ACE: An Authenticated Encryption and Hash Algorithm
Submission to the NIST LWC Competition

SUBMITTERS/DESIGNERS:
Mark Aagaard, Riham AlTawy\(^1\), 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

https://uwaterloo.ca/communications-security-lab/lwc/ace

September 27, 2019

\(^1\)Currently with Department of Electrical and Computer Engineering, University of Victoria, 3800 Finnerty Rd, Victoria, BC, V8P 5C2, CANADA
# Contents

1 Introduction .................................................. 7  
  1.1 Notations .................................................. 8  
  1.2 Outline ..................................................... 9  

2 Specification ............................................... 10  
  2.1 Parameters ................................................ 10  
    2.1.1 ACE AEAD algorithm ..................................... 10  
    2.1.2 ACE Hash algorithm .................................... 11  
  2.2 Recommended Parameter Set ................................ 11  
  2.3 The ACE Permutation ....................................... 11  
    2.3.1 The nonlinear function SB-64 .......................... 11  
    2.3.2 Round and step constants ............................... 12  
  2.4 AEAD Algorithm: ACE-\texttt{AE-128} ...................... 14  
    2.4.1 Rate and capacity part of state ........................ 15  
    2.4.2 Padding ................................................ 15  
    2.4.3 Loading key and nonce .................................. 16  
    2.4.4 Initialization ......................................... 16  
    2.4.5 Processing associated data .............................. 16  
    2.4.6 Encryption ............................................. 16  
    2.4.7 Finalization ........................................... 17  
    2.4.8 Decryption ............................................. 17  
  2.5 Hash Algorithm: ACE-\texttt{H-256} ....................... 17  
    2.5.1 Message padding ...................................... 18  
    2.5.2 Loading initialization vector .......................... 18  
    2.5.3 Initialization ......................................... 18  
    2.5.4 Absorbing and squeezing ............................... 18  

3 Security Claims .............................................. 20  

4 Security Analysis ............................................ 21  
  4.1 Diffusion .................................................. 21  
  4.2 Differential and Linear Cryptanalysis ....................... 21
# ACE: Submission to the NIST LWC competition

4.2.1 Expected bounds on the maximum probabilities of differential and linear characteristics ........................................... 22
4.3 Algebraic Properties .................................................. 22
4.4 Self Symmetry-based Distinguishers .................................. 24
4.5 Security of $\text{ACE-}AE_{128}$ and $\text{ACE-H-256}$ ................. 25

## Design Rationale

5.1 Choice of the Mode: sLiSCP Sponge Mode .......................... 26
5.2 ACE State Size ....................................................... 27
5.3 ACE Step Function .................................................. 28
5.4 Nonlinear Layer: Simeck sbox ($SB_{64}$) ............................ 28
5.5 Linear Layer: $\pi = (3, 2, 0, 4, 1)$ ................................ 29
5.6 Round and Step Constants ........................................... 29
5.6.1 Rationale ......................................................... 29
5.6.2 Generation of round and step constants ......................... 29
5.7 Number of Rounds and Steps ....................................... 31
5.8 Choice of Rate Positions ............................................ 32
5.9 Statement ............................................................. 32

## Hardware Design and Analysis

6.1 Hardware Design Principles .......................................... 33
6.2 Interface and Top-level Module ....................................... 34
6.2.1 Interface protocol ................................................ 35
6.2.2 Protocol timing .................................................. 37
6.2.3 Control phases ................................................... 40
6.3 Hardware Implementation Details .................................... 42
6.3.1 State machine .................................................... 42
6.3.2 ACE datapath .................................................... 47
6.4 Hardware Implementation Results ..................................... 51
6.4.1 Tool configuration and implementation technologies .......... 52
6.4.2 Implementation results .......................................... 52

## Efficiency Analysis in Software

7.1 Software: High-performance CPU .................................... 55
7.2 Software: Microcontroller ........................................... 58

## Other NIST-LWC Submissions

A.1 Simeck sbox ........................................................ 65
A.2 ACE Permutation ..................................................... 66
A.3 $\text{ACE-}AE_{128}$ ...................................................... 66
A.4 $\text{ACE-H-256}$ ........................................................ 66
C Constants: Sequence to Hex Conversion
List of Figures

2.1 ACE-step .................................................................................................................. 12
2.2 Simeck sbox (SB-64) ................................................................................................. 12
2.3 Schematic diagram of ACE-AE-128 AEAD algorithm ........................................... 15
2.4 Hash algorithm ACE-H-256 ...................................................................................... 18

5.1 LFSR for generating ACE constants ......................................................................... 30
5.2 Schematic of the 3-way parallel LFSR for generation of the constants ........................ 30
5.3 Generation of three 8-bit step constants ..................................................................... 31

6.1 Top-level ACE_module and the interface with the environment .............................. 35
6.2 Interface protocol ....................................................................................................... 36
6.3 Timing diagram: Loading and initialization during ACE-AE-128 ......................... 38
6.4 Timing diagram: Encryption during ACE-AE-128 .................................................. 39
6.5 Timing of tag phase during ACE-AE-128 ............................................................... 39
6.6 Phases and datapath operations .................................................................................. 41
6.7 Control flow between phases ..................................................................................... 43
6.8 Optimized control flow between phases ..................................................................... 43
6.9 State machine: Loading .............................................................................................. 46
6.10 State machine: Hashing ............................................................................................. 46
6.11 State machine: Encryption and decryption ............................................................... 47
6.12 The ACE-datapath .................................................................................................... 48
6.13 Area$^2$ vs Throughput ............................................................................................. 53
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Recommended parameter set for $\text{ACE-\mathcal{AE}-128}$ and $\text{ACE-\mathcal{H}-256}$</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Round and step constants of $\text{ACE}$</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Security goals of $\text{ACE-\mathcal{AE}-128}$ (in bits)</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Security goals of $\text{ACE-\mathcal{H}-256}$ (in bits)</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Minimum number of active Simeck sboxes for $s$-step $\text{ACE}$</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Bounds on the algebraic degree of $\text{ACE}$</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Integral distinguishers of $\text{ACE}$</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>MEDCP of $\text{ACE}$ for $s = 15, 16$</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>Interface signals</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Modes of operation</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Control table for datapath based on phases from Figure 6.6</td>
<td>42</td>
</tr>
<tr>
<td>6.4</td>
<td>3-to-1 “i data multiplexers”</td>
<td>50</td>
</tr>
<tr>
<td>6.5</td>
<td>4-to-1 “o data multiplexers”</td>
<td>50</td>
</tr>
<tr>
<td>6.6</td>
<td>2-to-1 “o data multiplexers”</td>
<td>50</td>
</tr>
<tr>
<td>6.7</td>
<td>$\text{ACE}$ permutation hardware area estimate and implementation results</td>
<td>51</td>
</tr>
<tr>
<td>6.8</td>
<td>Tools and implementation technologies</td>
<td>52</td>
</tr>
<tr>
<td>6.9</td>
<td>ASIC implementation results</td>
<td>53</td>
</tr>
<tr>
<td>6.10</td>
<td>FPGA implementation results</td>
<td>54</td>
</tr>
<tr>
<td>7.1</td>
<td>Benchmarking the results for the $\text{ACE}$ permutation and its modes</td>
<td>58</td>
</tr>
<tr>
<td>7.2</td>
<td>Performance of $\text{ACE}$ on microcontrollers at a clock frequency of 16 MHz</td>
<td>59</td>
</tr>
<tr>
<td>A.1</td>
<td>Submissions with sLiSCP-light like permutations</td>
<td>64</td>
</tr>
<tr>
<td>B.1</td>
<td>Test vector for $\text{ACE}$ permutation</td>
<td>66</td>
</tr>
<tr>
<td>C.1</td>
<td>Generation of round and step constants for $i = 0$</td>
<td>68</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

ACE, often known as one of the strongest cards in a deck of cards, is a 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 [30] 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 security bounds provided using automated tools such as CryptoSMT solver [25] 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 [5]
- **Security of ACE modes.** 128-bit security
- **Hardware performance.** Efficient in hardware. Achieves a throughput of 360 Mbps and has an area of 4250 GE in a 65 nm ASIC. Implementation results are presented for four ASIC libraries and two FPGAs along with parallel implementations.
- **Software performance.** Bit-sliced implementation of ACE permutation achieves a speed of 9.97 cycles/byte
## 1.1 Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \odot Y, X \oplus Y, X</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>X</td>
</tr>
<tr>
<td>${0, 1}^n, {0, 1}^*, \phi$</td>
<td>length $n$ bitstring, variable length bitstring, empty string</td>
</tr>
<tr>
<td>$1^n, 0^n$</td>
<td>length $n$ bitstring with all 1’s, 0’s</td>
</tr>
<tr>
<td>$L^i$</td>
<td>left cyclic shift operator, i.e., for $x \in {0, 1}^n$, $L^i(x) = (x_i, x_{i+1}, \ldots, x_{n-1}, x_0, x_1, \ldots, x_{i-1})$</td>
</tr>
</tbody>
</table>

word/block | a 64-bit binary string |
| $S$ | 320 bit state of \textit{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 \textit{ACE} permutation |
| $A[j]$ | $j$-th byte of word $A$ starting from right |
| $A^i_1, A^i_0$ | right and left 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 $h = 256$ |
| $\ell_X$ | length of $X$ in words where $X \in \{AD, M, C\}$ |

step | one round of \textit{ACE} permutation (see Figure 2.1) |
| round | one round of \textit{Simeck} unkeyed function (see Figure 2.2) |

$SB-64$ | nonlinear operation of \textit{ACE} permutation |
| $u$ | number of rounds, $u = 8$ |
| $s$ | number of steps, $s = 16$ |
| $rc^i_0, rc^i_1, rc^i_2$ | 8-bit round constants |
| $sc^i_0, sc^i_1, sc^i_2$ | 8-bit step constants |

\textit{ACE-AE-k} | \textit{ACE} AEAD algorithm ($k = 128$) |
\textit{ACE-H-h} | \textit{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 algorithms. We summarize the security claims of our submission in Chapter 3 and provide the detailed security analysis 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 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 a 320-bit permutation that operates in a unified duplex sponge mode [5] and offers both AEAD and hashing functionalities in a single hardware circuit. The AEAD algorithm (\texttt{ACE-\AE-k}) and the hash algorithm (\texttt{ACE-\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 \texttt{ACE-\AE-k} is a combination of two algorithms, an authenticated encryption algorithm \texttt{ACE-\E} and the verified decryption algorithm \texttt{ACE-\D}.

