Cover Page

Submission Name:	CLAE
Submitters:	Dongxi Liu, Surya Nepal, Josef Pieprzyk, Willy Susilo
Corresponding Submitter:	Dongxi Liu
	dongxi.liu@data61.csiro.au, 61-2-93724152
	Commonwealth Scientific and Industrial Research Organisation
	Cnr Vimiera and Pembroke Roads
	Marsfield, NSW 2122, Australia

CLAE: a Lightweight AEAD Scheme of Resisting Side Channel Attacks

Dongxi Liu[1], Surya Nepal[1], Josef Pieprzyk[1], and Willy Susilo[2]

[1] CSIRO, Australia
[2] University of Wollongong, Australia

1 Introduction

CLAE is a lightweight scheme for authenticated encryption with associated data (AEAD). In CLAE, a configurable number of message bytes each is encrypted into a 2-byte ciphertext, which is defined over public nonces, messages, and secret keys, in a style similar to the samples in Learning with Errors (LWE) problem [3]. Such 2-byte ciphertexts look as if they are all randomly sampled (i.e., similar to the decision problem in LWE) and form the basis of CLAE to randomize the ciphertexts.

A 2-byte ciphertext contains 1-byte redundancy; if the ciphertext is changed or induced with faults, the decryption of the message byte will fail. In addition to the configurable number of message bytes, other message bytes each is encrypted into an 1-byte ciphertext, randomized directly or indirectly with the 2-byte ciphertexts and secret keys.

In CLAE, the changes of nonces, messages, and associated data are propagated across the whole ciphertext; one bit change of them causes random changes of ciphertexts. On the other hand, if the ciphertext is tampered with, its decryption algorithm spreads the changes or faults to each 2-byte ciphertext, where changes or faults are detected. Without knowing the secret key, CLAE ensures the ciphertexts or associated data cannot be changed systematically to pass integrity check. In this submission, CLAE is specified at 128-bit security level.

CLAE is inherently resistant to side-channel attacks (e.g., differential power analysis and fault attacks) for the following reasons.

- The bytes in the secret key are accessed nondeterministically during encryption, depending on the values of public nonces, keys, associated data, and messages. For example, two nonces of 1-bit difference can lead to very different access sequences of key bytes. It is hard to derive a deterministic relationship between power traces and key bytes in CLAE. Thus, the power traces cannot be used meaningfully to recover secret key bytes.
- A secret intermediate ciphertext byte is directly affected by only up to 4 bits of an adversary-controllable value, which can be messages, public nonces, and associated data. Thus, even with the same nonce, when the adversary launches attacks by varying messages or associated data, a particular byte in secret intermediate values can have 4 bits not affected, thus not detectable via differences of power traces and can be regarded as errors in LWE samples.
- Due to the redundancy in ciphertexts, CLAE can resist fault attacks, similar to other authentication schemes [1]. Moreover, in CLAE, faults also affect the selection of

key bytes in decryption; hence, even in a case where a fault is not detected by decryption, there is no deterministic relationship between the key bytes and the faulty outputs.

In addition, public nonces or initialization vectors (IVs) can be reused in CLAE. Even if this happens, 1-bit difference of messages or associated data is still encrypted into completely different ciphertexts under the same key.

2 Specification

In this section, after introducing notations and preliminary functions, the algorithms of key generation, encryption, and decryption will be described.

2.1 Notations

Given a byte array a of length l (e.g., a key, a message, a ciphertext, or the associated data), $a[i]$ $(0 \leq i \leq l\text{-}1)$ indicates the $(i+1)$th byte in a. $\oplus$ denotes exclusive-or (XOR) operation, $|$ indicating bitwise `or`, $\&$ for bitwise `and`. The right and left shift operations are indicated by $\gg$ and $\ll$, respectively.

Given two bytes b_0 and b_1, $b_0 \hat{+} b_1 = b_0 + b_1 \bmod 256$, $b_0 \hat{-} b_1 = b_0 - b_1 \bmod 256$, and $b_0 \hat{*} b_1 = b_0 * b_1 \bmod 256$. $b_0 == b_1$ returns 1, if they are equal, otherwise 0.

The secret key is denoted by k, which is 16-byte long in this submission. The number of public nonces or initialization vectors (i.e., CRYPTO_NPUBBYTES in api.h) is denoted by $\mathtt{L}_n$, which should be an even number in this submission. The number of tags (i.e., CRYPTO_TAGBYTES in encrypt.c) is denoted by $\mathtt{L}_t$, both in bytes.

Algorithm 1: search(k, b, o)
`Input` :
16-byte key k
1-byte value b
`Output` :
updated 4-byte o
`Steps` :
$f = k[b\gg4] \hat{+} k[b\&0\text{x}0\text{F}]$
$b = (f\gg3 \mid f\ll5) \oplus b$
$b_0 = (b\&0\text{x}1\text{E})\gg1$; $b_1 = (b\&0\text{x}01\ll3) \mid b\gg5$
$b_2 = (b\&0\text{x}3\text{C})\gg2$; $b_3 = (b\&0\text{x}03\ll2) \mid b\gg6$
$o[0] = k[b_0] \hat{+} (b_2\ll1)$; $o[1] = k[b_1] \hat{+} (b_3\ll3)$
$o[2] = k[b_2] \hat{+} (b_1\ll1)$; $o[3] = k[b_3] \hat{+} (b_0\ll3)$

Fig. 1: Search in key bytes

Algorithm 2: vecIV($k, iv, l_{iv}, s_0, s_1, o$)

