ACE: An Authenticated Encryption and Hash Algorithm

Submission to the NIST LWC Competition

Submitters/Designers:
Mark Aagaard, Riham AlTawy, Guang Gong,
Kalikinkar Mandal, and Raghvendra Rohit*

*Corresponding submitter:
Email: rsrohit@uwaterloo.ca
Tel: +1-519-888-4567 x45650

Communication Security Lab
Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, ON, N2L 3G1, CANADA

http://comsec.uwaterloo.ca/

March 29, 2019

Contents

1 Introduction **4**
- 1.1 Notations . . . 5
- 1.2 Outline . . . 6

2 Specification **7**
- 2.1 Parameters . . . 7
  - 2.1.1 ACE AEAD algorithm . . . 7
  - 2.1.2 ACE Hash algorithm . . . 8
- 2.2 Recommended Parameter Set . . . 8
- 2.3 The ACE Permutation . . . 8
  - 2.3.1 The nonlinear function SB-64 . . . 8
  - 2.3.2 Round and step constants . . . 9
- 2.4 AEAD Algorithm: ACE-$\mathcal{AE}$-128 . . . 10
  - 2.4.1 Rate and capacity part of state . . . 12
  - 2.4.2 Padding . . . 12
  - 2.4.3 Loading key and nonce . . . 13
  - 2.4.4 Initialization . . . 13
  - 2.4.5 Processing associated data . . . 13
  - 2.4.6 Encryption . . . 13
  - 2.4.7 Finalization . . . 14
  - 2.4.8 Decryption . . . 14
- 2.5 Hash Algorithm: ACE-$\mathcal{H}$-256 . . . 14
  - 2.5.1 Message padding . . . 15
  - 2.5.2 Loading initialization vector . . . 15
  - 2.5.3 Initialization . . . 15
  - 2.5.4 Absorbing and squeezing . . . 15

3 Security Claims **17**

4 Security Analysis **18**
- 4.1 Diffusion . . . 18
- 4.2 Differential and Linear Cryptanalysis . . . 18

4.2.1 Expected bounds on the maximum probabilities of differential and linear characteristics . . . 19
4.3 Algebraic Properties . . . 19
4.4 Self Symmetry-based Distinguishers . . . 21
4.5 Security of ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256 . . . 22

5 Design Rationale **23**
5.1 Choice of the Mode: sLiSCP Sponge Mode . . . 23
5.2 ACE State Size . . . 24
5.3 ACE Step Function . . . 25
5.4 Nonlinear Layer: Simeck box (SB-64) . . . 25
5.5 Linear Layer: $\pi = (3, 2, 0, 4, 1)$ . . . 26
5.6 Round and Step Constants . . . 26
5.6.1 Rationale . . . 26
5.6.2 Generation of round and step constants . . . 26
5.7 Number of Rounds and Steps . . . 28
5.8 Choice of Rate Positions . . . 29
5.9 Statement . . . 29

6 Hardware Design and Analysis **30**
6.1 Hardware Design Principles . . . 30
6.2 Interface and Top-level Module . . . 31
6.3 The ACE_module . . . 32
6.3.1 ACE datapath . . . 32
6.3.2 ACE FSM and lfsr_c . . . 34
6.4 Hardware Implementation Results . . . 34
6.4.1 Hardware tools configuration . . . 34
6.4.2 Performance results . . . 35

7 Efficiency Analysis in Software **36**
7.1 Software: High-performance CPU . . . 36
7.2 Software: Microcontroller . . . 39

A Other NIST-LWC Submissions **45**

B Test Vectors **46**
B.1 Simeck Sbox . . . 46
B.2 ACE Permutation . . . 47
B.3 ACE-$\mathcal{AE}$-128 . . . 47
B.4 ACE-$\mathcal{H}$-256 . . . 47

C Constants: Sequence to Hex Conversion **48**

Chapter 1

Introduction

ACE, often known as one of the strongest cards in a deck of cards, is an 320-bit lightweight permutation. It is designed to achieve a balance between hardware cost and software efficiency for both Authenticated Encryption with Associated Data (henceforth "AEAD") and hashing functionalities, while providing sufficient security margins. To accomplish these goals, ACE components and its mode of operation are adopted from well known and analyzed cryptographic primitives. In a nutshell, the design of ACE, its security, functionalities and the features it offers are described as follows.

- ACE core operations. Bitwise XORs and ANDs, left cyclic shifts and 64-bit word shuffles
- ACE nonlinear layer. Unkeyed round-reduced Simeck block cipher [27] with block-size of 64-bits, which provides good cryptographic properties and low hardware cost
- ACE linear layer. Five 64-bit words are shuffled in an $(3, 2, 0, 4, 1)$ order, which offers good resistance against differential and linear cryptanalysis
- ACE security. Simple analysis and good bounds for security using automated tools such as CryptoSMT solver [22] and Gurobi [1]
- Functionality. All-in-one primitive, provides both AEAD and hashing functionalities using the same hardware circuit
- ACE mode of operation. Unified sponge duplex mode [4] with keyed initialization and finalization phases forthe AEAD algorithm
- Security of ACE modes. 128-bit security
- Hardware performance. Achieves a throughput of 476 Mbps and has an area of 4286 GE in CMOS 65 nm ASIC
- Software performance. Bit-sliced implementation of ACE permutation achieves a speed of 9.97 cycles/byte.

1.1 Notations

Notation	Description
$X \odot Y, X \oplus Y, X\|\|Y$	bitwise AND, XOR and concatenation of X and Y
$\|X\|$	length of X in bits
$\{0,1\}^n$, $\{0,1\}^\star$, ϕ	length n bitstring, variable length bitstring, empty string
1^n, 0^n	length n bitstring with all 1's, 0's
L^i	left cyclic shift operator, i.e., for $x \in \{0,1\}^n$, $\mathsf{L}^\mathsf{i}(x) = (x_i, x_{i+1}, \ldots, x_{n-1}, x_0, x_1, \ldots, x_{i-1})$
word	a 64-bit binary string
S	320 bit state of ACE
S_r, S_c	r-bit rate part and c-bit capacity part of S ($r = 64, c = 256$)
A, B, C, D, E	Five 64-bit words of S, i.e., $S = A\|\|B\|\|C\|\|D\|\|E$
S^i	state at i-th iteration (also step) of ACE permutation
$A[j]$	j-th byte of word A starting from right
A_1^i, A_0^i	upper and lower half of word A^i
K, N, T	key, nonce and tag
k, n, t	length of key, nonce and tag in bits ($k = n = t = 128$)
AD, M, C	associated data, plaintext and ciphertext (in blocks AD_i, M_i, C_i)
IV, iv	fixed initialization vector and its length in bits
H, h	message digest (in blocks H_i) and its length, $h = 256$
ℓ_X	length of X in words where $X \in \{AD, M, C, H\}$
step	one round of ACE permutation (see Figure 2.1)
round	one round of Simeck unkeyed function (see Figure 2.2)
SB-64	nonlinear operation of ACE permutation
u	number of rounds, $u = 8$
s	number of steps, $s = 16$
rc_0^i, rc_1^i, rc_2^i	8-bit round constants
sc_0^i, sc_1^i, sc_2^i	8-bit step constants
ACE-$\mathcal{AE}$-k	ACE AEAD algorithm ($k = 128$)
ACE-$\mathcal{H}$-h	ACE Hash algorithm ($h = 256$)

1.2 Outline

The rest of the document is organized as follows. In Chapter 2, we present the complete specification of the ACE permutation, ACE AEAD and ACE hash algrithms. We summarize the security claims of our submission in Chapter 3 and provide the detailed security analyis in Chapter 4. In Chapter 5, we present the rationale behind our design and justify the parameter choices. The details of our hardware implementations and performance results in CMOS 65 nm ASIC and FPGA are provided in Chapter 6. In Chapter 7, we discuss the efficiency of ACE in software including bit-sliced and microcontroller implementations. Finally, we conclude with references and test vectors in Appendix B.

Chapter 2

Specification

2.1 Parameters

ACE is an 320-bit permutation that operates in an unified duplex sponge mode [4] and offers both AEAD and hashing functionalities in a single hardware circuit. The AEAD algorithm (ACE-$\mathcal{AE}$-k) and the hash algorithm (ACE-$\mathcal{H}$-h) are parameterized by the size k of the secret key and the length of the message digest h in bits, respectively. Both the algorithms process the same amount of data per permutation call (i.e, rate r is same) and hence r value is ignored in the individual parameters' description.

2.1.1 ACE AEAD algorithm

The AEAD algorithm ACE-$\mathcal{AE}$-k is a combination of two algorithms, an authenticated encryption algorithm ACE-$\mathcal{E}$ and the verified decryption algorithm ACE-$\mathcal{D}$.

An authenticated encryption algorithm ACE-$\mathcal{E}$ takes as input a secret key K of length k bits, a public message number N (nonce) of size n bits, a block header AD (a.k.a, associated data) and a message M. The output of ACE-$\mathcal{E}$ is an authenticated ciphertext C of same length as M, and an authentication tag T of size t bits. Mathematically, ACE-$\mathcal{E}$ is defined as

$$\text{ACE-}\mathcal{E} : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^t$$

with

$$\text{ACE-}\mathcal{E}(K, N, AD, M) = (C, T).$$

The decryption and verification algorithm takes as input the secret key K, nonce N, associated data AD, ciphertext C and tag T, and outputs the plaintext M of same length as C only if the verification of tag is correct, and $\perp$ if the tag verification fails. More formally,

$$\text{ACE-}\mathcal{D}(K, N, AD, C, T) \in \{M, \perp\}.$$

2.1.2 ACE Hash algorithm

A hash algorithm takes as input a message M, and the standard initialization vector IV of length iv bits, and returns a fixed size output H, called hash or message digest. Formally, the hash algorithm using ACE permutation is specified by

$$\mathsf{ACE}\text{-}\mathcal{H}\text{-}h : \{0,1\}^* \times \{0,1\}^{iv} \rightarrow \{0,1\}^h$$

with $H = \mathsf{ACE}\text{-}\mathcal{H}\text{-}h(M, IV)$.

Note that IV and N refer to two different things. IV is for a hash function and is fixed, while N is for an AEAD algorithm and never repeated for a fixed key.

2.2 Recommended Parameter Set

In Table 2.1, we list the recommended parameter set for the AEAD and hash fuctionalities using the ACE permutation. The length of each parameter is given in bits and d denotes the amount of allowed data (including both AD and M) before a re-keying is required.

Table 2.1: Recommended parameter set for ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256

Functionality	Algorithm	r	k	n	t	$\log_2(d)$	h	iv
AEAD	ACE-$\mathcal{AE}$-128	64	128	128	128	124	-	-
Hash	ACE-$\mathcal{H}$-256	64	-	-	-	-	256	24

2.3 The ACE Permutation

ACE is an iterative permutation that takes a 320-bit state as an input and outputs an 320-bit state after iterating the step function ACE-step for $s = 16$ times (Figure 2.1). The nonlinear operation SB-64 is applied on even indexed words (i.e., A, C and E, see Figure 2.1) and hence the permutation name. We present the algorithmic description of ACE in Algorithm 1.

2.3.1 The nonlinear function SB-64

In ACE, we use unkeyed reduced-round Simeck block cipher [27] with block size 64 and $u = 8$ as the nonlinear operation, and denote it by SB-64. Below we provide the details of SB-64, henceforth referred to as Simeck box.