An authenticated encryption algorithm \texttt{ACE-\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 \texttt{ACE-\E} is an authenticated ciphertext $C$ of same length as $M$, and an authentication tag $T$ of size $t$ bits. Mathematically, \texttt{ACE-\E} is defined as

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

with

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

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 $\perp$ (error symbol) if the tag verification fails. More formally,

\[
\text{ACE-\D}(K, N, AD, C, T) \in \{M, \perp\}.
\]
2.1.2 **ACE Hash algorithm**

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

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

with $H = ACE-H-h(M, IV)$.

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 functionalities 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>
<thead>
<tr>
<th>Functionality</th>
<th>Algorithm</th>
<th>$r$</th>
<th>$k$</th>
<th>$n$</th>
<th>$t$</th>
<th>$\log_2(d)$</th>
<th>$h$</th>
<th>$iv$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEAD</td>
<td>ACE-$\mathcal{E}$-128</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>124</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Hash</td>
<td>ACE-$\mathcal{H}$-256</td>
<td>64</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>256</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

2.3 **The ACE Permutation**

ACE is an iterative permutation that takes a 320-bit state as an input and outputs a 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 [30] 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 sbox.
Definition 1 (SB-64: Simeck sbox [5]) Let $rc = (q_7, q_6, \ldots, q_0)$ where $q_j \in \{0, 1\}$ and $0 \leq j \leq 7$. A Simeck sbox is a permutation of a 64-bit input, constructed by iterating the Simeck-64 block cipher for 8 rounds with round constant addition $\gamma_j = 131||q_j$ in place of key addition.

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

$$(x_9||x_8) \leftarrow SB-64(x_1||x_0, rc)$$

where

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

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

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

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

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

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

6: Function ACE-step($S^i$):
7: $A^i \leftarrow$ SB-64($A^i_1 || A^i_0, rc^i_0$)
8: $C^i \leftarrow$ SB-64($C^i_1 || C^i_0, rc^i_1$)
9: $E^i \leftarrow$ SB-64($E^i_1 || E^i_0, rc^i_2$)
10: $B^i \leftarrow B^i \oplus C^i \oplus (1^{56} || sc^i_0)$
11: $D^i \leftarrow D^i \oplus E^i \oplus (1^{56} || sc^i_1)$
12: $E^i \leftarrow E^i \oplus A^i \oplus (1^{56} || sc^i_2)$
13: $A^{i+1} \leftarrow D^i$
14: $B^{i+1} \leftarrow C^i$
15: $C^{i+1} \leftarrow A^i$
16: $D^{i+1} \leftarrow E^i$
17: $E^{i+1} \leftarrow B^i$
18: return ($A^{i+1} || B^{i+1} || C^{i+1} || D^{i+1} || E^{i+1}$)

19: Function SB-64($x_1 || x_0, rc$):
20: $rc = (q_7, q_6, \ldots, q_0)$
21: for $j = 2$ to 9 do
22: $x_j \leftarrow (L^5(x_{j-1}) \odot x_{j-1}) \oplus L^1(x_{j-1}) \oplus x_{j-2} \oplus (1^{31} || q_{j-2})$
23: 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 sboxes 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.

Table 2.2: Round and step constants of ACE

<table>
<thead>
<tr>
<th>Step</th>
<th>Round constants $(rc^i_0, rc^i_1, rc^i_2)$</th>
<th>Step constants $(sc^i_0, sc^i_1, sc^i_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 3</td>
<td>(07, 53, 43), (ba, 5d, e4), (9b, 49, 5e), (e0, 7f, cc)</td>
<td>(50, 28, 14), (5c, ae, 57), (91, 48, 24), (8d, c6, 63)</td>
</tr>
<tr>
<td>4 - 7</td>
<td>(d1, be, 32), (1a, 1d, 4e), (22, 28, 75), (f7, 6c, 25)</td>
<td>(53, a9, 54), (60, 30, 18), (68, 34, 9a), (e1, 70, 38)</td>
</tr>
<tr>
<td>8 - 11</td>
<td>(62, 82, 6d), (96, 47, f9), (71, 6b, 76), (aa, 88, a0)</td>
<td>(46, 7b, bd), (94, ce, 67), (40, 20, 10), (4f, 27, 13)</td>
</tr>
<tr>
<td>12 - 15</td>
<td>(2b, dc, b0), (e9, 8b, 09), (cf, 59, 1e), (b7, c6, ad)</td>
<td>(be, 5f, 2f), (5b, ad, d6), (e9, 74, ba), (71, 3f, 1f)</td>
</tr>
</tbody>
</table>
2.4 AEAD Algorithm: **ACE-Æ-128**

In Algorithm 2, we present a high-level overview of **ACE-Æ-128**. The encryption (**ACE-Æ**) and decryption (**ACE-Ð**) processes of **ACE-Æ-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-Æ** and **ACE-Ð**.

**Algorithm 2 ACE-Æ-128 algorithm**

```
1: Authenticated encryption **ACE-Æ**(K, N, AD, M):
2: S ← Initialization(N, K)
3: if |AD| ≠ 0 then:
4: S ← Processing-Associated-Data(S, AD)
5: (S, C) ← Encryption(S, M)
6: T ← Finalization(S, K)
7: return (C, T)
8: Initialization(N, K):
9: S ← load Æ(N, K)
10: S ← ACE(S)
11: for i = 0 to 1 do:
12: S ← (Sr ⊕ Ki, Sr)
13: S ← ACE(S)
14: return S
15: Processing-Associated-Data(S, AD):
16: (AD0||⋯||AD|AD| − 1) ← padKM(AD)
17: for i = 0 to |AD| − 1 do:
18: S ← (Sr ⊕ ADi, Sr ⊕ 06−2)|10
19: S ← ACE(S)
20: return S
21: Encryption(S, M):
22: (M0||⋯||M|M| − 1) ← padKM(M)
23: for i = 0 to |M| − 1 do:
24: Ci ← Mi ⊕ Sr
25: S ← (Ci, Sr ⊕ 06−2)|10
26: S ← ACE(S)
27: C|M|−1 ← trunc-msb(C|M|−1, |M| mod r)
28: C ← (C0, C1, ..., C|M|−1)
29: return (S, C)
30: padKM(X):
31: X ← X|06−1−(X) mod r|
32: return X
33: trunc-lsb(X, n):
34: return (x0, x1, ..., xn−1)
```

1: Verified decryption **ACE-Ð**(K, N, AD, C, T):
2: S ← Initialization(N, K)
3: if |AD| ≠ 0 then:
4: S ← Processing-Associated-Data(S, AD)
5: (S, M) ← Decryption(S, C)
6: T* ← Finalization(S, K)
7: if T* ≠ T then:
8: return ⊥
9: else:
10: return M
11: Decryption(S, C):
12: (C0||⋯||CM|M|−1) ← padKM(C)
13: for i = 0 to |CM| − 2 do:
14: Mi ← Ci ⊕ Sr
15: S ← (Ci, Sr ⊕ 06−2)|10
16: S ← ACE(S)
17: M|M|−1 ← Sr ⊕ CM|M|−1
18: C|M|−1 ← trunc-msb(C|M|−1, |C| mod r) −|trunc-lsb(M|M|−1, r − |C| mod r))
19: M|M|−1 ← trunc-msb(M|M|−1, |C| mod r)
20: M ← (M0, M1, ..., M|M|−1)
21: S ← ACE(C|M|−1, Sr ⊕ 06−2)|10
22: return (S, M)
23: Finalization(S, K):
24: for i = 0 to 1 do:
25: S ← ACE(Sr ⊕ Ki, Sr)
26: T ← tagextract(S)
27: return T
28: trunc-msb(X, n):
29: if n = 0 then:
30: return φ
31: else:
32: return (x0, x1, ..., xn−1)
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 $\ell_M$ $r$-bit blocks $M_0 \parallel \cdots \parallel M_{\ell_M-1}$. A similar procedure is carried out on the associated data $AD$ which results in $\ell_{AD}$ $r$-bit blocks $AD_0 \parallel \cdots \parallel AD_{\ell_{AD}-1}$. 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.

$$\text{pad}_r(M) \leftarrow M \parallel 1 \parallel 0^{r-1-\lfloor |M| \mod r \rfloor}$$

$$\text{pad}_r(AD) \leftarrow \begin{cases} AD \parallel 1 \parallel 0^{r-1-\lfloor |AD| \mod r \rfloor} & \text{if } |AD| > 0 \\ \phi & \text{if } |AD| = 0 \end{cases}$$
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 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. The word $D$ is initialized by the zero bytes. Symbolically, the state is initialized as follows.

$$
A[7], A[6], \ldots, A[0] \leftarrow K_0[7], K_0[6], \ldots, K_0[0] \\
C[7], C[6], \ldots, C[0] \leftarrow K_1[7], K_1[6], \ldots, K_1[0] \\
B[7], B[6], \ldots, B[0] \leftarrow N_0[7], N_0[6], \ldots, N_0[0] \\
E[7], E[6], \ldots, E[0] \leftarrow N_1[7], N_1[6], \ldots, N_1[0] \\
D[7], D[6], \ldots, D[0] \leftarrow 0x00, 0x00, \ldots, 0x00
$$