`Input` :
16-byte key k
nonces iv of l_{iv} bytes
two state bytes s_0 and s_1

`Output` :
updated o of two bytes

`Steps` :
$o[0] = s_0$; $o[1] = s_1$
for $i = 0, ..., l_{iv}$ - 1 do
$b_0 = iv[i]$&0x0F
$o[0] = (o[0] \hat{+} k[o[1]>>4] \hat{+} k[o[0]$&0x0F$] \hat{+} k[b_0]) \oplus b_0$
$b_1 = iv[l_{iv} - i - 1]>>4$
$o[1] = (o[1] \hat{+} k[o[0]>>4] \hat{+} k[o[1]$&0x0F$] \hat{+} k[b_1]) \oplus b_0 \oplus b_1$

Fig. 2: Generate o from iv and k based on s_0 and s_1

2.2 Preliminary Functions

Four preliminary functions are defined below.

search(k, b, o)**:** Based on the higher four bits and lower four bits in b, this algorithm in Figure 1 selects two bytes from the key k, that is, $k[b$&0x0F$]$ and $k[b>>4]$. Then, b is updated with their sum f. Based on updated b, four bytes from k are selected to update the output o. If there is one bit change in the input b, f and then o can be totally different.

vecIV$(k, iv, l_{iv}, s_0, s_1, o)$**:** This function given in Figure 2 merges the input iv into the 2-byte array o, based on two state bytes and the key. Each byte of intermediate o is dependent on only four bits of iv; thus, even if iv is changed to generate different power traces, there is no power consumption difference for 4 bits of each byte in intermediate o. Also, the change of iv also changes the selected key bytes $k[b_0]$ and $k[b_1]$, leading to different key byte accesses.

spill(k, c, l_c, i)**:** This function in Figure 3 merges the key bytes selected with $c[i]$ to other three ciphertext bytes. If $c[i]$ is changed due to attacks, three other ciphertext bytes, one of them probably at a different position, will be updated with different key bytes.

spillA(k, s, c, l_c, i, b)**:** Similarly, this algorithm merges associated data into ciphertexts. This function is stateful. An adversary can control the associated data, but not the state s. Hence, the output of key byte search cannot be manipulated by the adversary by making up associated data. An intermediate ciphertext byte in this function is also affected by only 4 bits of associated data.

2.3 Key Generation

The secret key k in CLAE is 128-bit (or 16-byte) long. Each byte in k can be any number and there is no extra processing of the secret key.

Algorithm 3: spill(k, c, l_c, i)

`Input` :

16-byte key k, ciphertext c of l_c bytes, $l_c \geq 16$
$i \in \{0, l_c\text{ - }1\}$

`Output` :

updated c

`Steps` :

search($k, c[i], o$)
if $i > 0$ then $l = i$ - 1 else $l = l_c$ - 1
if $i < l_c$ - 1 then $r_0 = i + 1$ else $r_0 = 0$
$r_1 = r_0 + ((o[0] \hat{+} o[3])$&0x0F)
if $r_1 \geq l_c$ then $r_1 = r_1$ - l_c
if $r_1 = i$ then
  if $r_0 < l_c$ - 1 then $r_1 = r_0 + 1$ else $r_1 = 0$
$c[l] = c[l] \oplus (o[0] \hat{+} o[1])$
$c[r_1] = c[r_1] \oplus (o[1] \hat{+} o[2])$
$c[r_0] = c[r_0] \oplus (o[2] \hat{+} o[3])$

Fig. 3: Propagation of ciphertext

Algorithm 4: spillA(k, s, c, l_c, i, b)

`Input` :

16-byte key k, 1-byte state s, ciphertext c of l_c bytes, $l_c \geq 16$
$i \in \{0, l_c\}$, 1-byte value b

`Output` :

a new state, and updated c

`Steps` :

search($k, (k[b$>>4$] \hat{+} k[b$&0x0F$] \hat{+} s) \oplus (b$>>4$), o$)
$l_0 = i + (s$&0x0F$)$
if $l_0 \geq l_c$ then $l_0 = l_0$ - l_c
$r_0 = i + ((o[0] \hat{+} o[1])$&0x0F$)$
if $r_0 \geq l_c$ then $r_0 = r_0$ - l_c
$c[l_0] = c[l_0] \oplus (o[0] \hat{+} o[1] \hat{+} s) \oplus (b$&0xAA$)$
$c[r_0] = c[r_0] \oplus (o[2] \hat{+} o[3] \hat{+} s) \oplus (b$&0x55$)$
return $o[0] \hat{+} o[2] \hat{+} o[3]$

Fig. 4: Propagation of associate data

2.4 Encryption

The encryption algorithm is given in Figure 7, which depends two byte encryption algorithms, called 2-byte encryption and 1-byte encryption, given in Figure 5 and Figure 6, respectively.

In 2-byte encryption, according to the state byte $s[0]$, four bytes from the secret key k are selected and used together with two byte arrays derived from IV to define the two ciphertext bytes $c[0]$ and $c[1]$. Their definition is in a style similar to samples in LWE [3], in the sense that $c[0]$ or $c[1]$ can be regarded as containing 4-bit errors (or unknown values) for the adversary, since the adversary can determine at most 4 bits in each of them by varying the message m and analyzing power differences.

The value of c obtained just after $c[0] = c[0] \hat{+} (c[1] \gg 3 | c[1] \ll 5)$ is used as the updated state s to influence the encryption of the next message byte. After $o = c$, the ciphertext c is updated with s. This update is exploited by the decryption algorithm to propagate changes in ciphertexts, as to be explained more later.