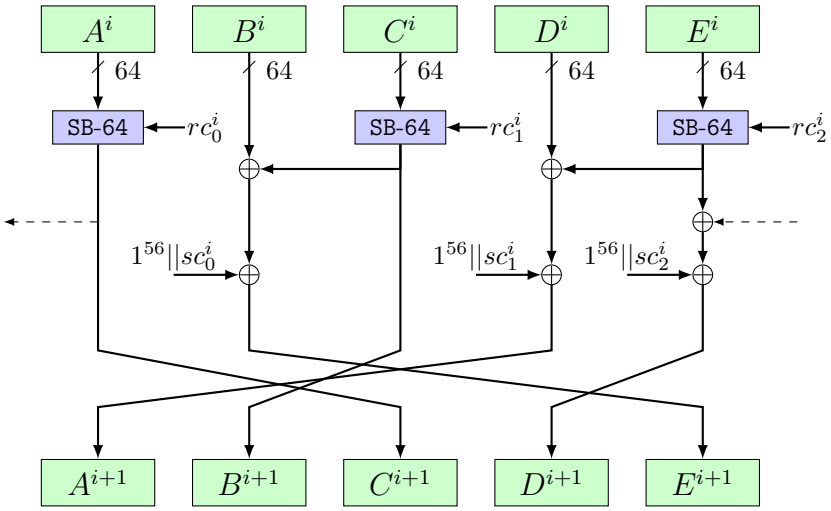

Figure 2.1: ACE-step

Definition 1 (SB-64: Simeck box [4]) *Let $rc = (q_7, q_6, \ldots, q_0)$ where $q_j \in \{0,1\}$ and $0 \leq j \leq 7$. A Simeck box is a permutation of a 64-bit input, constructed by iterating the Simeck-64 block cipher for 8 rounds with round constant addition $\gamma_j = 1^{31}||q_j$ in place of key addition.*

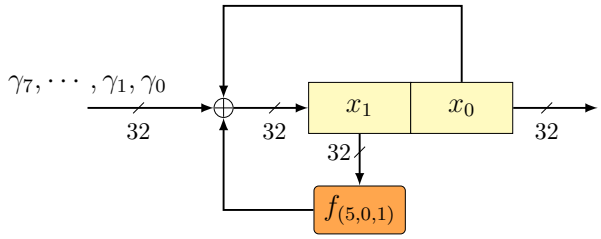

Figure 2.2: Simeck box (SB-64)

An illustrated description of the Simeck box is shown in Figure 2.2 and is given by

$$(x_9||x_8) \leftarrow \mathsf{SB\text{-}64}(x_1||x_0, rc)$$

where

$$x_j \leftarrow f_{(5,0,1)}(x_{j-1}) \oplus x_{j-2} \oplus \gamma_{j-2}, \ 2 \leq j \leq 9$$

and $f_{(5,0,1)} : \{0,1\}^{32} \rightarrow \{0,1\}^{32}$ is defined as

$$f_{(5,0,1)}(x) = (\mathsf{L}^5(x) \odot x) \oplus \mathsf{L}^1(x).$$

2.3.2 Round and step constants

The step function of ACE is parameterized by two sets of triplets (rc_0^i, rc_1^i, rc_2^i) and (sc_0^i, sc_1^i, sc_2^i) where each rc_j^i and sc_j^i is of length 8 bits and $j = 0, 1, 2$. We call them

Algorithm 1 ACE permutation

Input: $S^0 = A^0||B^0||C^0||D^0||E^0$
Output: $S^{16} = A^{16}||B^{16}||C^{16}||D^{16}||E^{16}$

for $i = 0$ to 15 **do**:
    $S^{i+1} \leftarrow \mathsf{ACE\text{-}step}(S^i)$
return S^{16}

Function $\mathsf{ACE\text{-}step}(S^i)$:
    $A^i \leftarrow \mathtt{SB\text{-}64}(A^i_1||A^i_0, rc^i_0)$
    $C^i \leftarrow \mathtt{SB\text{-}64}(C^i_1||C^i_0, rc^i_1)$
    $E^i \leftarrow \mathtt{SB\text{-}64}(E^i_1||E^i_0, rc^i_2)$
    $B^i \leftarrow B^i \oplus C^i \oplus (1^{56}||sc^i_0)$
    $D^i \leftarrow D^i \oplus E^i \oplus (1^{56}||sc^i_1)$
    $E^i \leftarrow E^i \oplus A^i \oplus (1^{56}||sc^i_2)$
    $A^{i+1} \leftarrow D^i$
    $B^{i+1} \leftarrow C^i$
    $C^{i+1} \leftarrow A^i$
    $D^{i+1} \leftarrow E^i$
    $E^{i+1} \leftarrow B^i$
    return $(A^{i+1}||B^{i+1}||C^{i+1}||D^{i+1}||E^{i+1})$

Function $\mathtt{SB\text{-}64}(x_1||x_0, rc)$:
    $rc = (q_7, q_6, \ldots, q_0)$
    for $j = 2$ to 9 **do**
        $x_j \leftarrow (\mathsf{L}^5(x_{j-1}) \odot x_{j-1}) \oplus \mathsf{L}^1(x_{j-1}) \oplus x_{j-2} \oplus (1^{31}||q_{j-2})$
    return $(x_9||x_8)$

round constants and step constants, respectively. As shown in Figure 2.1, the round constant triplet (rc^i_0, rc^i_1, rc^i_2) is used within the Simeck boxes while the step constant (sc^i_0, sc^i_1, sc^i_2) is XORed to the words B, D and E.

In Table 2.2 we list the hexadecimal values of the constants and show the procedure to generate these constants in Section 5.6.2.

2.4 AEAD Algorithm: ACE-$\mathcal{AE}$-128

In Algorithm 2, we present a high level overview of ACE-$\mathcal{AE}$-128. The encryption (ACE-$\mathcal{E}$) and decryption (ACE-$\mathcal{D}$) processes of ACE-$\mathcal{AE}$-128 are shown in Figure 2.3. In what follows, we first illustrate the position of rate and capacity bytes of the state, and the padding rule. We then describe each phase of ACE-$\mathcal{E}$ and ACE-$\mathcal{D}$.

Table 2.2: Round and step constants of ACE

Step i	**Round constants** (rc_0^i, rc_1^i, rc_2^i)	**Step constants** (sc_0^i, sc_1^i, sc_2^i)
0 - 3	(07, 53, 43), (0a, 5d, e4), (9b, 49, 5e), (e0, 7f, cc)	(50, 28, 14), (5c, ae, 57), (91, 48, 24), (8d, c6, 63)
4 - 7	(d1, be, 32), (1a, 1d, 4e), (22, 28, 75), (f7, 6c, 25)	(53, a9, 54), (60, 30, 18), (68, 34, 9a), (e1, 70, 38)
8 - 11	(62, 82, fd), (96, 47, f9), (71, 6b, 76), (aa, 88, a0)	(f6, 7b, bd), (9d, ce, 67), (40, 20, 10), (4f, 27, 13)
12 - 15	(2b, dc, b0), (e9, 8b, 09), (cf, 59, 1e), (b7, c6, ad)	(be, 5f, 2f), (5b, ad, d6), (e9, 74, ba), (7f, 3f, 1f)

Algorithm 2 ACE-$\mathcal{AE}$-128 algorithm

Authenticated encryption ACE-$\mathcal{E}(K, N, AD, M)$:
  $S \leftarrow$ Initialization(N, K)
  if $|AD| \neq 0$ **then:**
    $S \leftarrow$ Processing-Associated-Data(S, AD)
  $(S, C) \leftarrow$ Encyption(S, M)
  $T \leftarrow$ Finalization(S, K)
  return (C, T)

Initialization(N, K):
  $S \leftarrow$ load-$\mathcal{AE}(N, K)$
  $S \leftarrow$ ACE(S)
  for $i = 0$ to 1 **do:**
    $S \leftarrow (S_r \oplus K_i, S_c)$
    $S \leftarrow$ ACE(S)
  return S

Processing-Associated-Data(S, AD):
  $(AD_0||\cdots||AD_{\ell_{AD}-1}) \leftarrow \mathsf{pad}_r(AD)$
  for $i = 0$ to $\ell_{AD} - 1$ **do:**
    $S \leftarrow (S_r \oplus AD_i, S_c \oplus 0^{c-2}||01)$
    $S \leftarrow$ ACE(S)
  return S

Encryption(S, M):
  $(M_0||\cdots||M_{\ell_M-1}) \leftarrow \mathsf{pad}_r(M)$
  for $i = 0$ to $\ell_M - 1$ **do:**
    $C_i \leftarrow M_i \oplus S_r$
    $S \leftarrow (C_i, S_c \oplus 0^{c-2}||10)$
    $S \leftarrow$ ACE(S)
  $C_{\ell_M-1} \leftarrow$ trunc-msb$(C_{\ell_M-1}, |M| \bmod r)$
  $C \leftarrow (C_0, C_1, \ldots, C_{\ell_M-1})$
  return (S, C)

$\mathsf{pad}_r(X)$:
  $X \leftarrow X||10^{r-1-(|X| \bmod r)}$
  return X

trunc-lsb(X, n):
  return $(x_{r-n}, x_{r-n+1}, \ldots, x_{r-1})$

Verified decryption ACE-$\mathcal{D}(K, N, AD, C, T)$:
  $S \leftarrow$ Initialization(N, K)
  if $|AD| \neq 0$ **then:**
    $S \leftarrow$ Processing-Associated-Data(S, AD)
  $(S, M) \leftarrow$ Decyption(S, C)
  $T' \leftarrow$ Finalization(S, K)
  if $T' \neq T$ **then:**
    return $\perp$
  else:
    return M

Decryption(S, C):
  $(C_0||\cdots||C_{\ell_C-1}) \leftarrow \mathsf{pad}_r(C)$
  for $i = 0$ to $\ell_C - 2$ **do:**
    $M_i \leftarrow C_i \oplus S_r$
    $S \leftarrow (C_i, S_c \oplus 0^{c-2}||10)$
    $S \leftarrow$ ACE(S)
  $M_{\ell_C-1} \leftarrow S_r \oplus C_{\ell_C-1}$
  $C_{\ell_C-1} \leftarrow$ trunc-msb$(C_{\ell_C-1}, |C| \bmod r)||$trunc-lsb$(M_{\ell_C-1}, r-|C| \bmod r))$
  $M_{\ell_C-1} \leftarrow$ trunc-msb$(M_{\ell_C-1}, |C| \bmod r)$
  $M \leftarrow (M_0, M_1, \ldots, M_{\ell_C-1})$
  $S \leftarrow$ ACE$(C_{\ell_C-1}, S_c \oplus 0^{c-2}||10)$
  return (S, M)

Finalization(S, K):
  for $i = 0$ to 1 **do:**
    $S \leftarrow$ ACE$(S_r \oplus K_i, S_c)$
  $T \leftarrow$ tagextract(S)
  return T

trunc-msb(X, n):
  if $n = 0$ **then:**
    return ϕ
  else:
    return $(x_0, x_1, \ldots, x_{n-1})$

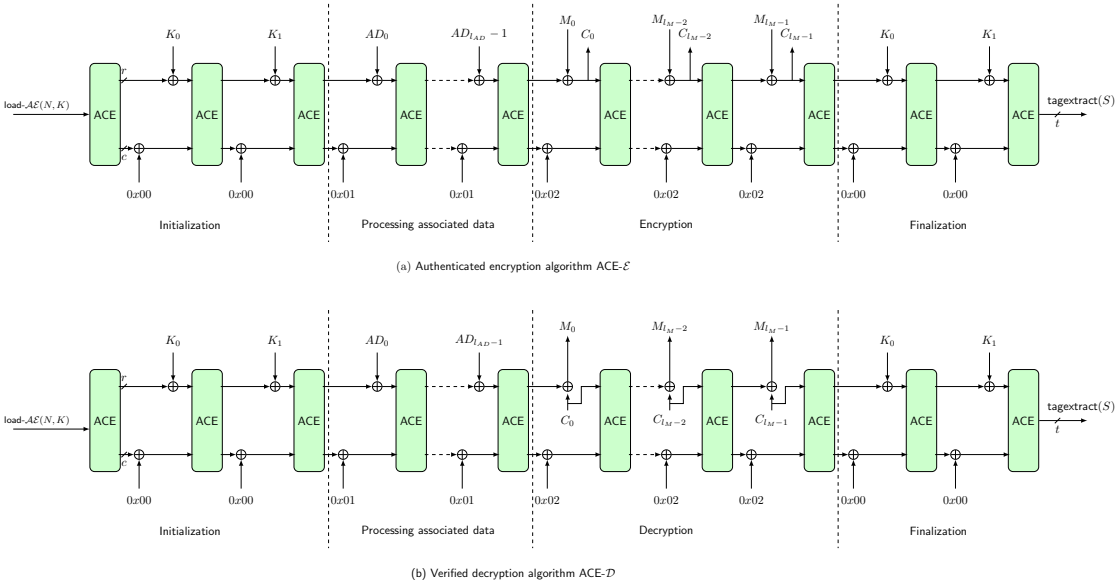

Figure 2.3: Schematic diagram of ACE-$\mathcal{AE}$-128 AEAD algorithm

2.4.1 Rate and capacity part of state

The following 8 bytes constitute the S_r part of state and are used for both absorbing and squeezing.

$$A[7], A[6], A[5], A[4], C[7], C[6], C[5], C[4]$$

The rationale of these byte positions is explained in Section 5.8. The remaining bytes form the S_c part of state.

2.4.2 Padding

Padding is necessary when the length of the processed data is not a multiple of the rate r value. Since the key size is a multiple of r, we get two key blocks K_0 and K_1, so no padding is needed. Afterwards, the padding rule (10^*), denoting a single 1 followed by the required number of 0's, is applied to the message M, so that its length after padding is a multiple of r. The resulting padded message is divided into ℓ_M r-bit blocks $M_0 \| \cdots \| M_{\ell_M - 1}$. A similar procedure is carried out on the associated data AD which results in ℓ_{AD} r-bit blocks $AD_0 \| \cdots \| AD_{\ell_{AD}-1}$. In the case where no associated data is present, no processing is necessary. We summarize the padding rules for the message and associated data below.

$$\begin{aligned} \mathsf{pad_r}(M) &\leftarrow M \| 1 \| 0^{r-1-(|M| \bmod r)} \\ \mathsf{pad_r}(AD) &\leftarrow \begin{cases} AD \| 1 \| 0^{r-1-(|AD| \bmod r)} & \text{if } |AD| > 0 \\ \phi & \text{if } |AD| = 0 \end{cases} \end{aligned}$$

Note that in case of AD or M whose length is a multiple of r, an additional r-bit padded block is appended to AD or M to distinguish between the processing of partial and complete blocks.

