and the De Mulder *et al*. attack [10] are based on the Biryukov *et al*. affine and linear equivalence algorithm [13], respectively. Our implementation is obfuscated with affine encodings, so the MGH attack is more applicable to our implementation. Therefore, in this section, we take the MGH attack as an example to analyze the resistance of our WBSM4 against this type of attacks.

As is discussed in Section 5.4, the 32-bit affine encoding is not cancelled out by the first type of table compositions shown in Figure 3, and the 8×8-bit blocks of the 32×8-bit matrix of $\mathrm{TD}_{r+3,1,j}$ may be not invertible (the rank of the 32×8-bit matrix is only 8). Therefore, the Biryukov *et al*. affine equivalence algorithm as well as the MGH attack are not applicable to this type of table compositions.

For the second type of table compositions shown in Figure 4, we have

$$\mathbb{S}' = \mathbb{A}'_{r+3} \circ \mathbb{S} \circ \mathbb{A}'_r \tag{11}$$

where $\mathbb{A}'_{r+3} = \mathbb{A}_{r+3} \circ \oplus_{\mathbb{A}^{-1}_{r-1}(0)} \circ \mathbb{M}$, and $\mathbb{A}'_r = \oplus_{\mathbb{k}_r} \circ \oplus_{\mathbb{A}^{-1}_{r+1}(0)} \circ \oplus_{\mathbb{A}^{-1}_{r+2}(0)} \circ \mathbb{A}^{-1}_r$. It needs to detect affine equivalences between $\mathbb{S}$ and $\mathbb{S}'$ with the Biryukov *et al*. techniques [13], the work factor of which is about $32^3 \times 2^{2\times 32} = 2^{79}$. It needs to recover four round keys to extract the key of our WBSM4 by inverting the key expansion process, which requires at least four affine equivalence problems to be solved. The work factor is at least $4 \times 2^{79} = 2^{81}$.

DOI: 10.1002/sec

5.6. Resistance against the Lepoint–Rivain attack

The Lepoint–Rivain attack [7][15] exploits collisions of the outputs to find the one-to-one relations between the inputs.

For the first type of table compositions shown in Figure 3, denote the composition function that computes $\mathbb{v}_{r+3,1,0}$ as $\mathbb{F}''_0(\mathbb{s}_{r,0,0}, \mathbb{s}_{r,1,0}, \ldots, \mathbb{s}_{r,0,3}, \mathbb{s}_{r,1,3}, \mathbb{x}'_{r-1})$. The Lepoint–Rivain attack needs to detect collisions defined as

$$\mathbb{F}''_0(\mathbb{s}_{r,0,0}, 0, \cdots, 0) = \mathbb{F}''_0(0, 0, \mathbb{s}_{r,0,1}, 0, \cdots, 0) \qquad (12)$$

However, the rank of the 32×32-bit matrix $\mathbb{L}''_{r+3,0} = \mathcal{L}(\mathbb{L}^{-1}_{r+3}) \cdot \mathcal{T}(\mathbb{L}_{r+3})$ is only 16, where $\mathcal{L}(\mathbb{L}^{-1}_{r+3})$ is the left 32×16-bit strip of $\mathbb{L}^{-1}_{r+3}$, and $\mathcal{T}(\mathbb{L}_{r+3})$ is the top 16×32-bit strip of $\mathbb{L}_{r+3}$. So the one-to-one relations between $\mathbb{s}_{r,0,0}$ and $\mathbb{s}_{r,0,1}$ cannot be detected from at most 256 collisions of the outputs, which means that the Lepoint–Rivain attack does not work for this type of table compositions.

For the second type of table compositions shown in Figure 4, The Lepoint–Rivain attack needs to detect collisions defined as

$$\mathbb{F}'(0, \mathbb{x}'_r, 0, 0) = \mathbb{F}'(0, 0, \mathbb{x}'_{r+1}, 0) \qquad (13)$$

which can be rewritten as $\mathbb{L}^{-1}_r(\mathbb{x}'_r) = \mathbb{L}^{-1}_{r+1}(\mathbb{x}'_{r+1})$. So the relation between $\mathbb{L}^{-1}_r$ and $\mathbb{L}^{-1}_{r+1}$ can be recovered, and that between $\mathbb{L}^{-1}_r$ and $\mathbb{L}^{-1}_{r+2}$ can be recovered using similar methods. Further, detection of affine equivalences between $\mathbb{F}'$ and $\mathbb{S}$ is required, the time complexity of which is very large, as discussed in Section 5.4 and Section 5.5.

6. CONCLUSION

This paper presents a lookup-table-based white-box SM4 implementation, which can protect the embedded 32-bit affine encodings from being cancelled out by composition of lookup tables, and hence can resist a series of white-box attacks such as the BGE, the MGH, De Mulder *et al*., the Lepoint–Rivain, and the Lin–Lai attack. Our implementation requires 32.5 MB of memory to store the lookup tables and increases the speed significantly compared with previous implementations. Specially, it is about 9 times as fast as the Xiao–Lai WBSM4.

The large memory requirement may be a major drawback of our WBSM4. However, the order of magnitude of our memory requirement is the same as or much smaller than other currently unbroken white-box implementations. To achieve good white-box security properties, the large memory requirement is still acceptable for many platforms, which have enough memories. Moreover, rapid development of hardware techniques can minimize this drawback.

Two performance metrics proposed in this paper, the number of inputs of the lookup tables and the number of outputs of bitwise tables, can help designers to strike a balance between good white-box security properties and good performance.