In the 1-byte encryption algorithm, based on the state s, four bytes from k are selected. Then, state bytes and key bytes are used to generate the 1-byte ciphertext c in two steps. In each step, only 4-bits of m is XORed with secret values.

Algorithm 5a: $\mathbb{E}_{\mathtt{b}}(k, s, iv, m, c)$
`Input` :
16 - byte key k
2 - byte state s, nonces iv of $\mathtt{L}_n$ bytes
1 - byte message m, 2-byte array c
`Output` :
2-byte updated ciphertext c
2-byte updated state s
`Steps` :
search(k, $s[0]$, o)
vecIV$(k, iv, \mathtt{L}_n/2, s[0], s[1], b)$
$c[0] = (o[0] \hat{*} b[0] \hat{+} o[1] \hat{*} b[1]) \oplus (m \& 0\text{x}0\text{F})$
vecIV$(k, iv + \mathtt{L}_n/2, \mathtt{L}_n/2, s[1], c[0], b)$
$c[1] = (o[2] \hat{*} b[0] \hat{+} o[3] \hat{*} b[1]) \oplus (m \gg 4)$
$c[0] = c[0] \hat{+} (c[1] \gg 3 \mid c[1] \ll 5)$
$o = c$
$c[0] = c[0] \hat{+} s[1]$
$c[1] = c[1] \hat{+} s[0]$
$s = o$

Fig. 5: 2-byte encryption for 2-byte ciphertext

The encryption algorithm first determines the number l_p of padding bytes. For a message of l_m bytes, if $l_m < \mathtt{L}_t$, then $\mathtt{L}_t - l_m$ bytes need to be padded. CLAE pads l_p bytes of 0xFF for a short message. The length of ciphertext is $\mathtt{L}_t + l_m + l_p$. The size gap

Algorithm 5b: $\mathbb{E}_{\mathtt{b}'}(k, s_0, s_1, m)$
`Input` :
16 - byte key k
two byte states s_0, s_1
1 - byte message m
`Output` :
1-byte ciphertext c
`Steps` :
search(k, $s_0 \oplus s_1$, o)
c_0 = $(s_1 \hat{*} o[3] \hat{+} s_0 \hat{*} o[2]) \oplus (m\&0\text{x}0\text{F})$
c_1 = $(s_0 \hat{*} o[1] \hat{+} s_1 \hat{*} o[0]) \oplus (m\&0\text{x}\text{F}0)$
return $c_0 \oplus c_1$

Fig. 6: 1-byte encryption for 1-byte ciphertext

between the message and the ciphertext is at most $2 * \mathtt{L}_t$ when $l_m = 0$. If $l_m \geq \mathtt{L}_t$, the gap is $\mathtt{L}_t$. The 2-byte initial state s is generated with the vecIV function.

Based on the initial state, the first $\mathtt{L}_t$ message bytes each is encrypted into a 2-byte ciphertext, with the state updated. Following the 2-byte encryption is a loop updating the 2-byte ciphertext of a message byte with the 2-byte ciphertext of the succeeding message byte. The updates in this loop propagate ciphertext changes in the decryption algorithm.

After the loop of 2-byte ciphertext updating, the encryption algorithm encrypts the remaining message bytes each into a 1-byte ciphertext. The 1-byte encryption algorithm refers to the 2-byte ciphertexts and the previous ciphertext byte as the states to randomize the 1-byte ciphertexts. After each 1-byte encryption, the 1-byte ciphertext and the message byte are propagated into the preceding ciphertexts. Such propagation helps detect the changes in the 1-byte ciphertext or message m in decryption.

For each byte of the associated data, the encryption algorithm uses the spillA algorithm to merge it into other two ciphertext bytes. Then, a loop updates a ciphertext byte $c[l_c - i - 1]$ with its succeeding one $c[l_c - i]$. At last, the encryption algorithm uses the spill algorithm twice to merge each byte in the ciphertext with other ciphertext bytes. The spill and spillA algorithms ensure the adversary cannot make valid changes to ciphertexts and associated data, without knowing the secret key. The repeated application of spill algorithm is to increase the resistance to differential analysis attack, as to be discussed later.

In the encryption algorithm, key bytes are selected in a nondeterministic way. Given a key, the sequence of key bytes accessed via the search algorithm or the vecIV function is random, depending on public nonces, messages, and associated data. For two encryptions of the same message with different public nonces, the access sequences of key bytes can be different. A key byte can be repeatedly selected, or not selected when encrypting a message. In addition, a longer message needs to access more key bytes to encrypt.

Algorithm 6: $\mathbb{E}(c, l_c, m, l_m, a, l_a, iv, k)$

`Input` :

16-byte key k, 12-byte nonces iv
l_m-byte message m
buffer c and its length l_c
associated data a and its length l_a

`Output` :

ciphertext c
updated l_c

`Steps` :