2.4.3 Loading key and nonce

The state is loaded byte-wise with a 128-bit nonce $N = N_0||N_1$ and 128-bit key $K = K_0||K_1$, and the remaining eight bytes are set to zero. All nonce bytes are divided and loaded in the words, B and E, in a descending byte order. The key is loaded in words A and C in the same manner. Word D is initialized by the zero bytes. Formally, the state is initialized as follows.

$$\begin{aligned}
A[7], A[6], \cdots, A[0] &\leftarrow K_0[7], K_0[6], \cdots, K_0[0] \\
C[7], C[6], \cdots, C[0] &\leftarrow K_1[7], K_1[6], \cdots, K_1[0] \\
B[7], B[6], \cdots, B[0] &\leftarrow N_0[7], N_0[6], \cdots, N_0[0] \\
E[7], E[6], \cdots, E[0] &\leftarrow N_1[7], N_1[6], \cdots, N_1[0] \\
D[7], D[6], \cdots, D[0] &\leftarrow 0x00, 0x00, \cdots, 0x00
\end{aligned}$$

We use load-$\mathcal{AE}(N, K)$ to denote the process of loading the state with nonce N and key K bytes in the positions described above.

2.4.4 Initialization

The goal of this phase is to initialize the state S with public nonce N and key K. The state is first loaded using load-$\mathcal{AE}(N, K)$ as described above. Afterwards, the permutation ACE is applied to the state, and the two key blocks are absorbed into the state with ACE applied each time. The initialization steps are described below.

$$\begin{aligned}
S &\leftarrow \mathsf{ACE}(\text{load-}\mathcal{AE}(N, K)) \\
S &\leftarrow \mathsf{ACE}(S_r \oplus K_0, S_c) \\
S &\leftarrow \mathsf{ACE}(S_r \oplus K_1, S_c)
\end{aligned}$$

2.4.5 Processing associated data

If there is associated data, each AD_i block, $i = 0, \ldots, \ell_{AD} - 1$ is XORed with the S_r part of the internal state S and one-bit domain separator is XORed to lsb of $E[0]$. Then, the ACE permutation is applied to the whole state.

$$S \leftarrow \mathsf{ACE}(S_r \oplus AD_i, S_c \oplus (0^{c-2}\|01)),\ i = 0, \ldots, \ell_{AD} - 1$$

This phase is defined in Algorithm 2.

2.4.6 Encryption

Similar to the processing of associated data, however, with a different domain separator, each message block M_i, $i = 0, \ldots, \ell_M - 1$ is XORed to the S_r part of the internal state

S resulting in the corresponding ciphertext C_i, which is extracted from the S_r part of the state. After the computation of each C_i, the whole internal state is permuted by applying ACE.

$$\begin{aligned} C_i &\leftarrow S_r \oplus M_i, \\ S &\leftarrow \mathsf{ACE}(C_i, S_c \oplus (0^{c-2} \| 10)),\ i = 0, \cdots, \ell_M - 1 \end{aligned}$$

To keep a minimal overhead, the last ciphertext block $C_{\ell_M - 1}$ is truncated so that its length is equal to that of the last unpadded message block. The details of encryption procedure is given in Algorithm 2.

2.4.7 Finalization

After the extraction of last ciphertext block and a single call of ACE, the domain separator is reset to $0x00$ indicating the start of the finalization phase. Afterwards, the 2 key blocks are absorbed into the state. Finally, the tag is extracted from the same byte positions that are used for loading the key The finalization steps are mentioned below and illustrated in Algorithm 2.

$$\begin{aligned} S &\leftarrow \mathsf{ACE}(S_r \oplus K_i, S_c),\ i = 0, 1 \\ T &\leftarrow \mathsf{tagextract}(S). \end{aligned}$$

The function $\mathsf{tagextract}(S)$ extracts the 128-bit tag $T = T_0 || T_1$ from the state bytes as follows.

$$\begin{aligned} T_0[7], T_0[6], \cdots, T_0[0] &\leftarrow A[7], A[6], \cdots, A[0] \\ T_1[7], T_1[6], \cdots, T_1[0] &\leftarrow C[7], C[6], \cdots, C[0] \end{aligned}$$

2.4.8 Decryption

The decryption procedure is symmetrical to the encryption procedure and illustrated in Algorithm 2.

2.5 Hash Algorithm: ACE-$\mathcal{H}$-256

The hash algorithm ACE-$\mathcal{H}$-256 takes as input a message M, and the standard initialization vector IV of length 24 bits, and then returns a 256-bit message digest H. The depiction of the ACE-$\mathcal{H}$-256 is shown in Figure 2.4 and illustrated in Algorithm 3. We now describe each phase of ACE-$\mathcal{H}$-256 in detail.

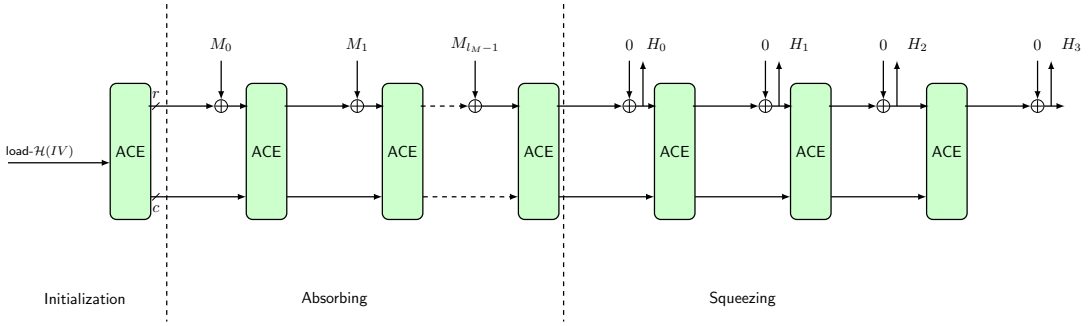

Figure 2.4: Hash algorithm ACE-$\mathcal{H}$-256

2.5.1 Message padding

The same padding rule (10^*) as is used in ACE-$\mathcal{AE}$-128 is applied to the input message M, where a single 1 followed by enough 0's is appended to it such that its length after padding is a multiple of r. We denote the padding rule by

$$\mathsf{pad}_r(M) = M \| 10^{r-1-(|M| \bmod r)}$$

The resulting padded message is then divided into ℓ_M r-bit blocks $M_0 \| \cdots \| M_{\ell_M - 1}$.

2.5.2 Loading initialization vector

The state is first initialized by $IV = h/2 \| r \| r'$, where r' denotes the number of bits squeezed per permutation call ($r = r' = 64$ for ACE-$\mathcal{H}$-256). Eight bits are used to encode each of the used $h/2$, r and r' sizes [19] and loaded in word B as follows.

$$B[7] \leftarrow 0x80$$
$$B[6] \leftarrow 0x40$$
$$B[5] \leftarrow 0x40$$

The remaining bytes are set to $0x00$. We denote this process by load-$\mathcal{H}(IV)$.

2.5.3 Initialization

The load-$\mathcal{H}(IV)$ procedure loads the state with the IV. Then a single call of ACE completes the initialization phase.

$$S \leftarrow \mathsf{ACE}(\text{load-}\mathcal{H}(IV))$$

2.5.4 Absorbing and squeezing

Each message block is absorbed by XORing it to the S_r part of the state (see Section 2.4.1), then the ACE permutation is applied. After absorbing all the message blocks, the h-bit output is extracted from the S_r part of the state r bits at a time followed by the application of the ACE permutation until a total of 4 extractions are completed.

Algorithm 3 ACE-$\mathcal{H}$-256 algorithm

ACE-$\mathcal{H}$-256(M, IV):
    $S \leftarrow$ Initialization(IV)
    $S \leftarrow$ Absorbing(S, M)
    $H \leftarrow$ Squeezing(S)
    return H

Initialization(IV):
    $S \leftarrow$ load-$\mathcal{H}(IV)$
    $S \leftarrow$ ACE(S)
    return S

pad$_r(M)$:
    $M \leftarrow M||10^{r-1-(|M| \bmod r)}$
    return M

Absorbing(S,M):
    $(M_0||\cdots||M_{\ell_M-1}) \leftarrow$ pad$_r(M)$
    for $i = 0$ to $\ell_M - 1$ **do:**
        $S \leftarrow$ ACE$(S_r \oplus M_i, S_c)$
    return S

Squeezing(S):
    for $i = 0$ to 2 **do:**
        $H_i \leftarrow S_r$
        $S \leftarrow$ ACE(S)
    $H_3 \leftarrow S_r$
    return $H_0||H_1||H_2||H_3$

Chapter 3

Security Claims

ACE is an all-in-one primitive and provides both authenticated encryption with associated data and hashing functionalities. The AEAD mode assumes a nonce respecting adversary and we do not claim security in the event of nonce reuse. If the verification procedure fails, the decrypted ciphertext and the new tag should not be given as output. Moreover, we claim no security for reduced-round versions of ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256. In summary, the security claims of ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256 are given in Table 3.1 and Table 3.2, respectively.

Note that the integrity security in Table 3.1 includes the integrity of plaintext, associated data and nonce.

Table 3.1: Security goals of ACE-$\mathcal{AE}$-128 (in bits)

Confidentiality	Integrity	Authenticity	Data limit
128	128	128	124

Table 3.2: Security goals of ACE-$\mathcal{H}$-256(in bits)

Collision	Preimage	Second preimage
128	192	128

Chapter 4

Security Analysis

In this chapter, we first analyze the security of the ACE permutation by assessing its behavior against various distinguishing attacks. We primarily focus on the diffusion behavior, expected upper bounds on the probabilities of differential and linear characteristics, algebraic properties and self-symmetry based distinguishers. Next, we present the security bounds of ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256, whose results directly follow the security proofs of sponge modes.

In our analysis, we denote the linear layer by π, i.e., π permutate the words of state. For example, if $\pi(0, 1, 2, 3, 4) = (3, 2, 0, 4, 1)$ then after applying π, the state $A||B||C||D||E$ is transformed to $D||C||A||E||B$. Moreover, by the component function f_j we refer to the Algebraic Normal Form (ANF) of the j-th bit.

4.1 Diffusion

To assess the diffusion behavior, we evaluate the minimum value of $u \times s$ such that each component function of the state after s steps depends on all the input state bits. We find that $u = 11$ gives full bit diffusion within a Simeck box. Since ACE has five words that are updated in each step, we note that s has to be at least 5. Accordingly, we search for the following values of $(u, s) \in \{(i, 5)|1 \leq i \leq 11\}$. Note that for $u = 8$ and $s = 5$, the number of linear layers satisfying the full bit diffusion property are 13, and $\pi = (3, 2, 0, 4, 1)$ is one among them.

Given that $(u, s) = (8, 16)$ and $\pi = (3, 2, 0, 4, 1)$ for ACE, we claim that meet/miss-in-the-middle distinguishers cannot cover more than ten steps, because ten steps guarantees full bit diffusion in both forward and backward directions.

4.2 Differential and Linear Cryptanalysis

To analyze the security of ACE w.r.t differential and linear distinguishers [16, 25], we model ACE using Mixed Integer Linear Programming (MILP) and bound the minimum number of active Simeck boxes (SB-64). We then provide expected bounds for the

maximum probabilities of differential (resp. linear) characteristics. Table 4.1 depicts the minimum number of active Simeck boxes for all linear layers.

4.2.1 Expected bounds on the maximum probabilities of differential and linear characteristics

Let $n_s(\pi)$ be the minimum number of active Simeck boxes in s steps for a linear layer π, and p denote the Maximum Differential Probability bound (MDP) for a u-round Simeck box in $\log_2(\cdot)$ scale. An in-depth analysis of values of p has been provided in [5] (cf. 4.2). We now choose u, s and π such that

- the upper bound on the maximum differential characteristic probability is less than 2^{-320}, i.e., $|n_s(\pi)p| > 320$.
- $u \times s$ is minimum and s is at least three times the number of steps required for full bit diffusion. This implies $s \geq 15$ for ACE.

For $(u, s) = (8, 16)$ and $\pi = (3, 2, 0, 4, 1)$, $n_s(\pi) = 21$ and $p = -15.8$. Thus, $|21 \times (-15.8)| \approx 331.8 > 320$ and maximum differential characteristic probability bound is $2^{-331.8}$. The maxiumum squared correlation of a linear characteristic is computed analogously using $\gamma = -15.6$ and equals $2^{-327.6}$, where γ is the maximum square correlation of a 8-round Simeck box (cf 4.2.2 [5]).

4.3 Algebraic Properties

In this section, we provide bounds for the algebraic degree of ACE and evaluate its security against integral distinguishers. We use the bit based division property [26, 8] to compute the algebraic degree. We find that the algebraic degree of a 8-round Simeck box is 36. Note that the algebraic degree (after 8 rounds) of all component functions from $f_0 - f_{31}$ is 36 while it is 27 for the component functions $f_{32} - f_{63}$. Thus, to evaluate the algebraic degree of ACE it is enough to find bounds for algebraic degree of the component functions $f_0, f_{32}, f_{64}, f_{96}, f_{128}, f_{160}, f_{192}, f_{224}, f_{256}$ and f_{288}. Table 4.2 provides bounds of the algebraic degree for the above component functions.

Note that since the number of words in ACE is odd, due to slow diffusion the algebraic degree is 63 and 62 for the component functions f_{64} and f_{96} after 2 steps, respectively. A similar trend can be seen for the component functions f_{256} and f_{288}. This non-uniformity in degree continues till step five, after which the degree is stabilized to 304-313 due to full bit diffusion (Section 4.1). We expect that the degree reaches 319 in six steps.