We use $\text{load-}\mathcal{A}\mathcal{E}(N, K)$ 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 secret key $K$. The state is first loaded using $\text{load-}\mathcal{A}\mathcal{E}(N, K)$ as described above. Afterwards, the permutation $\mathcal{A}\mathcal{E}$ is applied to the state, and the two key blocks are absorbed into the state with $\mathcal{A}\mathcal{E}$ applied each time. The initialization steps are described below.

$$
S \leftarrow \mathcal{A}\mathcal{E}(\text{load-}\mathcal{A}\mathcal{E}(N, K)) \\
S \leftarrow \mathcal{A}\mathcal{E}(S_r \oplus K_0, S_c) \\
S \leftarrow \mathcal{A}\mathcal{E}(S_r \oplus K_1, S_c)
$$

### 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 $\mathcal{A}\mathcal{E}$ permutation is applied to the whole state.

$$
S \leftarrow \mathcal{A}\mathcal{E}(S_r \oplus AD_i, S_c \oplus (0^{r-2} || 01)), \ i = 0, \ldots, \ell_{AD} - 1
$$

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 $\text{ACE}$, i.e.,

$$C_i \leftarrow S_r \oplus M_i,$$

$$S \leftarrow \text{ACE}(C_i, S_c \oplus (0^{c-2} || 10)), \text{ for } i = 0, \ldots, \ell_M - 1$$

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 the last ciphertext block and a single call of $\text{ACE}$, the domain separator is reset to $0\cdot00$ indicating the start of the finalization phase. Afterwards, the two 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.

$$S \leftarrow \text{ACE}(S_r \oplus K_i, S_c), \text{ for } i = 0, 1$$

$$T \leftarrow \text{tagextract}(S).$$

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

$$T_0[7], T_0[6], \ldots, T_0[0] \leftarrow A[7], A[6], \ldots, A[0]$$

$$T_1[7], T_1[6], \ldots, T_1[0] \leftarrow C[7], C[6], \ldots, C[0]$$

### 2.4.8 Decryption

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

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

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

The padding rule (10*) similar to ACE-\( \mathcal{A}E-128 \) is applied to the input message \( M \), where a single 1 followed by 0’s is appended to it such that its length after padding is a multiple of \( r \). We denote the padding rule by

\[
\text{pad}_r(M) = M || 10^{r-1-\lfloor |M| \mod r}.
\]

The resulting padded message is then divided into \( \ell_M \) \( r \)-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 [21] 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 \( \text{load-\( \mathcal{H} \)}(IV) \).

2.5.3 Initialization

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

\[
S \leftarrow \text{ACE(\( \text{load-\( \mathcal{H} \)}(IV))}\right]
\]

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

1: \(\text{ACE-}\mathcal{H}\text{-256}(M, IV)\):
2: \(S \leftarrow \text{Initialization}(IV)\)
3: \(S \leftarrow \text{Absorbing}(S, M)\)
4: \(H \leftarrow \text{Squeezing}(S)\)
5: \(\text{return } H\)

6: \(\text{Initialization}(IV)\):
7: \(S \leftarrow \text{load-}\mathcal{H}(IV)\)
8: \(S \leftarrow \text{ACE}(S)\)
9: \(\text{return } S\)

10: \(\text{pad}_v(M)\):
11: \(M \leftarrow M || (|M| - 1 \mod v)\)
12: \(\text{return } M\)

1: \(\text{Absorbing}(S, M)\):
2: \((M_0 || \cdots || M_{tM-1}) \leftarrow \text{pad}_v(M)\)
3: \(\text{for } i = 0 \text{ to } tM - 1 \text{ do:}\)
4: \(S \leftarrow \text{ACE}(S \oplus M_i, S_c)\)
5: \(\text{return } S\)

6: \(\text{Squeezing}(S)\):
7: \(\text{for } i = 0 \text{ to } 2 \text{ do:}\)
8: \(H_i \leftarrow S_r\)
9: \(S \leftarrow \text{ACE}(S)\)
10: \(H_3 \leftarrow S_r\)
11: \(\text{return } H_0 || H_1 || H_2 || H_3\)
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 any 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-AE-128 and ACE-H-256. In summary, the security claims of ACE-AE-128 and ACE-H-256 are given in Tables 3.1 and 3.2, respectively.

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

Table 3.1: Security goals of ACE-AE-128 (in bits)

<table>
<thead>
<tr>
<th>Confidentiality</th>
<th>Integrity</th>
<th>Authenticity</th>
<th>Data limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>128</td>
<td>128</td>
<td>2^{124}</td>
</tr>
</tbody>
</table>

Table 3.2: Security goals of ACE-H-256 (in bits)

<table>
<thead>
<tr>
<th>Collision</th>
<th>Preimage</th>
<th>Second preimage</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>192</td>
<td>128</td>
</tr>
</tbody>
</table>
Chapter 4

Security Analysis

In this chapter, we first analyze the security of the ACE permutation by assessing its indistinguishability properties 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-AE-128 and ACE-H-256, whose results directly follow the security proofs of sponge modes.

In our analysis, we denote the linear layer by $\pi$, i.e., $\pi$ permutates the words of state. For example, if $\pi(0, 1, 2, 3, 4) = (3, 2, 0, 4, 1)$, then after applying $\pi$, 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 single Simeck sbox. 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 [17, 28], we model ACE using Mixed Integer Linear Programming (MILP) and bound the minimum number of active Simeck sboxes (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 sboxes 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 sboxes in \( s \) steps for a linear layer \( \pi \), and \( p \) denote the Maximum Differential Probability bound (MDP) for a \( u \)-round Simeck sbox in \( \log_2(\cdot) \) scale. An in-depth analysis of values of \( p \) has been provided in [6] (cf. Section 4.2). We choose \( u, s \) and \( \pi \) 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 maximum squared correlation of a linear characteristic is computed analogously using \( \gamma = -15.6 \) and equals \( 2^{-327.6} \), where \( \gamma \) is the maximum square correlation of a 8-round Simeck sbox (cf. Section 4.2.2 [6]).

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 [29, 9] to compute the algebraic degree. We find that the algebraic degree of a 8-round Simeck sbox 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 \( f_0, f_{32}, f_{64}, f_{96}, f_{128}, f_{160}, f_{192}, f_{224}, f_{256}, f_{288} \). 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 degrees are 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 [24]. To search for the longest length integral distinguisher, we set a single 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 sboxes for $s$-step ACE