if $l_m < \mathtt{L}_t$ then $l_p = \mathtt{L}_t$ - l_m else $l_p = 0$
$l_c = \mathtt{L}_t + l_m + l_p$
Let s be 2-byte array, and $s[0] = 0$
for $i = 0, ..., 15$ do $s[0] = s[0] \hat{+} k[i]$
vecIV$(k, iv, \mathtt{L}_n, s[0], s[0]{>>}3 | s[0]{<<}3, s)$
for $i = 0, ..., \mathtt{L}_t$ - 1 do
  if $i < l_m$ then $\mathbb{E}_{\mathtt{b}}(k, s, iv, m[i], c + i * 2)$
  else $\mathbb{E}_{\mathtt{b}}(k, s, iv, 0\text{xFF}, c + i * 2)$
for $i = 0, ..., \mathtt{L}_t$ - 2 do
  $c[i * 2 + 0] = c[i * 2 + 0] \hat{+} c[(i + 1) * 2 + 0]$
  $c[i * 2 + 1] = c[i * 2 + 1] \hat{+} c[(i + 1) * 2 + 1]$
for $i = \mathtt{L}_t, ..., l_m$ - 1 do
  $c[2 * \mathtt{L}_t + i - \mathtt{L}_t] = \mathbb{E}_{\mathtt{b}'}(k, c[i - \mathtt{L}_t], c[2 * \mathtt{L}_t + i - \mathtt{L}_t - 1], m[i])$
  $c[i - \mathtt{L}_t + 1] = (c[i - \mathtt{L}_t + 1] \hat{+} c[2 * \mathtt{L}_t + i - \mathtt{L}_t]) \oplus (m[i] \& 0\text{xAA})$
  $c[i - \mathtt{L}_t + 2] = (c[i - \mathtt{L}_t + 2] \hat{+} c[2 * \mathtt{L}_t + i - \mathtt{L}_t]) \oplus (m[i] \& 0\text{x55})$
$s[0] = 0; j = 0$
for $i = 0, ..., 15$ do $s[0] = s[0] \hat{+} k[i]$
for $i = 0, ..., l_a$ - 1 do
  $s[0]$=spillA$(k, s[0], c, l_c, j, a[i])$
  $j = j + 1$; if $j > l_c$ then $j = 0$
for $i = 1, ..., l_c$ - 1 do
  $c[l_c - i - 1] = c[l_c - i - 1] \hat{+} (c[l_c - i]{>>}5 | c[l_c - i]{<<}3)$
for $i = 0, ..., l_c$ - 1 do
  spill(k, c, l_c, i)
for $i = 0, ..., l_c$ - 1 do
  spill(k, c, l_c, i)

Fig. 7: Encryption

2.5 Decryption

The decryption algorithm in Figure 10 relies on two byte decryption algorithms, called 2-byte decryption and 1-byte decryption, as shown in Figure 8 and Figure 9, respectively.

The 2-byte decryption algorithm starts with updating the 2-byte ciphertext with the state s. The updated 2-byte ciphertext then becomes the state to decrypt the next message byte. Hence, either a changed input state or the changed 2-byte ciphertext affects the subsequent states. In the 2-byte decryption, the 1-byte redundancy (i.e., higher 4 bits in m_0 and m_1, respectively) is used to check the integrity of the 2-byte ciphertext. A failed check returns -1; otherwise, the message byte is returned. The 1-byte decryption algorithm simply recovers the encrypted message byte.

Algorithm 7a: $\mathbb{D}_b(k, s, iv, c)$
`Input` :
16-byte key k
2-byte state s, nonces iv of $\mathtt{L}_n$ bytes
2-byte ciphertext c
`Output` :
2-byte updated state s
1-byte message or -1
`Steps` :
$c[0] = c[0] \hat{} s[1]$
$c[1] = c[1] \hat{} s[0]$
$c_0 = c[0] \hat{} (c[1] \gg 3 \mid c[1] \ll 5)$
search(k, $s[0]$, o)
vecIV$(k, iv + \mathtt{L}_n/2, \mathtt{L}_n/2, s[1], c_0, b)$
$m_1 = c[1] \oplus (o[2] \hat{*} b[0] \hat{+} o[3] \hat{*} b[1])$
vecIV$(k, iv, \mathtt{L}_n/2, s[0], s[1], b)$
$m_0 = c_0 \oplus (o[0] \hat{*} b[0] \hat{+} o[1] \hat{*} b[1])$
$s = c$
if $m_0 >$ 0x0F or $m_1 >$ 0x0F then return -1
else return m_0+($m_1 \ll 4$)

Fig. 8: Byte decryption for 2-byte ciphertext

The decryption algorithm first determines l_m. After that, it applies the spill and spillA algorithms to update all ciphertext bytes. The spill and spillA algorithms rely on the search algorithm. Any change in ciphertext or associated data leads the search algorithm to returning different key bytes, thus propagating changes to other bytes in the ciphertext.

Then, the 1-byte decryption algorithm is applied. Any fault induced to the decrypted m or its ciphertext is propagated to the preceding ciphertexts, and eventually propagated to the 2-byte ciphertexts. The 1-byte decryption is followed by the loop of updating 2-byte ciphertexts. This loop propagates the changes from the last 2-byte ciphertext to

Algorithm 7b: $\mathbb{D}_{\mathtt{b}'}(k, s_0, s_1, c)$
`Input` :
16 - byte key k
two byte states s_0, s_1
1 - byte ciphertext c
`Output` :
1-byte message m
`Steps` :
search(k, $s_0 \oplus s_1$, o)
$m = c \oplus (s_1 \hat{*} o[3] \hat{+} s_0 \hat{*} o[2]) \oplus (s_0 \hat{*} o[1] \hat{+} s_1 \hat{*} o[0])$
return m

Fig. 9: Byte decryption for 1-byte ciphertext

the first 2-byte ciphertext. Thus, each 2-byte decryption has the chance to detect the change.

When the 2-byte decryption algorithm return -1, the decryption algorithm continues to decrypt the subsequent bytes. Thus, the decryption time does not leak the position of the first message byte that fails the integrity check. At last, the bytes 0xFF at the tail of the message are supposed to be padded bytes, thus removed.

3 Security Analysis

The semantic security of CLAE and its resistance to side channel and fault attacks come from the following properties, which will be evaluated with experiments in this section.