Integral distinguishers [21]. To search for the longest length integral distinguisher, we set a bit of the input state as constant (0) and the rest are set to active (1). We then evaluate the algebraic degree at the s-th step of each component function in terms

Table 4.1: Minimum number of active Simeck boxes for s-step ACE

Linear layer	step s															
π	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
(1, 0, 3, 4, 2)	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
(1, 0, 4, 2, 3)	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
(1, 2, 0, 4, 3)	0	1	2	3	4	6	8	8	9	10	11	12	14	16	16	17
(1, 2, 3, 4, 0)	0	1	2	3	4	6	7	8	9	10	11	12	13	14	15	16
(1, 2, 4, 0, 3)	0	1	1	2	3	5	7	8	9	9	10	11	13	15	16	17
(1, 3, 0, 4, 2)	0	0	1	2	4	4	5	7	9	12	13	14	15	16	17	19
(1, 3, 4, 0, 2)	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5
(1, 3, 4, 2, 0)	0	0	1	2	3	4	5	6	7	9	11	12	12	13	14	15
(1, 4, 0, 2, 3)	0	1	2	3	4	6	8	8	9	10	11	12	14	16	16	17
(1, 4, 3, 0, 2)	0	1	1	2	4	5	6	7	9	10	11	12	14	15	16	18
(1, 4, 3, 2, 0)	0	1	2	3	5	6	7	9	11	12	13	14	15	17	18	19
(2, 0, 1, 4, 3)	0	1	2	3	4	6	6	8	9	10	11	12	14	15	16	17
(2, 0, 3, 4, 1)	0	1	2	3	4	5	7	8	9	10	11	12	13	14	15	16
(2, 0, 4, 1, 3)	0	0	1	2	3	3	4	5	7	9	10	11	12	12	13	14
(2, 3, 0, 4, 1)	0	0	1	2	3	5	7	9	10	10	11	12	13	15	17	19
(2, 3, 1, 4, 0)	0	0	1	3	4	6	7	8	8	9	11	12	13	14	15	16
(2, 3, 4, 0, 1)	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(2, 4, 0, 1, 3)	0	0	1	3	4	5	7	9	10	10	11	13	14	15	17	19
(2, 4, 1, 0, 3)	0	1	2	3	4	5	7	8	9	10	11	12	13	15	16	17
(2, 4, 3, 0, 1)	0	1	2	3	5	5	6	7	8	10	10	11	12	13	15	15
(2, 4, 3, 1, 0)	0	0	1	2	4	6	7	8	9	10	11	12	13	14	15	16
(3, 0, 1, 4, 2)	0	1	1	2	4	6	8	8	10	10	11	13	15	16	17	19
(3, 0, 4, 1, 2)	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5
(3, 0, 4, 2, 1)	0	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(3, 2, 0, 4, 1)	0	1	2	3	5	7	8	9	11	12	13	15	16	17	**19**	**21**
(3, 2, 1, 4, 0)	0	1	2	2	3	4	4	5	6	6	7	8	8	9	10	10
(3, 2, 4, 0, 1)	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
(3, 2, 4, 1, 0)	0	0	1	2	3	6	8	9	9	10	11	12	15	16	18	18
(3, 4, 0, 1, 2)	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
(3, 4, 0, 2, 1)	0	1	2	3	4	4	5	6	7	8	8	9	10	11	12	12
(3, 4, 1, 0, 2)	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
(3, 4, 1, 2, 0)	0	1	2	3	5	7	8	9	11	12	13	15	16	17	19	21
(4, 0, 1, 2, 3)	0	1	2	3	4	5	7	8	10	12	13	13	14	15	17	19
(4, 0, 3, 1, 2)	0	0	1	2	3	3	4	5	7	9	10	11	12	12	13	14
(4, 0, 3, 2, 1)	0	1	2	3	4	5	6	8	10	11	12	13	14	15	16	17
(4, 2, 0, 1, 3)	0	0	1	2	4	6	7	9	10	11	12	13	14	15	17	19
(4, 2, 1, 0, 3)	0	1	2	2	3	4	4	5	6	6	7	8	8	9	10	10
(4, 2, 3, 0, 1)	0	1	2	3	5	6	8	9	10	11	12	14	16	17	18	19
(4, 2, 3, 1, 0)	0	0	1	1	2	3	4	4	4	5	5	6	7	8	8	8
(4, 3, 0, 1, 2)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(4, 3, 0, 2, 1)	0	0	1	2	3	5	6	7	10	10	11	12	13	15	16	17
(4, 3, 1, 0, 2)	0	0	1	2	4	5	6	7	9	10	11	12	13	14	15	17
(4, 3, 1, 2, 0)	0	0	1	1	2	3	4	4	4	5	5	6	7	8	8	8

Table 4.2: Bounds on the algebraic degree of ACE

	Component function									
steps (s)	f_0	f_{32}	f_{64}	f_{96}	f_{128}	f_{160}	f_{192}	f_{224}	f_{256}	f_{288}
1	36	27	36	27	36	27	36	27	36	27
2	92	83	63	62	92	83	92	83	63	62
3	126	125	119	117-120	239-247	235-245	236-249	233-248	119	118-120
4	240-247	238-246	241-248	242-247	306-312	303-311	304-313	304-311	241-248	241-247

of the involved active bits. If the algebraic degree equals the number of active bits then the bit is unknown (i.e., XOR sum of the component function is unpredictable), otherwise, it is balanced in which case the XOR sum is always zero.

In Table 4.3, we list the integral distinguishers of ACE. Note that the positions of constant bits are chosen based on the degree of the Simeck box.

Table 4.3: Integral distinguishers of ACE

Steps s	Input division property	Balanced bits
8	$1^{32}\|\|0\|\|1^{287}$	64-127, 256-319
	$1^{96}\|\|0\|\|1^{223}$	None
	$1^{160}\|\|0\|\|1^{159}$	None
	$1^{224}\|\|0\|\|1^{95}$	64-127, 256-319
	$1^{288}\|\|0\|\|1^{31}$	None

4.4 Self Symmetry-based Distinguishers

A cryptographic permutation is vulnerable to attacks such as rotational distinguishers, slide distinguishers [17] and invariant subspace attack [23] which exploit the symmetric properties of a round function. For example, in ACE the nonlinear Simeck box is rotational invariant if constants are not added at each round. Thus, a proper choice of round constants is required to mitigate the above attacks.

ACE employs an 7-bit LFSR to generate round and step constants (see Section 5.6.2). Below we mention properties of the constants which ensure that each step function of ACE distinct.

- For $0 \leq i \leq 15$, $sc_0^i \neq sc_1^i \neq sc_2^i$
- For $0 \leq i \leq 15$, $(rc_0^i, rc_1^i, rc_2^i) \neq (sc_0^i, sc_1^i, sc_2^i)$
- For $0 \leq i, j \leq 15$ and $i \neq j$, $(rc_0^i, rc_1^i, rc_2^i) \neq (rc_0^j, rc_1^j, rc_2^j)$
- For $0 \leq i, j \leq 15$ and $i \neq j$, $(sc_0^i, sc_1^i, sc_2^i) \neq (sc_0^j, sc_1^j, sc_2^j)$.

4.5 Security of ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256

The security proofs of modes based on the sponge construction relies on the indistinguishability of the underlying permutation from a random one [10, 14, 13, 20]. In previous sections, we have shown that there are no distinguishers for 16 steps of ACE. Thus, the security bounds of sponge modes are applicable to both ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256.

ACE-$\mathcal{AE}$-128 security. We assume a nonce-respecting adversary, i.e, for a fixed K, the nonce N is never repeated. Then considering a data limit of 2^d, k-bit security is achieved if $c \geq k + d + 1$ and $d \ll c/2$ [13]. The parameter set of ACE-$\mathcal{AE}$-128 (see Table 2.1) with actual effective capacity 254 (2 bits are lost for domain separation) satisfies this condition, and hence ACE-$\mathcal{AE}$-128 provides 128-bit security for confidentiality, integrity and authenticity.

Note that we could use $r = 192$, $d = 64$ and obtain the same level of security [20]. However, this would require an additional 128 XORs and cannot meet our objective to achieve both AEAD and hash functionalities using the same hardware circuit.

ACE-$\mathcal{H}$-256 security. For a sponge based hash with $b = r + c$ and h-bit message digest, the generic security bounds [12, 19] are given by:

- **Collision:** $\min(2^{h/2}, 2^{c/2})$
- **Preimage:** $\min(2^{\min(h,b)}, \max(2^{\min(h,b)-r}, 2^{c/2}))$
- **Second-preimage:** $\min(2^{h}, 2^{c/2})$

Accordingly, ACE-$\mathcal{H}$-256 provides 128, 192 and 128-bit securities for collision, preimage and second preimage, respectively.

Chapter 5

Design Rationale

In this chapter, we provide the rationale for our design choices and justify the design principles of each component of ACE, ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256.

5.1 Choice of the Mode: sLiSCP Sponge Mode

Our adopted mode is a variation of the sponge duplex construction. Sponge constructions are very diversified in terms of the offered security level. Particularly, it is proven that the sponge and its single pass duplex mode offer a $2^{c/2}$ bound against generic attacks [11, 14] which provides a lower bound on the width of the underlying permutation. However, for authenticated encryption (AE), a security level of 2^{c-d} is proven when the number of queries is upper bounded by 2^d [15]. When restricting the data complexity to a maximum of 2^d queries with $d \ll c/2$, one can reduce the capacity and increase the rate for a better throughput with the same security level. Jovanovic *et.al.* [20] have shown that sponge based AE achieve higher security bound, i.e, $\min\{2^{b/2}, 2^c, 2^k\}$ compared to [13]. However, we are concerned with the former bound, as shown in Section 4.5.

In sponge keyed encryption modes, nonce reuse enables the encryption of two different messages with the same key stream, which undermines the privacy of the primitive. More precisely, the sponge duplex authenticated encryption mode requires the uniqueness of a nonce when encrypting different messages with the same key because the ability of the attacker to acquire multiple combinations of input and output differences leaks information about the inner state bits, which may lead to the reconstruction of the full state [14, 9]. Nonce reuse in duplex constructions reveals the XOR difference between the first two plaintexts by XORing their corresponding ciphertexts. On the other hand, a nonce reuse differential attack may be exploited if the attacker is able to inject a difference in the plaintext and cancel it out by another difference after the permutation application. However, such an attack depends on the probability of the best differential characteristic and the number of rounds of the underlying permutation. Accordingly, if such a permutation offers enough resistance to differential cryptanaly-

sis, the feasibility of nonce reuse differential attacks is minimal. The condition on the differential behavior of the underlying permutation is also important when considering resynchronization attacks, where related nonces are to be used. For that reason, even if nonce reuse is not permitted, the underlying permutation used in the initialization stage should be strong enough to mitigate differential attacks.

Given the above results, the sLiSCP sponge mode [4] realizes the following objectives:

- The flexibility to adapt the same circuitry to provide both authenticated encryption and hashing functionalities, as we adopt a unified round function for all functionalities.
- High key agility, which fits the lightweight requirements, because all keyed modes require no key scheduling.
- Simplicity, as there is no need to implement a decryption algorithm, because the same encryption algorithm is used for decryption.
- Both plaintext and ciphertext blocks are generated online without the need to process the whole input message and encrypted material first.
- Initialization and finalization phases that make key recovery hard even if the internal state is recovered and also renders universal forgery with the knowledge of the internal state unattainable.
- More hardware efficient initialization and finalization stages where the state is initialized with the key which is again absorbed in the rate part afterward.
- Domain separators run for all rounds of all stages and offer uniformity across different stages. We change the domain separators with each new transition and not before because we found that it leads to a more efficient hardware implementations. Such mechanism has been shown to be secure in [20].

5.2 ACE State Size

Our main objective is to choose b (state size) that provides 128-bit security for both hash and AEAD, i.e., 256-bit hash output and 128-bit key and tag. For b-bit state with $b = r + c$, r-bit rate and c-bit hash output, generic attacks with $2^{c/2}$ permutation queries exist [11]. Thus, to satisfy the security requirements of hash, c should be 256 which implies $b \geq 257$. The immediate choices are $b = 288, 320$ and 384. In ACE, we choose $b = 320$ as it provides the best trade-off among hardware and software requirements, security and efficiency. With this choice of b, ACE can have implementations in a wide range of platforms. We discard the other state sizes for the following reasons.

- Considering the lightweight applications, 384-bit state is too heavy in hardware.

- 288 is not a multiple of 64, hence, we can not efficiently use inbuilt 64-bit CPU instructions for software implementation.

5.3 ACE Step Function

The step function of the ACE permutation can be seen as a generalized five 64-bit word sLiSCP-light [5] structure. Since we aim to build a 320-bit permutation, we could have used a 4-word sLiSCP-light with 80-bit Simeck boxes. However, we found that it is not practical to evaluate most of the cryptographic properties for the resulting permutation using Simeck boxes with sizes > 64, and that our 80-bit based software implementation is not efficient. Consequently, we decided to use a 5-word sLiSCP-light with 3 Simeck boxes and wrap around the linear mixing between words A and E. We also decided to XOR SB-64(A) with SB-64(E) and not E to avoid the need for an extra temporary 64-bit register to store the initial value of E while intermediate results of the iterated SB-64 function are stored in E.