<table>
<thead>
<tr>
<th>Linear layer</th>
<th>step $s$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td>$(1, 0, 3, 4, 2)$</td>
<td>0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8</td>
</tr>
<tr>
<td>$(1, 0, 4, 2, 3)$</td>
<td>0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8</td>
</tr>
<tr>
<td>$(1, 2, 0, 4, 3)$</td>
<td>0 1 2 3 4 6 8 8 9 10 11 12 14 16 16 17</td>
</tr>
<tr>
<td>$(1, 2, 3, 4, 0)$</td>
<td>0 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td>$(1, 2, 4, 0, 3)$</td>
<td>0 1 1 2 3 5 7 8 9 10 11 13 15 16 17 17</td>
</tr>
<tr>
<td>$(1, 3, 0, 4, 2)$</td>
<td>0 0 1 2 4 4 5 5 7 9 12 13 14 15 16 17 19</td>
</tr>
<tr>
<td>$(1, 3, 4, 0, 2)$</td>
<td>0 0 1 1 1 2 2 2 3 3 4 4 4 4 5 5</td>
</tr>
<tr>
<td>$(1, 3, 4, 2, 0)$</td>
<td>0 0 1 2 3 4 5 6 7 9 11 12 12 13 14 15</td>
</tr>
<tr>
<td>$(1, 4, 0, 2, 3)$</td>
<td>0 1 2 3 4 6 8 8 9 10 11 12 14 16 16 17</td>
</tr>
<tr>
<td>$(1, 4, 3, 0, 2)$</td>
<td>0 1 1 2 4 5 6 7 9 10 11 12 14 15 16 18</td>
</tr>
<tr>
<td>$(1, 4, 3, 2, 0)$</td>
<td>0 1 2 3 5 6 7 9 11 12 13 14 15 17 18 19</td>
</tr>
<tr>
<td>$(2, 0, 1, 4, 3)$</td>
<td>0 1 2 3 4 6 6 8 9 10 11 12 14 16 16 17</td>
</tr>
<tr>
<td>$(2, 0, 3, 4, 1)$</td>
<td>0 1 2 3 4 5 7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td>$(2, 0, 4, 1, 3)$</td>
<td>0 0 1 2 3 3 4 5 7 9 10 11 12 12 13 14</td>
</tr>
<tr>
<td>$(2, 3, 0, 4, 1)$</td>
<td>0 0 1 2 3 5 7 9 10 11 12 13 15 16 17 19</td>
</tr>
<tr>
<td>$(2, 3, 1, 4, 0)$</td>
<td>0 0 1 3 4 6 7 8 9 11 12 13 14 15 16</td>
</tr>
<tr>
<td>$(2, 3, 4, 0, 1)$</td>
<td>0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</td>
</tr>
<tr>
<td>$(2, 3, 4, 1, 0)$</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>$(2, 4, 3, 0, 1)$</td>
<td>0 1 2 3 4 5 7 8 9 10 11 13 14 15 17 19</td>
</tr>
<tr>
<td>$(2, 4, 3, 0, 3)$</td>
<td>0 1 2 3 4 5 7 8 9 10 11 12 13 15 16 17</td>
</tr>
<tr>
<td>$(2, 4, 3, 1, 0)$</td>
<td>0 0 1 2 4 6 7 8 9 10 11 12 13 14 15 16</td>
</tr>
<tr>
<td>$(3, 0, 1, 4, 2)$</td>
<td>0 1 1 2 4 6 8 8 10 11 13 15 16 17 19</td>
</tr>
<tr>
<td>$(3, 0, 1, 4, 2)$</td>
<td>0 1 1 2 4 6 8 8 10 11 13 15 16 17 19</td>
</tr>
<tr>
<td>$(3, 0, 4, 1, 2)$</td>
<td>0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5</td>
</tr>
<tr>
<td>$(3, 0, 4, 2, 1)$</td>
<td>0 1 1 2 3 4 5 6 7 8 9 10 11 12 13 14</td>
</tr>
<tr>
<td>$(3, 2, 0, 4, 1)$</td>
<td>0 1 2 3 5 7 8 9 11 12 13 15 16 17 19 21</td>
</tr>
<tr>
<td>$(3, 2, 1, 4, 0)$</td>
<td>0 1 2 2 3 4 4 5 6 6 7 8 8 8 9 10 10</td>
</tr>
<tr>
<td>$(3, 2, 4, 0, 1)$</td>
<td>0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8</td>
</tr>
<tr>
<td>$(3, 2, 4, 1, 0)$</td>
<td>0 0 1 2 3 6 8 9 9 10 11 12 15 16 18 18</td>
</tr>
<tr>
<td>$(3, 4, 0, 1, 2)$</td>
<td>0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</td>
</tr>
<tr>
<td>$(3, 4, 0, 2, 1)$</td>
<td>0 1 2 3 4 4 5 6 7 8 8 9 10 11 12 12</td>
</tr>
<tr>
<td>$(3, 4, 1, 0, 2)$</td>
<td>0 1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 8 8</td>
</tr>
<tr>
<td>$(3, 4, 1, 2, 0)$</td>
<td>0 1 2 3 5 7 8 9 11 12 13 15 16 17 19 21</td>
</tr>
<tr>
<td>$(4, 0, 1, 2, 3)$</td>
<td>0 1 2 3 4 5 7 8 10 12 13 13 14 15 17 19</td>
</tr>
<tr>
<td>$(4, 0, 3, 1, 2)$</td>
<td>0 0 1 2 3 3 4 5 7 9 10 11 12 12 13 14</td>
</tr>
<tr>
<td>$(4, 0, 3, 2, 1)$</td>
<td>0 1 2 3 4 5 6 8 10 11 12 13 14 15 16 17</td>
</tr>
<tr>
<td>$(4, 2, 0, 1, 3)$</td>
<td>0 0 1 2 4 6 7 9 10 11 12 13 14 15 17 19</td>
</tr>
<tr>
<td>$(4, 2, 1, 0, 3)$</td>
<td>0 1 2 2 3 4 4 5 6 6 7 8 8 9 10 10 10</td>
</tr>
<tr>
<td>$(4, 2, 3, 0, 1)$</td>
<td>0 1 2 3 5 6 8 9 10 11 12 14 16 17 18 19</td>
</tr>
<tr>
<td>$(4, 2, 3, 1, 0)$</td>
<td>0 0 1 1 2 3 4 4 4 5 5 6 7 7 8 8 8 8</td>
</tr>
<tr>
<td>$(4, 3, 0, 1, 2)$</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>$(4, 3, 0, 2, 1)$</td>
<td>0 0 1 2 3 5 6 7 10 10 11 12 13 15 16 17</td>
</tr>
<tr>
<td>$(4, 3, 1, 0, 2)$</td>
<td>0 0 1 2 4 5 6 7 9 10 11 12 13 14 15 17 17</td>
</tr>
<tr>
<td>$(4, 3, 1, 2, 0)$</td>
<td>0 0 1 1 2 3 4 4 4 5 5 6 7 8 8 8 8</td>
</tr>
</tbody>
</table>
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 sbox.

Table 4.3: Integral distinguishers of ACE

<table>
<thead>
<tr>
<th>Steps s</th>
<th>Input division property</th>
<th>Balanced bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1^{32}</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1^{96}</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1^{164}</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1^{224}</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1^{288}</td>
<td>0</td>
</tr>
</tbody>
</table>

4.4 Self Symmetry-based Distinguishers

A cryptographic permutation is vulnerable to attacks such as rotational distinguishers, slide distinguishers [18] and invariant subspace attack [26] which exploit the symmetric properties of a round function. For example, in ACE the nonlinear Simeck sbox 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 list properties of the constants which ensure that each step function of ACE is 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 $\text{ACE-ÅE-128}$ and $\text{ACE-ÅH-256}$

The security proofs of modes based on the sponge construction rely on the indistinguishability of the underlying permutation from a random one \cite{11, 15, 14, 22}. 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 $\text{ACE-ÅE-128}$ and $\text{ACE-ÅH-256}$.

**$\text{ACE-ÅE-128}$ 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 $d \ll c/2$ \cite{14}. The parameter set of $\text{ACE-ÅE-128}$ (Table 2.1 in Section 2.2) with actual effective capacity 254 (2 bits are lost for domain separation) satisfies this condition, and hence $\text{ACE-ÅE-128}$ 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 \cite{22}. 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. Nevertheless, this is another option for ACE with high throughput.

**$\text{ACE-ÅH-256}$ security.** For a sponge based hash with $b = r + c$ and $h$-bit message digest, the generic security bounds \cite{13, 21} 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, $\text{ACE-ÅH-256}$ 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 \([12, 15]\) 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 \(2^d\) \([16]\). When restricting the data complexity to the 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 \textit{et.al.} \([22]\) have shown that sponge based AE achieve higher security bound, i.e., \(\min\{2^{b/2}, 2^c, 2^k\}\) compared to \([14]\). 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 \([15, 10]\). Nonce reuse in the 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 [5] realizes the following objec-
tives:

- The flexibility to adapt the same circuitry to provide both authenticated encryp-
tion and hashing functionalities, as we adopt a unified round function for all
functionalities.
- High key agility, which fits the lightweight requirements, because AE mode 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.
- Keyed initialization and finalization phases that make key recovery hard even if
the internal state is recovered and also renders universal forgery with the knowl-
edge of the internal state unattainable.
- Hardware efficient initialization and finalization stages where the state is initial-
ized 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 implemen-
tations. Such mechanism has been shown to be secure in [22].

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 [12]. 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 [6] structure. Since we aim to build a 320-bit permutation, we could have used a 4-word $sLiSCP$-light with 80-bit Simeck sboxes. However, we found that it is not practical to evaluate most of the cryptographic properties for the resulting permutation using Simeck sboxes 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 sboxes 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 sbox (SB-64)

The Simeck sbox is an unkeyed independently parameterized variant of the round function of the Simon round function [8]. Moreover, it has set a new record in terms of hardware efficiency and performance on almost all platforms [30]. In what follows, we list the reasons that motivated our adoption of Simeck sboxes 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 against algebraic and diffusion-based attacks.

• Each Simeck sbox 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 blocks, 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 sboxes 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 \( \pi \) or \( \pi' \), ACE reaches full bit diffusion in 5 steps and has 21 active Simeck sboxes (see Table 4.1). Accordingly, we picked \( \pi \) 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 [18]. 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 sbox in each round to destroy the preservation of any rotational properties. Moreover, we append 31 ‘1’ bits to each one bit constant, which results many 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 (for theory of \( m \)-sequences see [19]).

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 \( a \) 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.

$$rc_0^i = a_{24i+21}||a_{24i+18}||a_{24i+15}||a_{24i+12}||a_{24i+9}||a_{24i+6}||a_{24i+3}||a_{24i+0}$$

$$rc_1^i = a_{24i+22}||a_{24i+19}||a_{24i+16}||a_{24i+13}||a_{24i+10}||a_{24i+7}||a_{24i+4}||a_{24i+1}$$

$$rc_2^i = a_{24i+23}||a_{24i+20}||a_{24i+17}||a_{24i+14}||a_{24i+11}||a_{24i+8}||a_{24i+5}||a_{24i+2}$$

where

- $rc_0^i$ corresponds to the sequence $a$ with decimation 3
- $rc_1^i$ corresponds to the sequence $a$ shifted by 1, then decimated by 3
- $rc_2^i$ corresponds to the sequence $a$ 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,
Figure 5.3: Three 8-bit step constants, generated from 10 consecutive sequence elements annotated 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:

\[ sc^i_0 = a_{24i+28} \parallel a_{24i+27} \parallel a_{24i+26} \parallel a_{24i+25} \parallel a_{24i+24} \parallel a_{24i+23} \parallel a_{24i+22} \parallel a_{24i+21} \]
\[ sc^i_1 = a_{24i+29} \parallel a_{24i+28} \parallel a_{24i+27} \parallel a_{24i+26} \parallel a_{24i+25} \parallel a_{24i+24} \parallel a_{24i+23} \parallel a_{24i+22} \]
\[ sc^i_2 = a_{24i+30} \parallel a_{24i+29} \parallel a_{24i+28} \parallel a_{24i+27} \parallel a_{24i+26} \parallel a_{24i+25} \parallel a_{24i+24} \parallel a_{24i+23} \]

We provide an example of how to obtain hex values of constants from the \( m \)-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 indistinguishable 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 [20] 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. Note that having a smaller number of rounds results in a weaker Simeck sbox.

**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 sbox and the number of such active sboxes 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 [6] where the CryptoSMT tool [25] is used to obtain the optimal differential characteristics and corresponding probabilities.
Table 5.1: Optimal differential characteristic probability $p$ for $u$-round Simeck sbox and the corresponding MEDCP of ACE for $s = 15, 16$.

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

Accordingly, we have:

\[
\begin{align*}
n_{15} \times -15.8 &= 19 \times -15.8 = -300.2 > -320 \\
n_{16} \times -15.8 &= 21 \times -15.8 = -331.8 < -320.
\end{align*}
\]

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 $(u, s) = (8, 16)$.

### 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 \[5, 7\]. More precisely, we absorb message blocks in words $A$ and $C$. Such rate positions allow the input bits to be processed by the Simeck sboxes as soon as possible so we achieve faster diffusion. Also, our choice forces any injected differences to activate Simeck sboxes 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 \[27\].