- The 2-byte encryption algorithm generates pseudorandom 2-byte ciphertext for a message byte;
- Two nonces even with 1-bit difference randomize the whole ciphertexts including the output of 1-byte encryption differently;
- The bytes in the secret key are accessed randomly in encryption via the search and vecIV algorithms;
- The 2-byte decryption algorithm will fail if the 2-ciphertext is tampered with;
- A change to ciphertexts or associated data is propagated to 2-byte ciphertexts, where it is detected.

Given any two different messages with the same length, one of them is encrypted. Due to the first two properties, the cipherext is randomized for every cipherext byte, making it hard to distinguish which message is encrypted. This is the basic idea of semantic security of CLAE.

3.1 Security Claim and Differential Cryptanalysis

The differential cryptanalytic attack to CLAE requires 2^{128} computations on a classical computer in a single key setting, as estimated below.

Algorithm 8: $\mathbb{D}(m, l_m, c, l_c, a, l_a, iv, k)$

`Input` :

16-byte key k, nonces iv of $\mathtt{L}_n$ bytes
ciphertext c and its length l_c
buffer m and its length l_m
associated data a and its length l_a

`Output` :

0 or -1
updated m and l_m

`Steps` :

$l_m = l_c$ - $\mathtt{L}_t$
for $i = 0, ..., l_c$ - 1 do
  spill$(k, c, l_c, l_c - i - 1)$
for $i = 0, ..., l_c$ - 1 do
  spill$(k, c, l_c, l_c - i - 1)$
for $i = 1, ..., l_c$ - 1 do
  $c[i - 1] = c[i - 1] \hat{} (c[i] >> 5 | << 3)$
Let s be 2-byte array
$s[0] = 0; j = 0$
for $i = 0, ..., 15$ do $s[0] = s[0] \hat{+} k[i]$
for $i = 0, ..., l_a$ - 1 do
  s[0] = spillA$(k, s[0], c, l_c, j, a[i])$
  $j = j + 1$; if $j > l_c$ then $j = 0$
for $i = l_m$ - 1, ..., $\mathtt{L}_t$ do
  $m[i] = \mathbb{D}_{\mathtt{b}'}(k, c[i - \mathtt{L}_t], c[2 * \mathtt{L}_t + i - \mathtt{L}_t - 1], c[2 * \mathtt{L}_t + i - \mathtt{L}_t])$
  $c[i - \mathtt{L}_t + 1] = (c[i - \mathtt{L}_t + 1] \oplus (m[i] \& 0\text{xAA})) \hat{} c[2 * \mathtt{L}_t + i - \mathtt{L}_t]$
  $c[i - \mathtt{L}_t + 2] = (c[i - \mathtt{L}_t + 2] \oplus (m[i] \& 0\text{x55})) \hat{} c[2 * \mathtt{L}_t + i - \mathtt{L}_t]$
for $i = \mathtt{L}_t$ - 2, ..., 0 do
  $c[i * 2 + 0] = c[i * 2 + 0] \hat{} c[(i + 1) * 2 + 0]$
  $c[i * 2 + 1] = c[i * 2 + 1] \hat{} c[(i + 1) * 2 + 1]$
$s[0] = 0; j = 0$
for $i = 0, ..., 15$ do $s[0] = s[0] \hat{+} k[i]$
vecIV$(k, iv, \mathtt{L}_n, s[0], s[0] >> 3 | s[0] << 5, s)$
for $i = 0, ..., \mathtt{L}_t$ - 1 do
  $m[i] = \mathbb{D}_{\mathtt{b}}(k, s, iv, c + i * 2)$
  $j = j + (m[i] == 0)$
if $l_m = \mathtt{L}_t$ then
  for $i = 0, ..., \mathtt{L}_t$ - 1 do
    if $m[\mathtt{L}_t - i - 1] = 0$xFF then $l_m = l_m$ - 1
    else break
return -1*$(j > 0)$

Fig. 10: Decryption

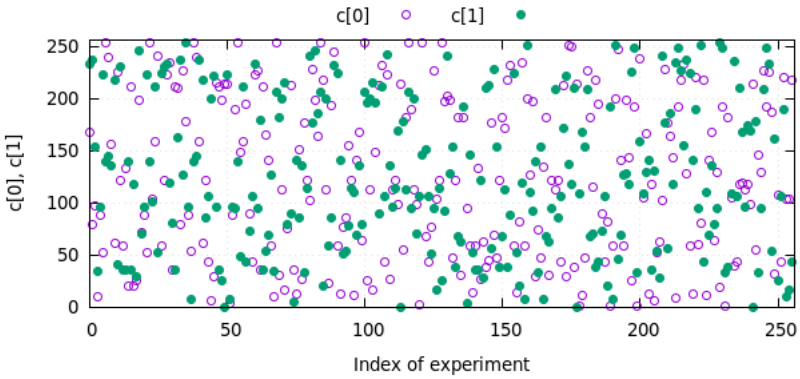

Fig. 11: Randomness of 2-byte encryption

The adversary controls public nonces, messages, and associated data. If the input differences come from nonces, the vecIV function creates different initial states, which has differences caused by different key byte combinations and not known by the adversary. Then, according to the first two properties above, the 2-byte encryption generates pseudorandom intermediate ciphertexts and such pseudorandomness is propagated across the whole intermediate values and ciphertexts. Hence, the adversary cannot decide meaningful differences between intermediate values from the differences of nonces.

With the same public nonce, but different messages or associated data, the intermediate values before the first spill in the encryption algorithm can reflect the input differences. If two messages longer than $\mathtt{L}_t$ are only different at the last byte, then the last byte in the intermediate values have the same difference. With the same message and nonce, if one encryption has no associated data and the other has 1-byte associated data, then most bytes of the two intermediate values will be the same. The encryption algorithm of CLAE exploits two spill at its last two steps to protect each byte of such intermediate values.