5.4 Nonlinear Layer: Simeck box (SB-64)

The Simeck box is an unkeyed independently parameterized variant of the round function of the Simon round function [7]. Moreover, it has set a new record in terms of hardware efficiency and performance on almost all platforms [27]. In what follows, we list the reasons that motivated our adoption of Simeck boxes as the nonlinear function of ACE permutation.

- Simeck has a hardware friendly round function that consists of simple bitwise XOR, AND and cyclic shift operations. Moreover, the hardware cost grows linearly with input size.
- It is practical to evaluate the SB-64 maximum (expected) differential probability and maximum (expected) linear squared correlation which are $2^{-15.8}$ and $2^{-15.6}$, respectively. Accordingly, we can provide an expected bounds against differential and linear cryptanalysis.
- SB-64 has an algebraic degree of 36 and the output component functions $f_0 - f_{31}$ (resp. $f_{32} - f_{63}$) depend on 61 (resp. 55) input state bits, which enables us to provides guarantees gainst algebraic and diffusion-based attacks.
- Each Simeck box is independently parameterized by the associated set of round constants, which suggests that the actual security against differential and linear cryptanalysis is better than the reported bounds.

5.5 Linear Layer: $\pi = (3, 2, 0, 4, 1)$

The choice of a linear layer is crucial for the proper mixing among the subblocks, which in turn affects the differential and algebraic properties. Out of 5! possible permutations of the words, 44 do not exhibit fixed points. Moreover, we found that iterating such permutations for multiple rounds achieves different differential and algebraic bounds. Accordingly, we searched their space to find the ones that offer the best diffusion and result in the minimum number of active Simeck boxes in the smallest number of steps. We found that only two permutations, $\pi = (3, 2, 0, 4, 1)$, and $\pi' = (3, 4, 1, 2, 0)$ achieve these conditions. More precisely, using either π or π', ACE reaches full bit diffusion in 5 steps and has 21 active Simeck boxes (see Table 4.1). Accordingly, we picked π as our linear layer.

5.6 Round and Step Constants

5.6.1 Rationale

We use the following set of constants to mitigate the self-symmetry distinguishers.

- **Three 8-bit unique step constants** (sc_0^i, sc_1^i, sc_2^i). The 3-tuple constant value is unique across all steps, hence it destroys any symmetry between the steps of the permutation. Accordingly, we mitigate slide distinguishers [17]. We also require that for any given step i, $sc_0^i \neq sc_1^i \neq sc_2^i$ in order to destroy any symmetry between word shuffles.

- **Three 8-bit unique round constants** (rc_0^i, rc_1^i, rc_2^i). One bit of each round constant is XORed with the state of the Simeck box in each round to destroy the preservation of any rotational properties. Moreover, we append 31 '1' bits to each one bit constant, which results in a lot of inversions, and accordingly breaks the propagation of the rotational property in one step.

Our choice of the LFSR polynomial to generate the constants ensures that each tuple of such constants does not repeat due to the periodicity of the 8-tuple sequence constructed from the decimated m-sequence of period 127.

5.6.2 Generation of round and step constants

We use an LFSR of length 7 with the feedback polynomial $x^7 + x + 1$ to generate the round and step constants of ACE. To construct these constants, the same LFSR is run in a 3-way parallel configuration, as illustrated in Figure 5.1. Let $\underline{a}$ denote the sequence generated by the initial state $(a_0, a_1, \ldots, a_6)$ of the LFSR without parallelization. The parallel version of this LFSR outputs three sequences, all of them using decimation exponent 3. Instead of one XOR gate feedback for the non-parallel implementation, three XOR gates are needed to compute three feedback values.

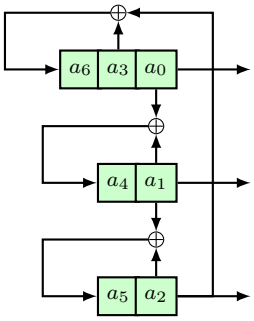

Figure 5.1: LFSR for generating ACE constants.

Figure 5.2 shows the same LFSR as Figure 5.1, but annotated with sequence elements at the moment when the last three bits for the round constants are available. The round constants are produced by the sequence elements a_{24i+21}, a_{24i+22} and a_{24i+23} in every clock cycle as follows.

$$
\begin{aligned}
rc_0^i &= a_{24i+21} \| a_{24i+18} \| a_{24i+15} \| a_{24i+12} \| \; a_{24i+9} \| a_{24i+6} \| a_{24i+3} \| a_{24i+0} \\
rc_1^i &= a_{24i+22} \| a_{24i+19} \| a_{24i+16} \| a_{24i+13} \| a_{24i+10} \| a_{24i+7} \| a_{24i+4} \| a_{24i+1} \\
rc_2^i &= a_{24i+23} \| a_{24i+20} \| a_{24i+17} \| a_{24i+14} \| a_{24i+11} \| a_{24i+8} \| a_{24i+5} \| a_{24i+2}
\end{aligned}
$$

where

- rc_0^i corresponds to the sequence $\underline{a}$ with decimation 3
- rc_1^i corresponds to the sequence $\underline{a}$ shifted by 1, then decimated by 3
- rc_2^i corresponds to the sequence $\underline{a}$ shifted by 2, then decimated by 3

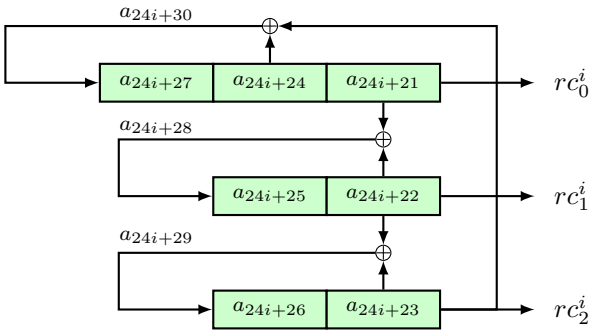

Figure 5.2: Schematic of the 3-way parallel LFSR for generation of the constants

In every 8^{th} clock cycle, the step constants are needed in addition to round constants. The computation of step constants does not need any extra circuitry, but rather uses the three feedback values a_{24i+28}, a_{24i+29} and a_{24i+30} together with all 7 state bits, annotated

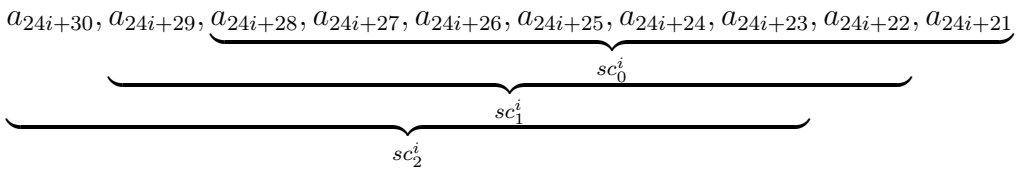

Figure 5.3: Three 8-bit step constants, generated from 10 consecutive sequence elements

in Figure 5.2. Figure 5.3 shows how the 10 consecutive sequence elements are used to generate step constants.
The step constants are given by:

$$sc_0^i = a_{24i+28} \| a_{24i+27} \| a_{24i+26} \| a_{24i+25} \| a_{24i+24} \| a_{24i+23} \| a_{24i+22} \| a_{24i+21}$$
$$sc_1^i = a_{24i+29} \| a_{24i+28} \| a_{24i+27} \| a_{24i+26} \| a_{24i+25} \| a_{24i+24} \| a_{24i+23} \| a_{24i+22}$$
$$sc_2^i = a_{24i+30} \| a_{24i+29} \| a_{24i+28} \| a_{24i+27} \| a_{24i+26} \| a_{24i+25} \| a_{24i+24} \| a_{24i+23}$$

We provide an example of how to obtain hex values of constants from LFSR sequence in Appendix C.

5.7 Number of Rounds and Steps

Our rationale for choosing the number of rounds u and number of steps s of ACE is based on achieving the best trade-off between security and efficiency. By security and efficiency, we mean the value of (u, s) for which ACE is indistiguishable from a random permutation and $u \times s$ is minimum. We now justify the choice of $(u, s) = (8, 16)$ for ACE.

Diffusion. Our first criteria is that s should be at least $3 \times m$ where m is the number of #steps needed to achieve full bit diffusion in the state. This choice is inspired from [18] and directly adds a 33% security margin against meet/miss-in-the-middle distinguishers, as in $2m$ steps full bit diffusion is achieved in both forward and backward directions. Hence, $m = 5 \implies u \geq 4$ and $s \geq 15$ (c.f. Section 4.1). However, we found that we cannot choose $u = 4, \ldots, 7$ because we also aim to achieve good resistance against differential and linear cryptanalysis, and having a smaller number of rounds results in a weaker Simeck box.

Maximum expected differential characteristic probability (MEDCP). Our second criteria is to push the MEDCP value of ACE to below 2^{-320}. This value depends on the MEDCP of a u-round Simeck box and the number of such active boxes in s steps (denote by n_s). We have $n_{15} = 19$ and $n_{16} = 21$ (see Table 4.1).

Table 5.1 depicts that $(u, s) \in \{(8, 15), (8, 16), (9, 15), (9, 16)\}$. However, if we consider the differential effect, then the differential probability is $2^{-15.8}$ when $u = 8$. An indepth analysis of such effect has been provided in [5] where the CryptoSMT tool [22]

Table 5.1: Optimal differential characteristic probability p for u-round Simeck box and the corresponding MEDCP of ACE for $s = 15, 16$.

u	4	5	6	7	8	9
$\log_2(p)$	-6	-8	-12	-14	-18	-20
$n_{15} \times \log_2(p)$	-114	-152	-228	-266	-342	-380
$n_{16} \times \log_2(p)$	-126	-168	-252	-294	-378	-420

is used to obtain the optimal differential characteristics and corresponding probabilities. Accordingly, we have:

$$n_{15} \times -15.8 = 19 \times -15.8 = -300.2 > -320$$

$$n_{16} \times -15.8 = 21 \times -15.8 = -331.8 < -320.$$

Thus, we ignore $(u, s) = (8, 15)$ and choose $(u, s) = (8, 16)$. The other two choices are discarded from the efficiency perspective as $u \times s = 135$ (resp. 144) when $(u, s) = (9, 15)$ (resp. (9,16)) compared to 128 iterations when $(u, s) = (8, 15)$.

5.8 Choice of Rate Positions

We have followed a similar strategy in choosing the rate position as the one that has been used in sLiSCP [4, 6]. More precisely, we absorb message blocks in words A and C. Such rate positions allow the input bits to be processed by the Simeck boxes as soon as possible so we achieve faster diffusion. Also, our choice forces any injected differences to activate Simeck boxes in the first step which also enhances ACE's resistance to differential and linear cryptanalysis. This observation has also been confirmed by a third party cryptanalysis of sLiSCP [24].

5.9 Statement

The authors declare that there are no hidden weaknesses in the ACE permutation, ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256.

Chapter 6

Hardware Design and Analysis

In this chapter, we describe the hardware implementation of ACE_module, which is a single module that supports all three functionalities: authenticated encryption, verified decryption, and hashing using the same hardware circuit.

6.1 Hardware Design Principles

In this section, we describe the design principles and assumptions that we followed while implementing ACE_module.

1. **Multi-functionality module.** The system should support all three operations, namely authenticated encryption, authenticated decryption, and hashing, in a single module, because lightweight applications generally cannot afford the extra area for separate modules. As a result, the area for the system will be greater compared to a single-function module.

2. **Single input/output ports.** In small devices, ports can be expensive, and optimizing the number of ports may require additional multiplexers and control circuitry. To ensure that we are not biasing our design in favour of the system and at the expense of the environment, the key, nonce, associated data, and message all use a single data-input port. Similarly, the output ciphertext, tag, and hash all use a single output port.

3. **Valid-bit protocol and stalling capability.** The environment may take an arbitrarily long time to produce any piece of data. For example, a small microprocessor could require multiple clock cycles to read data from memory and write it to the system's input port. We use a single-phase valid bit protocol, where each input or output data signal is paired with a valid bit to denote when the data is valid. The receiving entity must capture the data in a single clock cycle, which is a simple and widely applicable protocol. The system shall wait in an idle state, while signalling the environment that it is ready to receive.

4. **Use a "pure register-transfer-level" implementation style.** In particular, use only registers, not latches; multiplexers, not tri-state buffers; and synchronous, not asynchronous reset.

6.2 Interface and Top-level Module

In Figure 6.1, we depict the block diagram of the top-level ACE_module and the description of each interface signal is given in Table 6.1.