### 5.9 Statement

The authors declare that there are no hidden weaknesses in the ACE permutation, ACE-\AE-128 and ACE-\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. Section 6.1 outlines some of the principles underlying our hardware design. Section 6.2 describes the interface and top-level ACE_module module. Section 6.3 goes into the details of the state machine and datapath implementation. And, finally, Section 6.4 presents the implementation results for four ASIC libraries and two FPGAs.

6.1 Hardware Design Principles

The design principles and assumptions followed by the hardware implementations:

1. **Multi-functionality module.** The system should support all three operations, namely authenticated encryption, authenticated decryption, and hashing, in a single module (Figure 6.1), 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 (Table 6.1). Similarly, the output ciphertext, tag, and hash all use a single output port (Table 6.1). That being said, the authors agree with the proposed lightweight cryptography hardware API’s [23] use of separate public and private data ports and will update implementations accordingly.

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 systems 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 (Figure 6.4), 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. In reality, the environment can stall as well. In the future, ACE hardware implementations will be updated to match the proposed lightweight crypto hardware API’s use of a valid/ready protocol for both input and output ports.

4. Use a “pure register-transfer-level” implementation style. In particular, use only registers, not latches; multiplexers, not tri-state buffers; synchronous, not asynchronous reset; no scan-cell flip-flops; clock-gating is used for power and area optimization.

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.

<table>
<thead>
<tr>
<th>Input signal</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset</td>
<td>resets the state machine</td>
</tr>
<tr>
<td>i_mode</td>
<td>mode of operation</td>
</tr>
<tr>
<td>i_dom_sep</td>
<td>domain separator</td>
</tr>
<tr>
<td>i_padding</td>
<td>the last block is padded</td>
</tr>
<tr>
<td>i_data</td>
<td>input data</td>
</tr>
<tr>
<td>i_valid</td>
<td>valid data on i_data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output signal</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_ready</td>
<td>hardware is ready</td>
</tr>
<tr>
<td>o_data</td>
<td>output data</td>
</tr>
<tr>
<td>o_valid</td>
<td>valid data on o_data</td>
</tr>
</tbody>
</table>

Table 6.2: Modes of operation

<table>
<thead>
<tr>
<th>i_mode</th>
<th>Mode</th>
<th>Operation or phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) (0)</td>
<td>ACE-(\mathcal{E})</td>
<td>Encryption</td>
</tr>
<tr>
<td>0 0</td>
<td>ACE-(\mathcal{D})</td>
<td>Decryption</td>
</tr>
<tr>
<td>0 1</td>
<td>ACE-(\mathcal{H})-256</td>
<td>Absorb</td>
</tr>
<tr>
<td>1 0</td>
<td>ACE-(\mathcal{H})-256</td>
<td>Squeeze</td>
</tr>
<tr>
<td>1 1</td>
<td>ACE-(\mathcal{H})-256</td>
<td>Squeeze</td>
</tr>
</tbody>
</table>

The ACE-\(\mathcal{A\mathcal{E}}\)-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 input i_pad is used to indicate that the last i_data block is padded. The hardware is unaware of the lengths of individual phases, hence no internal counters for the number of processed blocks are needed. The domain separators are provided by the environment and serve as an indication of the phase change for AEAD functionality, i.e., whether the input data for ACE-\(\mathcal{A\mathcal{E}}\)-128 is the key, associated data or plaintext/ciphertext. For
ACE-H-256, the phase change is indicated by the change of the $i\_\text{mode}(0)$ signal, as shown in Table 6.2.

### 6.2.1 Interface protocol

The top-level ACE_module is in constant interaction with the environment application. We show this interaction in a form of protocol in Figures 6.2a-6.2d for the ACE-AE-128 encryption example, using the signal names from Figure 6.1. The environment is only allowed to send data to the ACE hardware when it is in the idle mode, which is indicated by the hardware using the $o\_\text{ready}$ signal. The only exception from this behaviour is the reset signal, which will force the hardware module to reset its state machine and return to the idle state and set the $o\_\text{ready}$ signal. Figure 6.2a shows the environment (on the left) resetting the ACE_module (on the right), waiting for the $o\_\text{ready}$ to be asserted, then sending the key and the nonce, as specified in load-AE$(N,K)$ in Section 2.4. Upon receiving all four key and nonce blocks, the ACE_module starts with the initial ACE permutation, as indicated on the right. After completing the permutation, ACE_module enters idle state and asserts the $o\_\text{ready}$ signal to the environment that it can accept new data. The environment proceeds with $K_0$, accompanied by the proper values for $i\_\text{mode}$ and $i\_\text{dom}$.sep. Loading and initialization is the same for both ACE-E and ACE-D, which is why Figure 6.2a shows $i\_\text{mode}=0$ (see Table 6.2). The ACE_module performs the ACE permutation and asserts $o\_\text{ready}$, and the environment responds with the second key block $K_1$.

Figure 6.2b shows the communication between the environment and the ACE_module for the first two blocks of associated data $AD_0$ and $AD_1$. The environment sets $i\_\text{dom}$.sep to value 1 and signals the arrival of new $AD$ block with $i\_\text{valid}$. After completing the ACE permutation, ACE_module asserts the $o\_\text{ready}$. Figure 6.2c shows the handshake signals for encryption of message blocks $M_5$ and $M_6$. The environment sends the plaintexts along with $i\_\text{dom}$.sep=2 and $i\_\text{mode}=00$. The ACE_module receives $M_5$, encrypts it and immediately returns $C_5$ together with asserted $o\_\text{valid}$, then starts the ACE permutation and asserts $o\_\text{ready}$ upon its completion. With exception of the tag generation, encryption (decryption) is the only phase during ACE-
Figure 6.2: Interface protocol
\textit{\texttt{ACE-128}} when the \texttt{ACE\_module} sends data to the environment.

Each time a data block is transmitted between the environment and the \texttt{ACE\_module} the valid bit protocol is used: the environment asserts the \texttt{i\_valid}, and the \texttt{ACE\_module} the \texttt{o\_valid} signal. This naming convention is centred around the hardware module. Another important part of the handshake between the environment and \texttt{ACE\_module} is the \texttt{o\_ready} signal: \texttt{ACE\_module} sets this signal to 1 when it is ready to receive new data from the environment, and to 0 when it is busy and wants the environment to wait.

When all message blocks are encrypted, the finalization and tag generation begins (Figure 6.2d). The environment changes \texttt{i\_dom\_sep} to value 0 and starts sending the key blocks. After receiving $K_0$, \texttt{ACE\_module} performs an \texttt{ACE} permutation and sets \texttt{o\_ready}. After receiving $K_1$, \texttt{ACE\_module} again performs an \texttt{ACE} permutation, but this time replies to the environment with two tag blocks $T_0$, $T_1$, each accompanied by the \texttt{o\_valid} signal. Afterwards, the \texttt{ACE\_module} returns to idle and asserts the \texttt{o\_ready} signal.

As was mentioned before, the \texttt{ACE\_module} is unaware of the number of $AD$ and $M$ blocks, and relies on the environment to set proper values for \texttt{i\_dom\_sep}. However, the \texttt{ACE} state machine is not completely free of counters: a small internal counter is needed to keep track of the number of blocks received and transmitted during the loading and initialization phase and the tag extraction. Another counter is needed to keep track of the \texttt{ACE} permutation, which requires 128 clock cycles to complete. More details follow in Subsection 6.2.3.

In streaming applications, the total length of the data might not be known at the time that the message begins streaming. Hence, each time data is sent to the cipher, the environment informs the cipher what type of data is being sent. This information is easily encoded using a two-bit mode signal to denote which operation is to be performed (Encryption, Decryption, Hashing) and the two-bit domain separator to denote the type of data being processed (associated data, message, ciphertext). The hardware uses \texttt{o\_ready} to signal that it is ready to receive new data, and the environment uses \texttt{i\_valid} to signal that it is sending data to the hardware.

### 6.2.2 Protocol timing

More detailed representation of the events between the environment and the \texttt{ACE\_module} is possible with the use of timing diagrams in Figures 6.3-6.5. In each diagram, the top few lines show the interface signals (Table 6.1), which were already discussed as a part of the communication protocol between the environment and the hardware module. Signals \texttt{i\_mode} and \texttt{i\_dom\_sep} are omitted from the timing diagrams: their current value is the same as shown in the corresponding protocol figure. The vertical ticks on the horizontal lines represent the time: a single column shows the signal values within the same clock period.

**Loading and initialization during \texttt{ACE-\texttt{AE-128}}.** Figure 6.3 shows the loading and
initialization up to the beginning of the second **ACE** permutation; it corresponds to the upper half of the protocol Figure 6.2a. At the top, we can see the interface signals reset, o_ready, i_valid and i_data, followed by the internal signals count, pcount and phase, which are a part of the **ACE** state machine. Counter count is needed to keep track of the number of loaded key and nonce blocks. It is then reused to count the number of key blocks processed during the rest of initialization and during finalization, and for counting the number of produced tag and hash blocks. Counter pcount keeps track of the 128 clock cycles needed for one **ACE** permutation. After the environment deasserts the reset signal, the **ACE_module** clears the internal state machine signals, enters the Load phase and sets the o_ready signal. The environment responds with the first key block $K_0$, accompanied with asserted i_valid. The **ACE_module** stores the new data into its internal state and increments count. Figure 6.3 shows an example when the response of the environment varies, e.g., the delay between $K_0$ and $K_1$ is bigger than delay between $K_1$ and $N_0$. After receiving $N_1$, the **ACE_module** performs the first **ACE** permutation, denoted LoadPerm: the o_ready signal is dropped and the pcount increments every clock cycle. After LoadPerm is finished, the state machine enters the Init phase and the o_ready signal is set to 1 while waiting for the first key block. Arrival of the next i_valid and $K_0$ triggers the second **ACE** permutation.