Acknowledgements

This work was supported by National High-Tech R&D Program of China (863 Program) (Grant no. 2013AA014002) and Strategic Priority Research Program of the Chinese Academy of Sciences (Grant no. XDA06010701) The authors thank all the referees for their valuable comments..

REFERENCES

1. Chow S, Eisen P, Johnson H, Van Oorschot PC. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, vol. 2595, Nyberg K, Heys H (eds), Lecture Notes in Computer Science. Springer: Berlin, 2003; 250–270.
2. Office of the State Commercial Cryptography Administration. *The SMS4 Cryptographic Algorithm for WLAN Products*, 2006. http://www.oscca.gov.cn/UpFile/20062101642319799O.pdf. (in Chinese).
3. Billet O, Gilbert H, Ech-Chatbi C. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, vol. 3357, Handschuh H, Hasan MA (eds), Lecture Notes in Computer Science. Springer: Berlin, 2005; 227–240.
4. Michiels W, Gorissen P, Hollmann HDL. Cryptanalysis of a generic class of white-box implementations. In *Selected Areas in Cryptography*, vol. 5381, Avanzi RM, Keliher L, Sica F (eds), Lecture Notes in Computer Science. Springer: Berlin, 2009; 414–428.
5. Daemen J, Rijmen V. *The design of Rijndael: AES — the advanced encryption standard*, Information Security and Cryptography. Springer: Berlin, 2002.
6. Tolhuizen Ludo. Improved cryptanalysis of an AES implementation, *Proceedings of the 33rd wic symposium on information theory in the benelux, boekelo, the netherlands, may 24–25, 2012*, WIC (Werkgemeenschap voor Inform.- en Communicatietheorie), 2012.
7. Lepoint T, Rivain M, De Mulder Y, Roelse P, Preneel B. Two attacks on a white-box AES implementation. In *Selected Areas in Cryptography – SAC 2013*, vol. 8282, Lange T, Lauter K, Lisoněk P (eds), Lecture Notes in Computer Science. Springer: Berlin, 2014; 265–285.
8. De Mulder Y, Roelse P, Preneel B. Revisiting the BGE attack on a white-box AES implementation. *IACR Cryptology ePrint Archive* 2013; **2013**: 450.

9. Xiao Y, Lai X. A secure implementation of white-box AES, *2nd international conference on computer science and its applications (CSA '09)*. IEEE, 2009; 1–6.
10. De Mulder Y, Roelse P, Preneel B. Cryptanalysis of the Xiao–Lai white-box AES implementation. In *Selected areas in cryptography*, vol. 7707, Knudsen LR, Wu H (eds), Lecture Notes in Computer Science. Springer: Berlin, 2013; 34–49.
11. Karroumi M. Protecting white-box AES with dual ciphers. In *Information Security and Cryptology - ICISC 2010*, vol. 6829, Rhee KH, Nyang D (eds), Lecture Notes in Computer Science. Springer: Berlin, 2011; 278–291.
12. Barkan E, Biham E. In how many ways can you write Rijndael? In *Advances in Cryptology - ASIACRYPT 2002*, vol. 2501, Zheng Y (ed), Lecture Notes in Computer Science. Springer: Berlin, 2002; 160–175.
13. Biryukov A, De Cannière C, Braeken A, Preneel B. A toolbox for cryptanalysis: linear and affine equivalence algorithms. In *Advances in Cryptology - EUROCRYPT 2003*, vol. 2656, Biham E (ed), Lecture Notes in Computer Science. Springer: Berlin, 2003; 33–50.
14. Raddum H. More dual Rijndaels. In *Advanced Encryption Standard — AES*, vol. 3373, Dobbertin Hans, Rijmen Vincent, Sowa Aleksandra (eds), Lecture Notes in Computer Science. Springer, 2005; 33–50.
15. Lepoint T, Rivain M. Another nail in the coffin of white-box AES implementations. *IACR Cryptology ePrint Archive* 2013; **2013**: 455.
16. Luo R, Lai X, You R. A new attempt of white-box AES implementation, *Security, pattern analysis, and cybernetics (SPAC), 2014 international conference on*. IEEE, 2014; 423–429.
17. Diffie W, Ledin G. SMS4 encryption algorithm for wireless networks. *IACR Cryptology ePrint Archive* 2008; **2008**: 329.
18. Xiao Y, Lai X. White-box cryptography and implementations of SMS4, *Proceedings of the 2009 CACR annual meeting*. Science Press, Beijing; 24–34. (in Chinese with English abstract).
19. Xiao Y. White-box cryptography and implementations of AES and SMS4. *Master's Thesis*, Shanghai Jiao Tong University, 2010. (in Chinese with English abstract).
20. Lin T, Lai X. Efficient attack to white-box SMS4 implementation. *Journal of Software* 2013; **24**(9): 2238–2249, (in Chinese with English abstract).
21. Biryukov A, Bouillaguet C, Khovratovich D. Cryptographic schemes based on the ASASA structure: black-box, white-box, and Public-Key (Extended Abstract). In *Advances in Cryptology - ASIACRYPT 2014*, vol. 8873, Sarkar P, Iwata T (eds), Lecture Notes in Computer Science. Springer: Berlin, 2014; 63–84.
22. Biryukov A, Shamir A. Structural cryptanalysis of SASAS. In *Advances in Cryptology - EUROCRYPT 2001*, vol. 2045, Pfitzmann B (ed), Lecture Notes in Computer Science. Springer: Berlin, 2001; 395–405.

DOI: 10.1002/sec