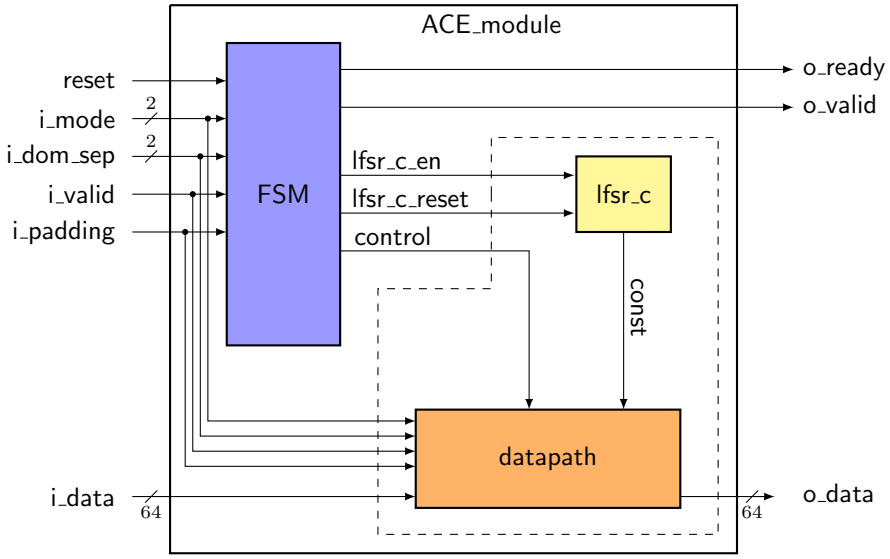

Figure 6.1: Top-level module and interface

Table 6.1: Interface signals

Input signal	Meaning
reset	resets the state machine
i_mode	mode of operation
i_dom_sep	domain separator
i_padding	the last block is padded
i_data	input data
i_valid	valid data on i_data

Output signal	Meaning
o_ready	hardware is ready
o_data	output data
o_valid	valid data on o_data

The ACE-$\mathcal{AE}$-128 mode can perform two operations: authenticated encryption (ACE-$\mathcal{E}$) and verified decryption (ACE-$\mathcal{D}$). The ACE-$\mathcal{H}$-256 mode has two phases: absorbing and squeezing, both of which have the same domain separator. We use the i_mode input signal (see Table 6.2) to distinguish between the operations or phases.

The environment separates the associated data and the message/ciphertext, and performs their padding if necessary, as specified in Sections 2.4 and 2.5. The control

Table 6.2: ACE_module modes of operation

i_mode			
(1)	(0)	Mode	Operation or phase
0	0	ACE-$\mathcal{E}$	Encryption
0	1	ACE-$\mathcal{D}$	Decryption
1	0	ACE-$\mathcal{H}$-256	Absorb
1	1	ACE-$\mathcal{H}$-256	Squeeze

input i_pad is used to indicate that the last i_data block is padded. No internal counters for the number of processed blocks are needed. The environment uses the domain separators to indicate whether the input data for ACE-$\mathcal{AE}$-128 is the key, associated data or plaintext/ciphertext. For ACE-$\mathcal{H}$-256, the phase change is indicated by the change of the i_mode(0) signal, as shown in Table 6.2.

6.3 The ACE_module

In this section, we illustrate the implementation details of ACE_module. More specifically, we provide the details of ACE datapath, ACE FSM and lfsr_c (orange, blue and yellow colored boxes, respectively in Figure 6.1). We also provide the estimated and implementation results for the areas.

6.3.1 ACE datapath

Figure 6.2 shows the schematic for the ACE datapath. The top of the figure depicts the five 64-bit registers A, B, C, D and E followed by the hardware components required for absorbing, replacing and driving the outputs. The rest of the Figure 6.2 shows one step of the ACE permutation, annotated on the left. The three parallel SB-64 modules are shown with a shaded grey box. The rounds and steps always use the same hardware, but in different clock cycles, which forces the use of multiplexers inside the ACE permutation.

In Table 6.3, we provide both the estimate based on the CMOS 65 nm ASIC library and actual hardware area of the ACE permutation. For the CMOS 65 nm we use an estimate of 3.75 GE for a 1-bit register and 2.00 GE for a 2-input XOR gate. The row "other XORs" contains the XOR gates needed for masking B, D and E with outputs of the three SB-64 modules, and for the addition of step constants. The estimate in Table 6.3 does not count the multiplexers that are required to update the registers A, B, C, D, E with Simeck round outputs or after one step of ACE.

Hardware circuitory for en/de-cryption and hash. Figure 6.2 depicts the ACE permutation, but also shows the circuitry needed to utilize the ACE permutation to have a specific mode. The absorbing hardware part is also shown in Figure 6.2. Apart

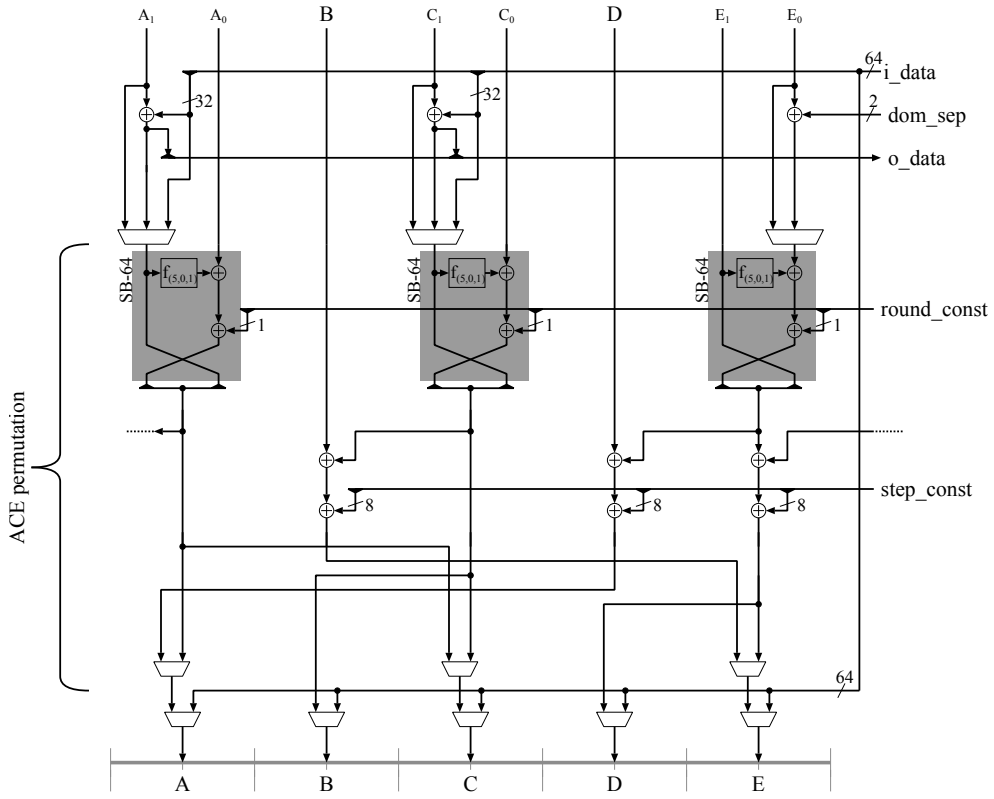

Figure 6.2: The ACE_module datapath

Table 6.3: ACE permutation hardware area estimate and implementation results

Component	**Estimate per unit** [GE]	**Count**	**Estimate per component** [GE]
State registers	3.75	320	1200
SB-64	154†	3	462
Other XORs	2.00	$3 \times (64 + 8)$	432
ACE permutation - Total estimated area			2092
pre-PAR CMOS 65 nm implementation area results			
ACE permutation			2716
ACE_module			4268

† pre-PAR implementation results

from the output generation, this behaviour is the same for ACE-$\mathcal{AE}$-128 initialization, processing associated data, encryption, finalization, and for the ACE-$\mathcal{H}$-256 absorbing and squeezing phase. For ACE-$\mathcal{AE}$-128 decryption, extra multiplexers are required on the inputs to the ACE permutation, hence the 3:1 multiplexers before the start of the ACE permutation.

6.3.2 ACE FSM and lfsr_c

The FSM (blue box Figure 6.1) controls the ACE permutation itself, i.e., it provides the control signals for the multiplexers based on a particular operation or phase (see Table 6.2). It uses an internal 8-bit counter to keep track of rounds and steps of the ACE permutation, and to set the control signals for the register updates, e.g., update a register with the round result or update with the step result. The FSM also sets the control signals for the environment, such as o_valid and o_ready. The last row in Table 6.3 shows the implementation results for the complete ACE_module, including the FSM and the lfsr_c used for generation of the round and step constants (see Section 5.6.2).

6.4 Hardware Implementation Results

In this section, we provide the ASIC CMOS and FPGA implementation results of ACE and its modes. We first give the details of the used synthesis and simulation tools and then present the performance results.

6.4.1 Hardware tools configuration

Below we provide the configuration details of synthesis and simulation tools and libraries for both ASIC and FPGA implementations.

	Synthesis and simulation tools and libraries for the ASIC implementation
Logic synthesis	Synopsys Design Compiler vN-2017.09
Physical synthesis	Cadence Encounter 2014.13-s036_1
Simulation	Mentor Graphics QuestaSim 10.5c
ASIC cell library	65 nm STMicroelectronics CORE65LPLVT, 1.25V, 40C
	Synthesis tools for the FPGA implementation
Logic synthesis	Mentor Graphics Precision 64-bit 2016.1.1.28 (for Intel/Altera), ISE (for Xilinx)
Physical synthesis	Altera Quartus Prime 15.1.0 SJ Standard Edition (for Intel/Altera), ISE (for Xilinx)

6.4.2 Performance results

In Tables 6.4 and 6.5, we present the performance results of the ACE permutation and the ACE_module. Note that the ACE_module is a single module that performs all three functionalities: authenticated encryption, verified decryption and hashing.

Table 6.4: ASIC implementation results

Module	**Area** [GE]	**Frequency** [MHz]	**Throughput** [Mbps]
ACE permutation	2716	2500	n/a
ACE_module	4268	952	476

Table 6.5: FPGA implementation results

Module	**Frequency** [MHz]	**# of slices**	**# of FFs**	**# of LUTs**
Xilinx Spartan 3 (xc3s200-5ft256)				
ACE permutation	181	215	327	381
ACE_module	68	727	353	1410
Xilinx Spartan 6 (xc6slx9-3ftg256)				
ACE permutation	306	127	327	378
ACE_module	123	429	365	1272

Module	**Frequency** [MHz]	**# of LC**	**# of FFs**	**# of LUTs**
Intel/Altera Stratix IV (EP4SGX70HF35M3)				
ACE permutation	128	327	327	296
ACE_module	51	781†	354	781

† ACE_module includes ALTSYNCRAM block memory with 35 bits.

Chapter 7

Efficiency Analysis in Software

The ACE permutation is designed to be efficient on a wide range of resource constrianed devices, which requires the primitive to be efficient in hardware as well as software. Even for lightweight applications, a server communicating with such devices needs to perform the encryption/decryption, and hashing operations at a high speed. We assess the efficiency of the ACE permutation and its modes on two different software platforms: high-performance CPUs and microcontrollers. For the high-performance CPU implementation, we consider a bit-sliced implementation of ACE using SIMD instruction sets.

7.1 Software: High-performance CPU

We implement ACE in the bit-slice fashion using SIMD instruction sets which provides resistance against cache-timing attacks and allows to execute multiple independent ACE instances in parallel. We consider SSE and AVX instruction sets in Intel processors where the SSE and AVX instruction sets, support 128-bit and 256-bit SIMD registers, known as XMM and YMM, respectively. Algorithm 4 depicts the detailed steps of our implementation. In our implementation, packing and unpacking of data are two important tasks, which are performed at the beginning and at the end of the execution of the permutation and also during the execution of the permutation.

Basic idea. The key idea for our software implementation of the ACE permutation is to split the state of the permutation among different registers for performing similar types of operations (e.g., SB-64). For instance, when eight parallel instances of ACE are evaluated using YMM registers, we pack data for SB-64 operation into six YMM registers and other blocks are stored in four other YMM registers. This allows us to perform the same operations in different registers to achieve efficiency in the implementation. Below we explain the bit-slice implementation details of ACE for YMM registers. The details for the SSE implementation using XMM registers are similar, and so are omitted.