![Figure 6.3: Timing diagram: Loading and initialization during **ACE-128**](image)

**Encryption during **ACE-128**.** Figure 6.4 shows the timing diagram during the encryption of message blocks $M_5$ and $M_6$, corresponding to protocol in Figure 6.2c. It clearly shows both sides of the valid-bit protocol. The first five lines show the top-level interface signals and line six shows the value of the permutation counter pcount: $8 \cdot 16 = 128$ clock cycles are needed to complete 16 steps, 8 rounds per step, for one **ACE** permutation. The values round and step, shown in the last two lines in Figure 6.4 are not two actual counters, but the lower and upper bits of counter pcount.

After completing the previous permutation, **ACE_module** asserts o_ready. The environment replies with a new message block $M_5$ accompanied by an i_valid signal. The hardware immediately encrypts, returns $C_5$ and asserts o_valid. This clock cycle is also the first round of a new **ACE** permutation and the o_ready is deasserted, indicating that the hardware is busy. Figure 6.4 also shows the **ACE_module** remaining...
Figure 6.4: Timing diagram: Encryption during ACE-\(\mathcal{A}\mathcal{E}\)-128

busy (\(o\_\text{ready} = 0\)) for the duration of one ACE permutation. When \(p\_\text{count}\) wraps around from 127 to 0, the hardware is again idle and ready to receive new input, in this case \(M_6\). The counter \(p\_\text{count}\) is not being used. Since processing of associated data is very similar to encryption, with exception of \(AD\) blocks instead of \(M\) blocks and no output for \(o\_\text{data}\), we do not show a separate timing diagram.

**Tag phase during ACE-\(\mathcal{A}\mathcal{E}\)-128.** Figure 6.5 shows a part of the finalization, tag extract and the return of the state machine into the loading phase, corresponding to the lower part of the protocol in Figure 6.2d. The timing diagram starts with the completion of the ACE permutation after block \(K_0\) was received, followed by \(K_1\) immediately, which triggers the second ACE permutation during finalization, which is also the last ACE permutation of ACE-\(\mathcal{A}\mathcal{E}\)-128. After the permutation, ACE\textunderscore module sends two tag blocks to the environment. The counter \(p\_\text{count}\) is used to return the ACE\textunderscore module to the Load phase, where it sets the \(o\_\text{ready}\) signal and awaits the new key and nonce in case of ACE-\(\mathcal{A}\mathcal{E}\)-128, or fixed IV in case of ACE-\(\mathcal{H}\)-256.

Figure 6.5: Timing of tag phase during ACE-\(\mathcal{A}\mathcal{E}\)-128
6.2.3 Control phases

In the previous section (6.2.2) we touched on the different phases of the control circuitry. The phases are categorizations of the active connections between the ACE permutation on the interface signals. The seven categories, i.e., phases, are shown in Figure 6.6 and described below:

- The first column shows **Load** (Figure 6.6(I.)) and **Permutation** (Figure 6.6(II.)). The domain separator is not used.
  - **Load** phase: the data is loaded (Figure 6.6(I.)) into $A, B, C$ and $E$ words of the internal state $S$ during **ACE-Æ-128**, and into word $B$ of the internal state $S$ during **ACE-H-256**, while the other words of $S$ are set to 0. During each **Load**, a portion of the internal state $S$ is replaced with new data.
  - **Permutation** phase: one iteration of the ACE permutation, without any interactions with the environment. This happens $128 \times$ during the **LoadPerm**, and $127 \times$ for all other ACE permutations.

- The second column shows **Init/ProcAD/Final/Absorb** (Figure 6.6(III.)) and **Squeeze** (Figure 6.6(IV.)). The domain separator is set to 1 for **ProcAD** and to 0 for all other phases in this column.
  - **Init/ProcAD/Final/Absorb** phase: the data is added (XORed) to the $S_r$ portion of the internal state $S$ before entering the ACE permutation, i.e., between two consecutive ACE permutations. This phase is treated as one phase from the hardware perspective, i.e., the state machine drives same control signals, but from the algorithm perspective it captures the behaviour during initialization, processing associated data and finalization for **ACE-Æ-128**, and during absorbing for **ACE-H-256**.
  - **Squeeze** phase: during **ACE-H-256** squeezing, $o_{data}$ is extracted from the $A$ and $C$ words of the internal state $S$, then ACE permutation is triggered (except for the last block $H_3$). There is no adding or replacing of any portion of the internal state $S$.

- The third column shows **Encrypt** (Figure 6.6(V.)) and **Decrypt** (Figure 6.6(VI.)) phase during **ACE-Æ-128**. For both phases, the domain separator is set to 2.
  - **Encrypt** phase: received $i_{data}$ (plaintext) is added to the $S_r$ portion of the internal state $S$ and the result of this operation (ciphertext) is passed to the $o_{data}$ output. The resulting ciphertext is a part of the internal state $S$ when the next ACE permutation begins.
  - **Decrypt** phase: received $i_{data}$ (ciphertext) is XORed with the $S_r$ portion of the internal state $S$ and the result of this operation (plaintext) is passed to the $o_{data}$ output. The resulting plaintext does not enter the next ACE permutation. Instead, the ciphertext from $i_{data}$ is used to replace the $S_r$ portion of the internal state before the next ACE permutation begins.
The last column shows the **Tag** phase (Figure 6.6(VII.)). The domain separator is not used.

- **Tag** phase: during tag extract, the **o_data** is extracted from the \( A \) and \( C \) words of the internal state \( S \).

![Diagram](image)

Figure 6.6: Phases and datapath operations

Note that the phases in Figure 6.6, that are showing **ACE** permutation, only show one round: in Figures 6.6(III.-VI.) the first round of **ACE** permutation, where interaction with the environment is required, and in Figure 6.6(II.) any round of **ACE** permutation, where no interaction with the environment takes place. For example, the encryption of \( M_5 \) block from timing diagram in Figure 6.4, consists one phase Encrypt (V.), followed by \( 127 \times \text{Permutation (II.)} \) - all together 128 rounds of **ACE** permutation. Table 6.3 summarizes the phases shown in Figure 6.6, including datapath operation taken (column dp. op.), and specifies the exact \( S_r \) input to the **ACE** permutation and the output of **ACE_module** for the environment, i.e., the **o_data** value.
Table 6.3: Control table for datapath based on phases from Figure 6.6

<table>
<thead>
<tr>
<th>Function</th>
<th>i_mode</th>
<th>i_dom_sep</th>
<th>Phase</th>
<th>Dp. op</th>
<th>Input to permutation</th>
<th>Output to environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Load</td>
<td>I</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Permutation</td>
<td>II</td>
<td>S_r</td>
<td>×</td>
</tr>
<tr>
<td>En/De-crypt</td>
<td>0−</td>
<td>00</td>
<td>Init</td>
<td>III</td>
<td>S_r ⊕ i_data</td>
<td>×</td>
</tr>
<tr>
<td>En/De-crypt</td>
<td>0−</td>
<td>01</td>
<td>ProcAD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td>10</td>
<td>00</td>
<td>Squeeze</td>
<td>IV</td>
<td>S_r</td>
<td>S_r</td>
</tr>
<tr>
<td>Encrypt</td>
<td>00</td>
<td>10</td>
<td>Encrypt</td>
<td>V</td>
<td>S_r ⊕ i_data</td>
<td>S_r ⊕ i_data</td>
</tr>
<tr>
<td>Decrypt</td>
<td>01</td>
<td>10</td>
<td>Decrypt</td>
<td>VI</td>
<td>i_data</td>
<td>S_r ⊕ i_data</td>
</tr>
<tr>
<td>En/De-crypt</td>
<td>0−</td>
<td>-</td>
<td>Tag</td>
<td>VII</td>
<td>×</td>
<td>A, C of S</td>
</tr>
</tbody>
</table>

- stands for “don’t care”  × stands for “not used”

6.3 Hardware Implementation Details

In this section, we describe the implementation details of ACE module. Section 6.3.1 describes how the state machine is derived from the interface protocols (Figures 6.2a–6.2d) and datapath phases (Figure 6.6). Section 6.3.2 describes the datapath.

6.3.1 State machine

Control flow between phases. Figure 6.7 shows all possible transitions between the state machine phases from Figure 6.6. After reset signal, we first enter Load, followed by a single ACE permutation LoadPerm. Although load-\(\mathcal{A}\mathcal{E}(N,K)\) and load-\(\mathcal{H}(IV)\) are different, they are considered as a unified Load phase for the high-level description of the state machine in Figure 6.7. After LoadPerm, the transition depends on the operation, i.e., mode: the left branch is taken during ACE-\(\mathcal{A}\mathcal{E}-128\), and the right branch during ACE-\(\mathcal{H}-256\).

The first phase on the ACE-\(\mathcal{A}\mathcal{E}-128\) branch is Init. The transition from Init depends on the value of i_dom_sep signal. When i_dom_sep=1, we enter the ProcAD phase, and remain there for as long as i_dom_sep=1. Regardless if current state is Init or ProcAD, the i_dom_sep=2 will trigger transition to either Encrypt or Decrypt, depending on the i_mode value. Note that because of our padding rule, there will always be at least one block with i_dom_sep=2. The FSM will enter the Final phase only when the domain separator changes to 0. The phase Final is followed by the Tag phase and the return to the reset state unconditionally.

For ACE-\(\mathcal{H}-256\), the state machine always transitions to Absorb phase, because the padding rule implies at least one message block. The domain separator is always 0, but we use the 2-bit i_mode signal to trigger the transition from Absorb to Squeeze phase, as shown in Figure 6.7.
Figure 6.7: Control flow between phases

Figure 6.8: Optimized control flow between phases
Optimized control flow between phases. Figure 6.8 shows the optimized control flow from Figure 6.7, with \texttt{ACE-AE-128} on the left and \texttt{ACE-H-256} on the right branch. The optimized figure is annotated with transient states in which the paths split or join. The values of the interface signals are checked in the splitting transient states. Note that transient states do not indicate a clock cycle boundary, while all other states do.