If the adversary intends to recover a byte in two intermediate values generated from two messages or associated data with particular differences, the adversary has to guess key bytes used in the two applications of spill algorithm. In the following analysis, we suppose the last byte of intermediate values is targeted by the adversary and estimate the number of key bytes the adversary has to guess. The last ciphertext $c[l_c - 1]$ is obtained by XORing a sum of two key bytes selected according to intermediate values of $c[l_c - 2]$ and $c[0]$ four times in the two applications of spill. If intermediate values of $c[l_c - 2]$ and $c[0]$ are known, $4 * 4$ key bytes has to be guessed from the 4-time use of the search algorithm.

However, the intermediate values of $c[l_c - 2]$ and $c[0]$ are unknown and they are XORed with key byte combinations twice searched according to $c[l_c]$ and its intermediate value. This requires $2 * 6$ key byte guesses from two search algorithm. In addition, two intermediate values of $c[0]$ are XORed with the key byte sum selected according to

$c[1]$ in two spill. The adversary needs to guess another two bytes for this key byte sum. Hence, for one intermediate value the adversary wants to know, $16+12$ key bytes and 2 summed key bytes have to be guessed, and for two intermediate values, there will be 56 key bytes and 4 summed key bytes to guess, which is much longer than 16-byte secret key due to repeated key bytes used by spill. Based on this estimation, CLAE requires 2^{128} computations for differential cryptanalysis on a classical computer in a single key setting.

3.2 Resistance to Side Channel Attacks

Power analysis (PA) and differential power analysis (DPA) are side-channel attacks. To successfully perform PA or DPA attacks, the adversary needs to know a deterministic relationship between known data inputs (such as known plaintexts and public nonces) and a subkey [2]. Then, the power traces from different known data inputs can be used to derive the subkey. From the third property, CLAE is resistant to PA or DPA, because the adversary cannot know exactly the sequence of key bytes accessed in encryption. In addition, a secret byte is XORed with 4 bits of a value controlled by the adversary. Thus, 4 bits in a secret byte cannot be leaked by the difference of power traces, regarded as 4-bit error in a secret intermediate value. Based on hardness of LWE, the secret key bytes in the partially exposed intermediate values cannot be recovered.

CLAE is also resistant to timing attack, since no execution branches of conditional statements are selected according to secret values. Even if a message byte decryption might fail, the decryption algorithm does not change its execution time. The length of messages or associated data affect the execution time. However, their length is supposed to be public.

In fault attacks, the adversary needs to observe the output of decrypting faulty ciphertexts or faulty intermediate ciphertexts. Due to the fourth and fifth properties, the byte encryption just return -1 when a fault is induced. Thus, fault attacks cannot happen because there is no faulty output to observe. Additionally, based on the third property, the access sequence of key bytes is random in encryption, and faults also change the access sequence of key bytes in decryption from where the fault comes into effect. Hence, even one faulty output is observed, the adversary cannot link faulty outputs to the nondeterministic key bytes used in encryption and decryption.

3.3 Experiment Evaluation of Properties

The five properties described above is the basis of CLAE security. In this section, we check these properties with experiments. In these experiments, the key is 16 bytes from 0 to 15, and the public nonce is 12 bytes of 0x00.

3.3.1 Randomness of 2-Byte Encryption In this experiment, the 2-byte encryption algorithm uses the same key to encrypt the byte 0x00 for 256 times, with the 6th byte of public nonces increasing from 0 to 255, respectively. As shown in Figure 11, the 2-byte ciphertext $c[0]$ and $c[1]$ are changed randomly even if the 6th nonce byte changes only one bit each time. This experiment confirms the pseudorandomness of the 2-byte encryption algorithm.

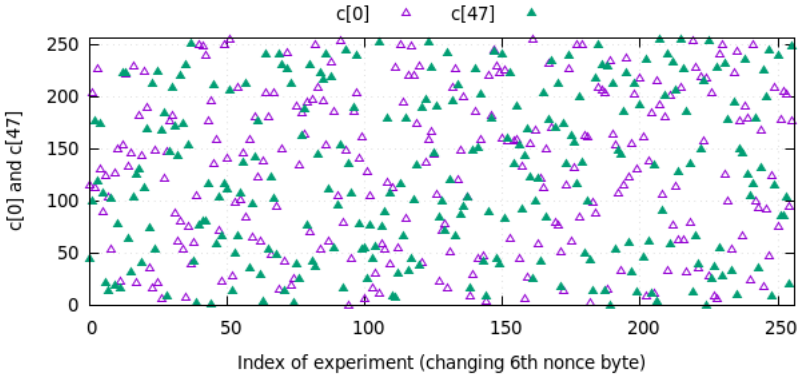

Fig. 12: $c[0]$ and $c[47]$ of encrypting 40-byte 0x00, $\mathrm{L}_t = 8$

3.3.2 Randomness Propagation in Encryption Let $\mathrm{L}_t = 8$. In this experiment, the message is 40 bytes of 0x00. It is encrypted with the same key for 256 times, with only the 6th nonce byte increasing from 0 to 255. The first ciphertext byte $c[0]$ and the last byte $c[47]$ are observed for the 256-time encryptions. Figure 12 shows the values $c[0]$ and $c[47]$, which are randomized between 0 and 255. This experiment confirms the randomness propagation of CLAE encryption algorithm; otherwise, $c[0]$ and $c[47]$ will not change with the 6th nonce byte.