Packing and unpacking for ACE. There are two different types of packing and unpacking operations in our implementation: 1) one pair is performed at the beginning and end of the permutation execution; and 2) the other one is performed at the beginning and end of the SB-64 layer in each step. We start by describing the first one. For the software implementation, we denote an ACE state by $S_i = s_0^i s_1^i s_2^i s_3^i s_4^i s_5^i s_6^i s_7^i s_8^i s_9^i$ where each s_j^i is a 32-bit word, $0 \leq i \leq 7$ and $0 \leq j \leq 9$. First, the eight independent states $S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7$ of ACE are loaded into ten 256-bit registers as follows.

$$
\begin{array}{ll}
R_0 \leftarrow s_7^0 s_6^0 s_5^0 s_4^0 s_3^0 s_2^0 s_1^0 s_0^0; & R_4 \leftarrow s_7^1 s_6^1 s_5^1 s_4^1 s_3^1 s_2^1 s_1^1 s_0^1 \\
R_1 \leftarrow s_7^2 s_6^2 s_5^2 s_4^2 s_3^2 s_2^2 s_1^2 s_0^2; & R_5 \leftarrow s_7^3 s_6^3 s_5^3 s_4^3 s_3^3 s_2^3 s_1^3 s_0^3 \\
R_2 \leftarrow s_7^4 s_6^4 s_5^4 s_4^4 s_3^4 s_2^4 s_1^4 s_0^4; & R_6 \leftarrow s_7^5 s_6^5 s_5^5 s_4^5 s_3^5 s_2^5 s_1^5 s_0^5 \\
R_3 \leftarrow s_7^6 s_6^6 s_5^6 s_4^6 s_3^6 s_2^6 s_1^6 s_0^6; & R_7 \leftarrow s_7^7 s_6^7 s_5^7 s_4^7 s_3^7 s_2^7 s_1^7 s_0^7 \\
R_8 \leftarrow s_9^3 s_8^3 s_9^2 s_8^2 s_9^1 s_8^1 s_9^0 s_8^0; & R_9 \leftarrow s_9^7 s_8^7 s_9^6 s_8^6 s_9^5 s_8^5 s_9^4 s_8^4
\end{array}
$$

Then the packing operation is defined as

$$
\begin{array}{l}
\mathsf{PACK}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9): \\
\begin{array}{ll}
R_0 \leftarrow s_5^1 s_4^1 s_1^1 s_0^1 s_5^0 s_4^0 s_1^0 s_0^0; & R_4 \leftarrow s_7^1 s_6^1 s_3^1 s_2^1 s_7^0 s_6^0 s_3^0 s_2^0 \\
R_1 \leftarrow s_5^3 s_4^3 s_1^3 s_0^3 s_5^2 s_4^2 s_1^2 s_0^2; & R_5 \leftarrow s_7^3 s_6^3 s_3^3 s_2^3 s_7^2 s_6^2 s_3^2 s_2^2 \\
R_2 \leftarrow s_5^5 s_4^5 s_1^5 s_0^5 s_5^4 s_4^4 s_1^4 s_0^4; & R_6 \leftarrow s_7^5 s_6^5 s_3^5 s_2^5 s_7^4 s_6^4 s_3^4 s_2^4 \\
R_3 \leftarrow s_5^7 s_4^7 s_1^7 s_0^7 s_5^6 s_4^6 s_1^6 s_0^6; & R_7 \leftarrow s_7^7 s_6^7 s_3^7 s_2^7 s_7^6 s_6^6 s_3^6 s_2^6 \\
R_8 \leftarrow s_9^3 s_8^3 s_9^2 s_8^2 s_9^1 s_8^1 s_9^0 s_8^0; & R_9 \leftarrow s_9^7 s_8^7 s_9^6 s_8^6 s_9^5 s_8^5 s_9^4 s_8^4
\end{array}
\end{array}
$$

where the SB-64 operation is performed on R_0, R_1, R_2, R_3, R_8, and R_9.

The unpacking operation, denoted by UNPACK(), is the inverse of the packing operation, which we omit here. Both operations are implemented using `vpermd` and `vperm2i128`, `vpunpcklqdq` and `vpunpckhqdq` instructions. Assume that we wish to apply the SB-64 operation on disjoint 64 bits (i.e., $a_{2i+1}a_{2i}$ or $b_{2i+1}b_{2i}$) in the registers $A = a_7a_6a_5a_4a_3a_2a_1a_0$ and $B = b_7b_6b_5b_4b_3b_2b_1b_0$. As SB-64 adopts the Feistel structure, the data in A and B are regrouped for the homogeneity of operations in SB-64. For this, we need the second pair of packing and unpacking operations for the SB-64 layer, which is given by

$$
\begin{array}{ll}
\mathsf{PACK\_SB\text{-}64}(A, B): & \mathsf{UNPACK\_SB\text{-}64}(A, B): \\
A \leftarrow b_6b_4b_2b_0a_6a_4a_2a_0; & A \leftarrow b_3a_3b_2a_2b_1a_1b_0a_0; \\
B \leftarrow b_7b_5b_3b_1a_7a_5a_3a_3 & B \leftarrow b_7a_7b_6a_6b_5a_5b_4a_4.
\end{array}
$$

ROAX operation. We create an instruction for one round of execution of SB-64,

denoted by ROAX, which is given by

$$\begin{aligned}
&\mathsf{ROAX}(A, B, q_1, q_2) : \\
&\mathsf{tmp} \leftarrow A; \;\; C \leftarrow \texttt{0xffffffe}; \\
&A \leftarrow (\mathsf{L}^5(A) \odot A) \oplus \mathsf{L}^1(A); \\
&A \leftarrow A \oplus B \oplus (C \oplus q_1, C \oplus q_2, \cdots, C \oplus q_1, C \oplus q_2); \\
&B \leftarrow \mathsf{tmp};
\end{aligned}$$

where A and B are either a XMM or YMM register, $\mathsf{L}^5(A)$ (resp. $\mathsf{L}^1(A)$) denotes the left cyclic shift by 5 (resp. 1) on every a_i in A, which is implemented using `vpslld` and `vpsrld` instructions.

Swapblock operation. With $R_0, R_1, \cdots, R_9$ as input, the swap block operation, denoted as SWAPBLKS, corresponding to $\pi = (3, 2, 0, 4, 1)$ is given by

$$\begin{aligned}
&\mathsf{SWAPBLKS}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) : \\
&R_0 \leftarrow s_1^1 s_0^1 s_7^1 s_6^1 s_1^0 s_0^0 s_7^0 s_6^0; \quad R_4 \leftarrow s_9^1 s_8^1 s_5^1 s_4^1 s_9^0 s_8^0 s_5^0 s_4^0 \\
&R_1 \leftarrow s_1^3 s_0^3 s_7^3 s_6^3 s_1^2 s_0^2 s_7^2 s_6^2; \quad R_5 \leftarrow s_9^3 s_8^3 s_5^3 s_4^3 s_9^2 s_8^2 s_5^2 s_4^2 \\
&R_2 \leftarrow s_1^5 s_0^5 s_7^5 s_6^5 s_1^4 s_0^4 s_7^4 s_6^4; \quad R_6 \leftarrow s_9^5 s_8^5 s_5^5 s_4^5 s_9^4 s_8^4 s_5^4 s_4^4 \\
&R_3 \leftarrow s_1^7 s_0^7 s_7^7 s_6^7 s_1^6 s_0^6 s_7^6 s_6^6; \quad R_7 \leftarrow s_9^7 s_8^7 s_5^7 s_4^7 s_9^6 s_8^6 s_5^6 s_4^6 \\
&R_8 \leftarrow s_3^3 s_2^3 s_3^2 s_2^2 s_3^1 s_2^1 s_3^0 s_2^0; \quad R_9 \leftarrow s_3^7 s_2^7 s_3^6 s_2^6 s_3^5 s_2^5 s_3^4 s_2^4.
\end{aligned}$$

The execution of the eight parallel instances of the ACE permutation is summarized in Algorithm 4.

Benchmarking

We implement the ACE permutation and ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256 modes in C using SSE2 and AVX2 instruction sets and measure their performances on two different Intel processors: Skylake and Haswell. The codes were compiled using gcc 5.4.0 on 64-bit machines with the compiler flags `-O2 -funroll-all-loops -march=native`. For both implementations, we evaluate eight parallel instances and compute the throughput of the permutation and its modes. Table 7.1 presents the performance results in cycles per byte for both implementations where the message digest is computed for 1024 bits and encryption is also done for 1024 bits and the associated data length is set to 128 bits. In our implementation, we include the costs for all packing and unpacking operations. The best speed achieved is 9.97 cycles/byte for ACE, using the AVX2 implementation on Skylake.

Algorithm 4 Eight parallel instances of the ACE permutation

Input: $(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9)$
Output: $(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9)$

$(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) \leftarrow \mathsf{PACK}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9)$;
for $i = 0$ to 15 **do**:
    $R_0, R_1 \leftarrow \mathsf{PACK\_SB\text{-}64}(R_0, R_1)$;
    $R_2, R_3 \leftarrow \mathsf{PACK\_SB\text{-}64}(R_2, R_3)$;
    $R_8, R_9 \leftarrow \mathsf{PACK\_SB\text{-}64}(R_8, R_9)$;

    for $j = 0$ to 7 **do**:
        $R_0, R_1 \leftarrow \mathsf{ROAX}(R_0, R_1, rc_0^i[j], rc_1^i[j])$; ▷ $rc_0^i[j]$: j-th lsb of rc_0^i
        $R_2, R_3 \leftarrow \mathsf{ROAX}(R_2, R_3, rc_0^i[j], rc_1^i[j])$;
        $R_8, R_9 \leftarrow \mathsf{ROAX}(R_8, R_9, rc_2^i[j], rc_2^i[j])$;
    end for
    $R_0, R_1 \leftarrow \mathsf{UNPACK\_SB\text{-}64}(R_0, R_1)$;
    $R_2, R_3 \leftarrow \mathsf{UNPACK\_SB\text{-}64}(R_2, R_3)$;
    $R_8, R_9 \leftarrow \mathsf{UNPACK\_SB\text{-}64}(R_8, R_9)$;
    $C \leftarrow$ `0xffffff00`; $D \leftarrow$ `0xffffffff`;
    $\mathsf{tmp0} \leftarrow (D, C \oplus sc_0^i, D, C \oplus sc_1^i, D, C \oplus sc_0^i, D, C \oplus sc_1^i)$
    $\mathsf{tmp1} \leftarrow (D, C \oplus sc_2^i, D, C \oplus sc_2^i, D, C \oplus sc_2^i, D, C \oplus sc_2^i)$
    $R_4 \leftarrow R_4 \oplus \mathsf{tmp0}$; $R_5 \leftarrow R_5 \oplus \mathsf{tmp0}$;
    $R_6 \leftarrow R_6 \oplus \mathsf{tmp0}$; $R_7 \leftarrow R_7 \oplus \mathsf{tmp0}$;
    $R_8 \leftarrow R_8 \oplus \mathsf{tmp1}$; $R_9 \leftarrow R_9 \oplus \mathsf{tmp1}$;
    $(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) \leftarrow \mathsf{SWAPBLKS}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9)$;
end for
$(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) \leftarrow \mathsf{UNPACK}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9)$;
return $(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9)$;

7.2 Software: Microcontroller

We implement the ACE permutation and ACE-$\mathcal{AE}$-128 on two distinct microcontroller platforms. For ACE-$\mathcal{AE}$-128, we implement only encryption, as decryption is the same as encryption, except updating the rate with ciphertext. Our codes are written in assembly language to achieve optimal performance. We choose: 1) MSP430F2370, a 16-bit microcontroller from Texas Instruments with 2.3 Kbytes of programmable flash memory, 128 Bytes of RAM, and 12 general purpose registers of 16 bits, and 2) ARM Cortex M3 LM3S9D96, a 32-bit microcontroller with 524.3 Kbytes of programmable flash memory, 131 Kbytes of RAM, and 13 general purpose registers of 32 bits. We focus on four key performance measures, namely throughput (Kbps), code size (Kbytes), energy (nJ), and RAM (Kbytes) consumptions.

For ACE-$\mathcal{AE}$-128, the scheme is instantiated with a random 128-bit key and 128-bit nonce. Note that the throughput of the modes decreases compared to the permutation as the messages are processed on 64-bit blocks and $(5 + \ell)$(resp. $(4 + \ell)$) executions of the permutation are needed to evaluate the AE (resp. hash) mode where ℓ is the number of the processed data in blocks including padding if needed. Table 7.2 presents the performance of the ACE permutation and its modes.

Table 7.1: Benchmarking the results for the ACE permutation and its AE and Hash modes.

Primitive	**Speed** [cpb]	**Instruction Set**	**CPU Name Spec.**
ACE	15.66	SSE2	Skylake
	9.97	AVX2	`Intel i7-6700`
	16.96	SSE2	Haswell
	10.56	AVX2	`Intel i7-4790`
ACE-$\mathcal{AE}$-128	110.29	SSE2	Skylake
	68.53	AVX2	`Intel i7-6700`
	128.66	SSE2	Haswell
	89.10	AVX2	`Intel i7-4790`
ACE-$\mathcal{H}$-256	95.65	SSE2	Skylake
	58.81	AVX2	`Intel i7-6700`
	108.15	SSE2	Haswell
	66.12	AVX2	`Intel i7-4790`

Table 7.2: Performance of ACE on microcontrollers at a clock frequency of 16 MHz

Platform	Primitive	$\#AD$ blocks	$\#M$ blocks	Memory (bytes)		#cycles	Throughput	Energy/bit
				SRAM	Flash		[Kbps]	[nJ]
16-bit MSP430F2370	ACE permutation	-	-	304	1456	69440	73.73	225
	ACE-$\mathcal{AE}$-128	0	16	330	1740	1445059	11.34	1461
	ACE-$\mathcal{AE}$-128	2	16	330	1786	1582892	10.35	1600
	ACE-$\mathcal{H}$-256	-	2	330	1682	413056	4.96	3340
	ACE-$\mathcal{H}$-256	-	16	330	1684	1375672	11.91	1390
32-bit Cortex M3 LM3S9D96	ACE permutation	-	-	523	1598	13003	393.76	846
	ACE-$\mathcal{AE}$-128	0	16	559	1790	269341	60.83	5479
	ACE-$\mathcal{AE}$-128	2	16	559	1858	294988	55.54	6001
	ACE-$\mathcal{H}$-256	-	2	559	1822	77114	26.56	12550
	ACE-$\mathcal{H}$-256	-	16	559	1822	256524	63.87	5218

Acknowledgment

The submitters would like to thank Nusa Zidaric and Marat Sattarov for their work on the hardware implementation and Yunjie Yi for the microcontroller implementation.