Summary of control flow. The high-level algorithms for \texttt{ACE} (Figures 2.3 and 2.4) were designed to simplify the state machine. Functionally, it is equivalent for the boundary between phases to occur either before or after the permutation. The boundary was placed after the permutation updates the state register. As will be demonstrated in Section 6.3.2, with this structure the two-bit domain separator is sufficient to determine the value of many of the multiplexer select lines and other control signals. All phases that have a domain separator of "00" have the same multiplexer select values. The same also holds true for "01". Unfortunately, this cannot be achieved for "10", because encryption and decryption require different control signal values, but the same domain separator. Using the domain separator to signal the transition between phases for encryption and decryption also simplified the control circuit. For hashing, the transition from absorbing to squeezing is controlled by the \texttt{i_mode} signal.

A final note about the control flow diagrams in this section: only when the number of iterations in a certain phase depends on the length of the data, i.e., $\ell_X$, $X \in \{AD,M,C\}$, which is indicated by the value change of interface signal \texttt{i_mode} or \texttt{i_dom_sep}, we show the transition to itself. While the phases Load, Init, Final, Tag and Squeeze also take more than one iteration, the number of iterations is fixed by the \texttt{ACE-AE-128} and \texttt{ACE-H-256} algorithms specified in Chapter 2 and hence not shown in the control flow diagrams in Figures 6.7 and 6.8. However, this level of detail is included in the state machine Figures 6.9-6.11.

Derivation of state machine from control flow. The implementation details for the control flow from Figure 6.8 are shown as state machines, spread out through Figures 6.9-6.11. All the phases are split into at least two parts, following the convention i\texttt{StateName} and p\texttt{StateName}, where the prefix “i” stands for “idle” and “p” for “permutation”, and the state name corresponds to the phase name. In the “idle” states, the \texttt{ACE_module} is waiting for new input from the environment, i.e., for \texttt{i_valid}=1. In the “permutation” states the \texttt{ACE} permutation is running and the \texttt{ACE_module} is busy, i.e., \texttt{o_ready}=0. The normal structure can be seen in the i\texttt{Init} and p\texttt{Init} states in Figure 6.11. There are a few exceptions to the normal structure:

- The phase Load (Figure 6.9) can receive multiple blocks consecutively without running a permutation. Hence, there is a third state Load without a prefix in addition to iLoad and pLoad.
- The phase Tag (Figure 6.11) can transmit multiple blocks consecutively without running a permutation. Hence, there is no need for a p-state.
- The states pAbsorb and pSqueeze (Figure 6.10) have the same behaviour for
idling, and so their idle states are merged into $i_{\text{Hash}}$.

- Similarly, states $p_{\text{ProcAD}}$, $p_{\text{Encrypt}}$ and $p_{\text{Decrypt}}$ all use the same merged idle state $i_{\text{AED}}$.

Each state is annotated with four circles to denote the four bits encoding the current values on the interface signals $i_{\text{mode}}$ and $i_{\text{dom} \_\text{sep}}$. An empty circle denotes 0, a filled circle denotes 1, and a circle with a dash is a “don’t care” value, meaning the behaviour is independent of this bit. Each transition between a pair of states is annotated with a roman numeral from Figure 6.6 denoting the datapath operation to be performed.

**State machine: Loading.** The loading part of the state machine is shown in Figure 6.9. The reset control signal places the state machine into the $i_{\text{Load}}$ idle state, where it awaits the first key block in the case of ACE-$\mathcal{AE}$-128 or a single IV block in the case of ACE-$\mathcal{H}$-256. In case of ACE-$\mathcal{AE}$-128, the environment sends 4(2 + 2) key and nonce blocks, and the state machine uses an internal counter $\text{count}$ to keep track of the loading, i.e., to keep track of iterations back to state $i_{\text{Load}}$. This behaviour was explained in timing diagram in Figure 6.3. After the loading blocks are received, the state machine enters the permutations state $p_{\text{Load}}$, which is controlled by the counter $p_{\text{count}}$ counting steps and rounds as was shown in Figure 6.4. The transition to the left branch is made in the case of $i_{\text{mode}}=0-$, i.e., for ACE-$\mathcal{AE}$-128, in which case the state machine continues in Figure 6.11, and to the right branch in the case of $i_{\text{mode}}=1-$, i.e., in the case of ACE-$\mathcal{H}$-256 in which case the state machine continues in Figure 6.10. In both cases, the next state is an idle state.

**State machine: Hashing.** The hashing states continue in Figure 6.10, corresponding to the Absorb and Squeeze phase in the right branch of the control flow in Figure 6.8. The first state entered is the merged idle state $i_{\text{Hash}}$. Upon the next $i_{\text{valid}}$, we can continue to one of the permutation states based on the value of the $i_{\text{mode}}(0)$ signal, which is set by the environment. The left loop in Figure 6.10 shows the state machine entering $p_{\text{Absorb}}$ when $i_{\text{mode}}=10$, followed by the $i_{\text{Hash}}$ idle state. The end of the message for ACE-$\mathcal{H}$-256 is indicated by the mode change to $i_{\text{mode}}=11$, which causes the transition from $i_{\text{Hash}}$ to $p_{\text{Squeeze}}$. Note that since the number of hash blocks is fixed and does not require any input from the environment, we do not need a return loop from $p_{\text{Squeeze}}$ to $i_{\text{Hash}}$. The state machine uses the counter $\text{count}$ to keep track of the hash blocks produced so far. No ACE permutation is needed after $H_3$.

**State machine: Encryption and decryption.** The left branch in Figure 6.9, taken in the case of ACE-$\mathcal{AE}$-128, is shown in Figure 6.11. Similarly, we first enter the $i_{\text{Init}}$ state and transition to $p_{\text{Init}}$ at the arrival of the next $i_{\text{valid}}$. The counter $p_{\text{count}}$ is, as usual, used to keep track of the ACE permutation, and the counter $\text{count}$ of the number of iterations in Init phase. After the second ACE permutation in the Init phase is completed, we proceed to $i_{\text{AED}}$, a merged idle state for $i_{\text{ProcAD}}$, $i_{\text{Encrypt}}$ and $i_{\text{Decrypt}}$. The next $i_{\text{valid}}$ triggers the transition
to pProcAD if $i_{\text{dom} \text{sep}}=01$ or to transient state on the right if $i_{\text{dom} \text{sep}}=10$, for either pEncrypt or pDecrypt, depending on the $i_{\text{mode}}$. Either way, an ACE permutation starts, and the pcount runs for 128 cycles. After completing an ACE permutation, the state machine returns to the idle state $i_{\text{AED}}$. When we observe $i_{\text{dom} \text{sep}}=00$, the state machine will transition into $p_{\text{Final}}$ state and run the first ACE permutation in Final phase. Again, counter count is used to keep track of the two iterations. The idle state $i_{\text{Final}}$ is entered only once. Finally, the state machine enters the $\text{Tag}$ state, and $\text{ACE module}$ transmits two tag blocks to the environment. Again, counter count is used, but this time, no ACE permutation is required.

**Summary of state machine.** The state machine is responsible for the $o_{\text{valid}}$ and $o_{\text{ready}}$ interface signals. It is also tasked with control signals for the multiplexers in ACE datapath, which accommodate different interactions between ACE module and the environment, i.e., different phases from Figure 6.6. This will be discussed in more detail in Section 6.3.2.
The state machine and encodings for control signals were designed to take advantage of similarities in structure to enable optimizations in the control circuitry. The only control-flow decision made within an idle state is to exit when \( i_{\text{valid}} = '1' \). This reduces the number of idle states and facilitates combinational logic optimizations due to the uniform structure of the control flow. Loading, squeezing, and finalization all use the same one-hot counter to count their iterations. Also, all seven of the states that perform the permutation have the same control structure, which provides opportunities for logic synthesis optimizations, such as common subexpression elimination.

### 6.3.2 ACE datapath

Figure 6.12 shows the schematic for the ACE datapath. The top of the figure depicts the five 64-bit signals A, B, C, D and E, with the A and C registers split into half. The triangular shapes on signal lines denote splitting or joining of signals, e.g., the 64-bit data input \( i_{\text{data}} \) is split into two halves, one for each 32-bit part of the \( S_r \). Similarly, the output of the 32-bit 3-to-1 multiplexer is joined with the unmodified \( A_0 \).
(half of signal A) before entering the SB-64\textsubscript{1}. Next the hardware components required for absorbing, replacing and driving the outputs, annotated with “\texttt{i.data} and \texttt{o.data} muxes”, are shown. The rest of the Figure 6.12 shows one step of the ACE permutation, annotated on the left.

Module SB-64\textsubscript{1} in Figure 6.12 implements one out of eight rounds required for SB-64. The three parallel SB-64\textsubscript{1} modules are shown in Figure 6.12. There is an important difference between Figures 2.1 and 6.12:

- Figure 2.1 shows the ACE-step, containing three parallel Simeck boxes SB-64
- Figure 6.12 shows the ACE-datapath, the actual hardware implementation, where we implement one round SB-64\textsubscript{1} and reuse it 8 times to complete one SB-64.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{ace_datapath}
\caption{The ACE-datapath}
\end{figure}

The repeated use of SB-64\textsubscript{1} is functionally equivalent to SB-64. However, implementing SB-64\textsubscript{1} has multiple advantages. The SB-64\textsubscript{1} module is one-eighth the size of a full SB-64 module. Also using SB-64\textsubscript{1} enables the step and round to use same hardware circuitry. The rounds and steps always use the same hardware, but in different clock cycles, which forces the use of multiplexers inside the ACE permutation. Additions with round and step constants, SB-64\textsubscript{1}, linear permutation layer and the multiplexers to choose between round and step constants are annotated as well. The step/round
multiplexers are the last layer of multiplexers to choose between loading and permutation outputs before updating the 64-bit registers A, B, C, D and E.