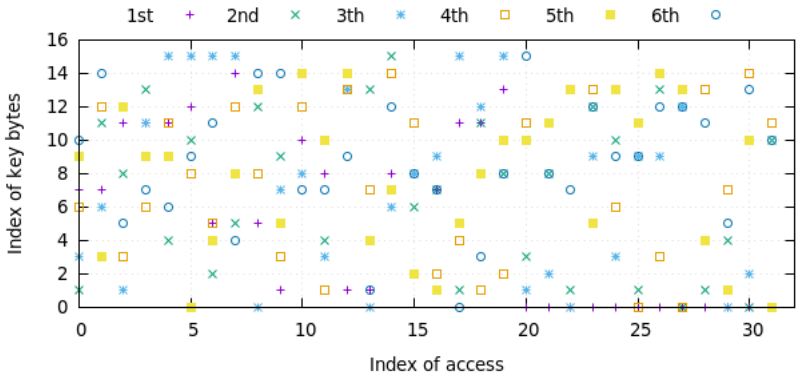

Fig. 13: Partial key byte sequence of encrypting 8-byte 0x00, $\mathrm{L}_t = 8$

3.3.3 Random Access Sequence of Key Bytes in Encryption In this experiment, $\mathrm{L}_t = 8$. With the same key, the 8-byte message 0x00 is encrypted six times, with the

6th nonce byte increasing from 0 to 5. There is no associated data in this experiment. The key bytes are accessed via the search and vecIV functions. Figure 13 shows a part of the access sequences of key bytes via the search algorithm. It shows that the access sequences are dynamically changing, even with one bit change of the nonce.

3.3.4 Integrity Check of 2-Byte Decryption In this experiment, the 2-byte encryption algorithm encrypts the byte 0x00 into a 2-byte ciphertext. For the 2-byte ciphertext, it can be tampered with by at most $2^{16} - 1$ ways. This experiment is repeated with 256 times, with the 6th nonce byte changing from 0 to 255. For each repeated experiment, among $2^{16} - 1$ changes, our experiment shows that $2^8 - 1$ changes pass the integrity check. Since the 2-byte ciphertext contains only 1-byte redundancy, the detection rate $255/65535$ is expected.

3.3.5 Change Propagation in Decryption The above experiment shows that the brute-force changes to the 2-byte ciphertext of a 1-byte redundancy lead to 255 successful attacks among $2^{16} - 1$ ones. In this experiment, we still use the same attack method as in the last experiment (i.e., changing the first two ciphertext bytes in a brute-force way), but attack the ciphertexts with 2-byte redundancy (i.e., $\mathtt{L}_t = 2$). Among 256 experiments, the 6th byte of the nonce increases from 0 to 255. The message is 40 bytes of 0x00. The similar code is also applied to tamper with associated data of 50 bytes.

The change propagation property of CLAE decryption algorithm should ensure that the 2-byte ciphertext of each message byte is tainted. Thus, the changes can be detected with 2-byte redundancies and the number of successful attacks is expected to significantly reduce. This expectation is confirmed in the Figure 14, which shows the number of successful attacks is decreased from 255 with 1-byte redundancy to a small number (i.e., 0, 1, 2, ...) with 2-byte redundancy in each experiment.

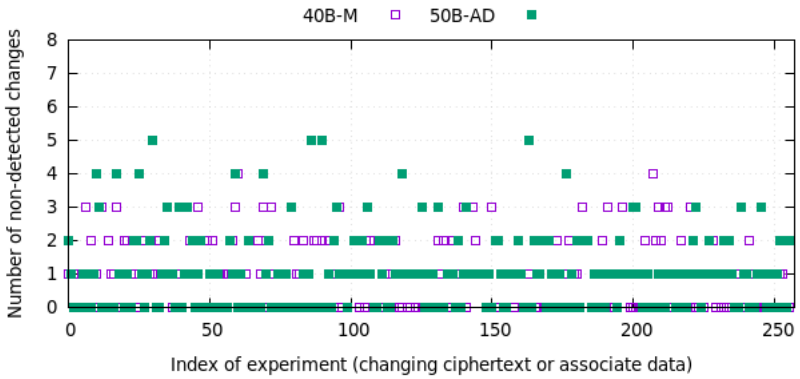

Fig. 14: Change propagation among ciphertexts: 40-byte messages, 50-byte AD, $\mathtt{L}_t = 2$

3.3.6 Reuse of Nonces In this experiment, we use the same key and nonces to encrypt messages of 1 bit difference, without associated data. The initial message is 40 bytes of 0x00. Also, we encrypt the initial message but with the 50-byte associated data changed 1-bit each time. The changes are applied to message byte $m[20]$ and associated byte $ad[25]$. The experiment results are shown in Figure 15 and Figure 16. Even with the same key and nonces, the ciphertext bytes are completely different for each encryption.