Bibliography

[1] Gurobi: MILP optimizer. http://www.gurobi.com/.

[2] AlTawy, R., Gong, G., He, M., Jha, A., Mandal, K., Nandi, M., and Rohit, R. SpoC: Submission to NIST-LWC.

[3] AlTawy, R., Gong, G., He, M., Mandal, K., and Rohit, R. SPIX: An authenticated cipher. Submission to NIST-LWC.

[4] AlTawy, R., Rohit, R., He, M., Mandal, K., Yang, G., and Gong, G. sLiSCP: Simeck-based Permutations for Lightweight Sponge Cryptographic Primitives. In *SAC* (2017), C. Adams and J. Camenisch, Eds., Springer, pp. 129–150.

[5] Altawy, R., Rohit, R., He, M., Mandal, K., Yang, G., and Gong, G. Sliscp-light: Towards hardware optimized sponge-specific cryptographic permutations. *ACM Transactions on Embedded Computing Systems (TECS) 17*, 4 (2018), 81.

[6] Altawy, R., Rohit, R., He, M., Mandal, K., Yang, G., and Gong, G. Towards a cryptographic minimal design: The sLiSCP family of permutations. *IEEE Transactions on Computers 67*, 9 (2018), 1341–1358.

[7] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. http://eprint.iacr.org/2013/404.

[8] Bernstein, D. J., Kölbl, S., Lucks, S., Massolino, P. M. C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.-X., Todo, Y., and Viguier, B. Gimli: a cross-platform permutation, 2017.

[9] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. Caesar submission: Ketje v2, 2014. http://ketje.noekeon.org/Ketjev2-doc2.0.pdf.

[10] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. Sponge functions. In *ECRYPT hash workshop* (2007), vol. 2007.

[11] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. On the indifferentiability of the sponge construction. In *EUROCRYPT* (2008), N. Smart, Ed., Springer, pp. 181–197.

[12] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. Keccak specifications. submission to NIST (Round 2), 2009.

[13] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. On the security of the keyed sponge construction. In *Symmetric Key Encryption Workshop* (2011).

[14] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *SAC* (2012), A. Miri and S. Vaudenay, Eds., Springer, pp. 320–337.

[15] Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. Permutation-based encryption, authentication and authenticated encryption. *DIAC* (2012).

[16] Biham, E., and Shamir, A. Differential cryptanalysis of DES-like cryptosystems. *Journal of CRYPTOLOGY 4*, 1 (1991), 3–72.

[17] Biryukov, A., and Wagner, D. Slide attacks. In *FSE* (1999), L. Knudsen, Ed., Springer, pp. 245–259.

[18] Gueron, S., and Mouha, N. Simpira v2: A family of efficient permutations using the aes round function. In *ASIACRYPT* (2016), J. H. Cheon and T. Takagi, Eds., Springer, pp. 95–125.

[19] Guo, J., Peyrin, T., and Poschmann, A. The photon family of lightweight hash functions. In *CRYPTO* (2011), P. Rogaway, Ed., Springer, pp. 222–239.

[20] Jovanovic, P., Luykx, A., and Mennink, B. Beyond $2^{c/2}$ security in sponge-based authenticated encryption modes. In *ASIACRYPT* (2014), P. Sarkar and T. Iwata, Eds., Springe, pp. 85–104.

[21] Knudsen, L., and Wagner, D. Integral cryptanalysis. In *FSE* (2002), J. Daemen and V. Rijmen, Eds., vol. 2365 of *LNCS*, Springer, pp. 112–127.

[22] Kölbl, S., Leander, G., and Tiessen, T. Observations on the Simon block cipher family. In *CRYPTO* (2015), R. Gennaro and M. Robshaw, Eds., Springer, pp. 161–185.

[23] Leander, G., Abdelraheem, M. A., AlKhzaimi, H., and Zenner, E. A cryptanalysis of printcipher: The invariant subspace attack. In *CRYPTO* (2011), P. Rogaway, Ed., Springer, pp. 206–221.

[24] Liu, Y., Sasaki, Y., Song, L., and Wang, G. Cryptanalysis of reduced sliscp permutation in sponge-hash and duplex-AE modes. In *SAC* (2018), C. Cid and J. Michael J. Jacobson, Eds., vol. 11349, Springer, pp. 92–114.

[25] Matsui, M., and Yamagishi, A. A new method for known plaintext attack of FEAL cipher. In *Workshop on the Theory and Application of of Cryptographic Techniques* (1992), Springer, pp. 81–91.

[26] Todo, Y., and Morii, M. Bit-based division property and application to simon family. In *FSE* (2016), Springer, pp. 357–377.

[27] Yang, G., Zhu, B., Suder, V., Aagaard, M. D., and Gong, G. The simeck family of lightweight block ciphers. In *CHES* (2015), T. Güneysu and H. Handschuh, Eds., Springer, pp. 307–329.

Appendix A

Other NIST-LWC Submissions

In Table A.1, we list our other NIST-LWC submissions whose underlying permutation adopts a similar design as sLiSCP-light [5] family of permutations. ACE is an all in one primitive that utilizes a generalized version of sLiSCP-light with state size 320-bit and a different linear layer to offer both hashing and authenticated encryption functionalities. Spix adopts sLiSCP-light-256 in a monkey duplex to offer higher throughput than generic Sponge-based AE schemes. Spoc is an authenticated cipher that enables higher bound on the underlying state size to offer same security as other generic AE schemes, thus allowing larger rate size. Spoc adopts sLiSCP-light-192 and sLiSCP-light-256 to enable different performance and hence different target applications. In Table A.1, the submissions are classified based on their functionalities, mode of operation parameters and hardware area in CMOS 65 nm ASIC.

Table A.1: Submissions with sLiSCP-light like permutations

Algorithm	Permutation	Functionality	Parameters (in bits)			Mode of operation	Area
			State	Rate	Security		[GE]
ACE-$\mathcal{AE}$-128 and ACE-$\mathcal{H}$-256	ACE	AEAD & Hash	320	64	128	Unified sLiSCP sponge	4286
Spix [3]	sLiSCP-light-256	AEAD	256	64	128	Monkey Duplex	2611
Spoc-64 [2]	sLiSCP-light-192	AEAD	192	64	128	Spoc	2329
Spoc-128 [2]	sLiSCP-light-256	AEAD	256	128	128	Spoc	3020

Appendix B

Test Vectors

B.1 Simeck Sbox

Test vector for Simeck sbox with input = 0000000000000000 and $rc = 0x07$.

Round	State
0	0000000000000000
1	FFFFFFFF00000000
2	FFFFFFFFFFFFFFFF
3	00000000FFFFFFFF
4	0000000100000000
5	FFFFFFFC00000001
6	FFFFFF9AFFFFFFFC
7	00000C2DFFFFFF9A
8	00001C1E00000C2D

B.2 ACE Permutation

Test vector for ACE with all zero state.

Table B.1: Test vector for ACE permutation

Step	State
0	0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
1	FFFFC7CAFFFFE776 00003C9E00001C3A 00001C1E00000C2D FFFFDBD4FFFFEB67 FFFFC361FFFFE36A
2	A008C151D9C4D9F0 25ACD9A124884ECC 86A59002FBFA4BCD D9528A87DDC179A3 DA531AC0DB77ADAA
3	446BFB17FEAC5A5F 683F3428F9654513 68637D8AD2D9A691 F55A0C1AF1B48501 B26C12762212F44E
4	5453CAE4EC2F4442 1229976DFAB4E931 62A32D3D4BF8A3A4 C3AAEBC356636242 85E95CBAFC2E53AF
5	598621C1B175FD21 2D4271827840D029 7067E7DBE7730CF1 EA4B2DD90065936F C09419107D0BC64B
6	F082FCB61529AA71 7BCFD42DFA4C3E52 2D5F0057B73ACCE3 3796D138A276F5D5 A9725A507DF3111B
7	FEFB14A90FFC6647 9F5776716E158260 8E92C0A5D6800B6F 47FF05347B0A9853 1B675DA36BA6435A
8	89F30BAF3692897B C4FDA6EBEC5A67C9 14F005716C4AFC2A DAFC0BEA21D2ED4A A4552F657DB01A48
9	4C23136992E442A3 011A48EC0CB5FB43 66FE8CDB3B4199E5 F0219458887736CA 3A1811F81F10637C
10	2610F06C195D5056 17AE4FD7BD09471B FFBDD5A7EAB46BCE 298CB1937B9E0DFB E94BF8C44E4343C5
11	C0DC927D4DB070E3 CF7763937E89CB5C 15839159A987CDA1 FCD3B2B79FA9B089 2726D3BB3C7F7307
12	EB0021C196A1BD2A 0430040EFF58D77D BC9CEE20225F9C0F AB4F7D562B579198 34B898627E2EE36E
13	6665A40D97687B80 5930C806DDEBC73A 61B46748C3F87266 AC9EBE137FC7980E A2FF33F7DD4CEFF9
14	AB3B18A05461271D 8E535FE0229BC4A8 3A17D3E8D0C0DEBE 3DB2755BFB6661D6 289C6819008FFCC9
15	9FE7E5EA42C1167A 637EA3CF659E1667 A7C2AFF4D71079A3 05973F456EB70EC1 12D203D0B8FA2D26
16	5C93691AD5060935 DC19CE947EAD550D AC12BEE1A64B670E F516E8BE1DFA60DA 409892A4E4CCBC15

B.3 ACE-$\mathcal{AE}$-128

Key	00111122335588DD 00111122335588DD
Nonce	111122335588DD00 111122335588DD00
Associated data	1122335588DD0011 1122335588DD00
Plaintext	335588DD00111122 335588DD001111
Ciphertext	F9362385DC213A07 CEFEF38C34CEFF
Tag	AE85154F0242F0E4 0F9ECA3FE696D7C6

B.4 ACE-$\mathcal{H}$-256

Message 335588DD00111122 335588DD001111

Hash 1676336AB5C04A1D 9225FB283172A757 A0637A6523127B83 EFC3E990BABBD2E6

Appendix C

Constants: Sequence to Hex Conversion

In this section, we show how to obtain hex values of the constants for $i = 0$. Note that the LFSR is reset to initial all-one state $(1, \ldots, 1)$ at the beginning of each ACE permutation. The example is captured in Table C.1. First column in the table represents the clock cycle, which corresponds to the round within the step. The next column is showing the current LFSR state in this clock cycle. The bits are written in the same pattern as states in Figure 5.2, without showing the three feedback bits. The third column is showing 10 sequence bits, composed of the three feedback bits, followed by the state bits: the top row shows the subsequence with correct indices and the bottom row their respective values. The last three bits in every row are used directly as round constant in every clock cycle. The 10-bit subsequence from clock cycle 7 is used directly as the step constant and interpreted as shown in Figure 5.3.
The HEX values for step $i = 0$, listed in Table 2.2 are obtained as follows:

- from the last three columns of Table C.1 for the round constants
- from the last row of Table C.1 for the step constants

as follows:

$$\begin{array}{lllll} rc_0^0 & = & 00000111 & = & 0x07 \\ rc_1^0 & = & 01010011 & = & 0x53 \\ rc_2^i & = & 01000011 & = & 0x43 \end{array} \qquad \begin{array}{lllll} sc_0^0 & = & 01010000 & = & 0x50 \\ sc_1^0 & = & 00101000 & = & 0x28 \\ sc_2^i & = & 00010100 & = & 0x14 \end{array}$$

Table C.1: Generation of round and step constants for $i = 0$

clk. cycle	(current) LFSR state			(current) subsequence bits										
	1	1	1	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	
0		1	1											
		1	1	0	0	0	1	1	1	1	1	1	1	
	0	1	1	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	
1		0	1											
		0	1	0	0	0	0	0	0	1	1	1	1	
	0	0	1	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	
2		0	0											
		0	0	0	0	1	0	0	0	0	0	0	1	
	0	0	0	a_{18}	a_{17}	a_{16}	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	
3		1	0											
		0	0	0	0	0	0	0	1	0	0	0	0	
	0	0	0	a_{21}	a_{20}	a_{19}	a_{18}	a_{17}	a_{16}	a_{15}	a_{14}	a_{13}	a_{12}	
4		0	1											
		0	0	0	1	1	0	0	0	0	0	1	0	
	0	0	0	a_{24}	a_{23}	a_{22}	a_{21}	a_{20}	a_{19}	a_{18}	a_{17}	a_{16}	a_{15}	
5		1	0											
		1	0	0	0	0	0	1	1	0	0	0	0	
	0	0	0	a_{27}	a_{26}	a_{25}	a_{24}	a_{23}	a_{22}	a_{21}	a_{20}	a_{19}	a_{18}	
6		0	1											
		0	1	1	0	1	0	0	0	0	1	1	0	
	1	0	0	a_{30}	a_{29}	a_{28}	a_{27}	a_{26}	a_{25}	a_{24}	a_{23}	a_{22}	a_{21}	
7		1	0											
		0	0	0	0	0	1	0	1	0	0	0	0	$\leftarrow$ sc_2^0, sc_1^0, sc_0^0
											$\uparrow$ rc_2^0	$\uparrow$ rc_1^0	$\uparrow$ rc_0^0	