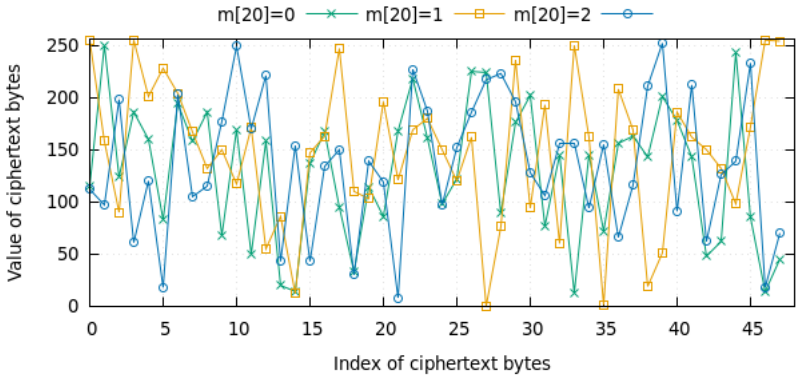

Fig. 15: Nonce reuse with 1-bit message changing; $\mathrm{L}_t = 8$

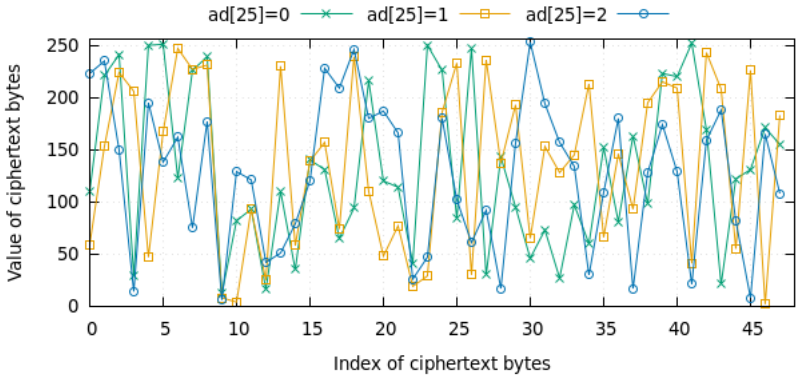

Fig. 16: Nonce reuse with 1-bit associated data changing; $\mathrm{L}_t = 8$

In another experiment, we increase the first byte of the secret key from 0 to 1, and then 2, without changing the nonce, the message, and the associated data. The three

ciphertexts are shown in Figure 17. One bit change of key also leads to completely different ciphertexts.

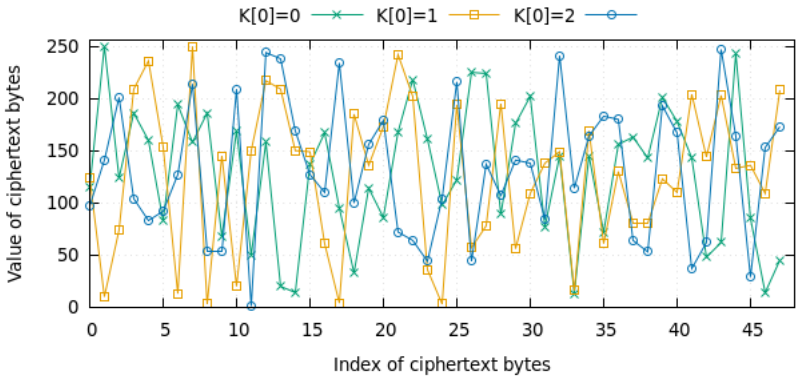

Fig. 17: 1-bit change of key with same nonce, message, AD; $\mathtt{L}_t = 8$

4 Parameter Configuration and Memory Usage

CLAE is specified at the 128-bit security level, with a 128-bit secret key. The length of public nonces is configured as 12 ($\mathtt{L}_n = 12$) in the reference implementation, with $\mathtt{L}_n = 8$ and $\mathtt{L}_n = 16$ also tested. The length of authentication tags $\mathtt{L}_t$ is configured as 8 bytes by default; the cases $\mathtt{L}_t = 12$ and $\mathtt{L}_t = 16$ are tested. For bigger $\mathtt{L}_t$, the value of CRYPTO_ABYTES needs to be accordingly adjusted.

The configuration in the reference implementation can encrypt more than 2^{50} messages under one key, because one bit change of nonces leads to a completely different ciphertext.

CLAE needs the necessary storage of 16 bytes for the secret key, l_m bytes for messages, $\mathtt{L}_t + l_m + l_p$ bytes for ciphertexts (as discussed in the encryption algorithm), l_a bytes of associated data, and $\mathtt{L}_n$ bytes for the public nonces. In addition, it needs a few extra bytes for storing local integer variables like i in a loop, the 4-byte array o, the 2-byte state s, two bytes for the output of vecIV function.

5 Advantages and Limitations

The advantages of CLAE have been explained in the previous section. The following is the summary:

– CLAE permits the reuse of public nonces, with two messages of 1-bit difference encrypted into completely different ciphertexts under the same key and the same nonce.

– CLAE resists side-channel and fault attacks inherently; the implementation does not need extra countermeasures to enhance CLAE for resisting such attacks.
– The security of CLAE can be explained with a few high-level properties, making design rationale and implementation easier to understand.
– In addition to the necessary memory requirement, CLAE does not need much extra memory for all algorithms in the specification.

A limitation of CLAE is that it pads a short message simply by appending byte 0xFF. Hence, a short message to be encrypted should avoid 0xFF as its tail bytes. In addition, a message of 0 byte long is padded and encrypted into a $2 * \mathtt{L}_t$ ciphertext, leading to the big gap (e.g., 16 bytes for $\mathtt{L}_t = 8$) between the message length and the ciphertext length.

In addition, this specification considers only secret keys of 128 bits long. To support a longer key of 256 bits, the algorithms like search and vecIV can be revised to select key bytes with 5-bit combination of a byte, instead of 4-bit combinations in this specification.

References

1. Dobraunig, C., Mangard, S., Mendel, F., Primas, R.: Fault attacks on nonce-based authenticated encryption: Application to keyak and ketje. In: Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers. pp. 257–277 (2018)
2. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology. pp. 388–397. CRYPTO '99 (1999)
3. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing. pp. 84–93. STOC '05, ACM (2005)