**Hardware circuitry for en/de-cryption and hash.** Figure 6.12 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 6.12. Apart from the output generation, this behaviour is the same for ACE-\(A\&\)-128 initialization, processing associated data, encryption, finalization, and for the ACE-\(H\)-256 absorbing and squeezing phase. For ACE-\(A\&\)-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.

**The FSM phases and “i\_data and o\_data muxes”.** The state machine is responsible to drive the control signals for all the multiplexers shown in Figure 6.12 based on the current state and values of counters \(\text{count}\) and \(\text{pcount}\). The “round/step muxes” choose the step output (left mux input), when the \(\text{pcount}\) changes in its step part, and the round input (right mux input), when only the lower round bits of the counter \(\text{pcount}\) change, as explained in timing diagram in Figure 6.4. For most of the time, the “load/perm muxes” are passing the permutation signals (left mux inputs), with the exception of the Load stages before LoadPerm, as shown in control diagrams and state machines in Figures 6.7-6.9. The “i\_data and o\_data muxes” consist of six 32-bit multiplexers: two 3-to-1 “i\_data multiplexers” and two pairs of 4-to-1 and 2-to-1 “o\_data multiplexers”.

Tables 6.4-6.6 below show different scenarios for the “i\_data and o\_data muxes”. There are three sets of these multiplexers: 3:1 i\_data multiplexers (Table 6.4), 4:1 o\_data multiplexers (Table 6.5), and 2:1 o\_data multiplexers (Table 6.6). Each set of multiplexers has one instance in the A\(_1\) column of the datapath and one instance in the C\(_1\) column (Figure 6.12). All three tables are formatted in the same way. Column 2 corresponds to the multiplexer for A\(_1\) and column 3 to the multiplexer for C\(_1\). The multiplexer inputs are linked to ACE hardware phases in Figure 6.6 in column 4. Recall that phases III-VI include only the first iteration of the ACE permutation, which is the iteration in which all the i\_data and o\_data interaction with the environment takes place. Every one of these phases is then followed by the remaining 127 iterations of the ACE permutation, i.e., 127× phase II. In phase II, the i\_data and o\_data are disconnected from the ACE permutation circuit in the lower part of Figure 6.12. For the “i\_data multiplexers” in Table 6.4 there are two exceptions, which have the same multiplexer inputs for all 128 iterations of ACE permutation: the hash squeezing (IV followed by 127× II) and the loading permutation LoadPerm (128× II). Table 6.6 has two exceptions, where no output is generated during any part of the phase, including all 128 iterations of the ACE permutation: Init, ProcAD, Final, Absorb (Figure 6.6(III)), and the all Load and LoadPerm phases (Figure 6.6(I, II)).
Table 6.4: 3-to-1 “i_data multiplexers”

<table>
<thead>
<tr>
<th>mux input position *</th>
<th>S_r input for permutation</th>
<th>4-to-1 mux output for 2-to-1 mux subfigure</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i_data0</td>
<td>VI</td>
<td>VI</td>
</tr>
<tr>
<td>1</td>
<td>A_1 \oplus i_data0</td>
<td>IV, V</td>
<td>add new data into S_r</td>
</tr>
<tr>
<td>2</td>
<td>A_1</td>
<td>IV, II</td>
<td>hash squeezing and during ACE permutation iter. II</td>
</tr>
</tbody>
</table>

* mux input positions are numbered from the left

Table 6.5: 4-to-1 “o_data multiplexers”

<table>
<thead>
<tr>
<th>mux input position *</th>
<th>4-to-1 mux output for 2-to-1 mux o_data0 part</th>
<th>o_data1 part</th>
<th>4-to-1 mux output for 2-to-1 mux o_data0 part</th>
<th>o_data1 part</th>
<th>Fig.6.6 subfigure</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A_1 \oplus i_data0</td>
<td>C_1 \oplus i_data1</td>
<td>V, VI</td>
<td>data block C_i or M_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A_1</td>
<td>C_1</td>
<td>IV</td>
<td>hash block H_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A_1</td>
<td>A_0</td>
<td>VII</td>
<td>tag block T_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C_1</td>
<td>C_0</td>
<td>VII</td>
<td>tag block T_1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* mux input positions are numbered from the left

Table 6.6: 2-to-1 “o_data multiplexers”

<table>
<thead>
<tr>
<th>mux input position *</th>
<th>2-to-1 mux output for 4-to-1 mux 4_to_1 part</th>
<th>o_data1 part</th>
<th>Fig.6.6 subfigure</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4-to-1 o_data0</td>
<td>all 0 block</td>
<td>IV, V</td>
<td>output data block M_i, C_i, T_i or H_i</td>
</tr>
<tr>
<td>1</td>
<td>4-to-1 o_data1</td>
<td>all 0 block</td>
<td>VI, VII</td>
<td>M_i, C_i, T_i or H_i</td>
</tr>
</tbody>
</table>

* mux input positions are numbered from the left

**Estimated and synthesized cost of ACE permutation.** In Table 6.7, 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 XOR” 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.

First we provide the estimate for the ACE permutation without multiplexers. Then we add the round/step multiplexers and a minimum number of load multiplexers, in this case for word B which is loaded during ACE-H-256, and the synthesized area of the LFSR for the constant generation, to obtain the estimate (and synthesized results) for the ACE permutation with minimal amount of multiplexers added.

Finally, we add the estimates for the hardware needed to support the mode. We estimate the “i_data and o_data muxes” as 2-to-1 muxes, e.g., a 4-to-1 mux is estimated as 3 2-to-1 muxes, yielding in total 6 2-to-1 muxes per bit. The XORs for mode are the ones required for the encryption/decryption. Additional load multiplexers are needed to load the entire state. These components, listed in the second part of Table 6.7, amount to 1408 GE (estimated), but the synthesis reports show, the tools were able
to do some optimizations, e.g., using actual (and smaller) 3-to-1 muxes. Finally, we list the estimate and the synthesized result for the complete datapath, followed by the synthesized state machine and the results for \textit{ACE\_module}. The complete cipher results reported here are for logic synthesis (i.e., before place-and-route) with a sufficiently long clock period to get a minimum area. This differs from the results in Table 6.9, where the results are for physical synthesis (after place and route) and the selected result is the one with the maximum performance over area-squared ratio (Section 6.4.2).

Table 6.7: \textit{ACE} permutation hardware area estimate and implementation results

<table>
<thead>
<tr>
<th>Component</th>
<th>Estimate per unit [GE]</th>
<th>Count</th>
<th>Estimate per component [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>State registers</td>
<td>3.75</td>
<td>320</td>
<td>1200</td>
</tr>
<tr>
<td>\textit{SB-64}$_1$</td>
<td>154†</td>
<td>3</td>
<td>462</td>
</tr>
<tr>
<td>Other XORs</td>
<td>2.00</td>
<td>3 × (64 + 8)</td>
<td>432</td>
</tr>
<tr>
<td>Permutation without muxes (estimate)</td>
<td></td>
<td></td>
<td>2094</td>
</tr>
<tr>
<td>round/step muxes</td>
<td>2.00</td>
<td>320</td>
<td>640</td>
</tr>
<tr>
<td>min load muxes</td>
<td>2.00</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>LFSR for constants</td>
<td>49†</td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>Permutation with muxes (estimate)</td>
<td></td>
<td></td>
<td>2911</td>
</tr>
<tr>
<td>Synthesized result†</td>
<td></td>
<td></td>
<td>2870</td>
</tr>
<tr>
<td>\textit{i_data/ o_data} muxes</td>
<td>2.00</td>
<td>6 × 32 × 2 = 384</td>
<td>768</td>
</tr>
<tr>
<td>XORs for mode</td>
<td>2.00</td>
<td>32 × 2 = 64</td>
<td>128</td>
</tr>
<tr>
<td>Additional load muxes</td>
<td>2.00</td>
<td>64× = 256</td>
<td>512</td>
</tr>
<tr>
<td>Hardware for the mode (estimate)</td>
<td></td>
<td></td>
<td>1408</td>
</tr>
<tr>
<td>Synthesized result†</td>
<td></td>
<td></td>
<td>1128</td>
</tr>
<tr>
<td>Complete datapath (estimate)</td>
<td></td>
<td></td>
<td>4319</td>
</tr>
<tr>
<td>Synthesized result†</td>
<td></td>
<td></td>
<td>3998</td>
</tr>
<tr>
<td>State machine (synthesized result†)</td>
<td></td>
<td></td>
<td>318</td>
</tr>
<tr>
<td>\textit{ACE_module} (estimate)</td>
<td></td>
<td></td>
<td>4637</td>
</tr>
<tr>
<td>Synthesized result†</td>
<td></td>
<td></td>
<td>4317</td>
</tr>
</tbody>
</table>

† pre-PAR implementation results

### 6.4 Hardware Implementation Results

In this section, we provide the ASIC and FPGA implementation results of \textit{ACE} and its modes. We first give the details of the synthesis and simulation tools and then present the implementation results.
6.4.1 Tool configuration and implementation technologies

Table 6.8 lists the configuration details of synthesis and simulation tools and libraries for both ASIC and FPGA implementations. All area results are post place-and-route. Energy results are computed through timing simulation of the post place-and-route design at a clock speed of 10 MHz.

For ASICs, logic synthesis was done using the `compileUltra` command and clock gating; and physical synthesis (place-and-route) was done with a density of 95%. By selecting a target clock speed, synthesis for ASICs can exhibit a significant range in tradeoffs between speed and area for the same RTL code. The results reported here reflect the clock speed and area that obtained the highest ratio of performance over area-squared. We used area squared, because area is a reasonable approximation of power and is much less sensitive to the choice of the ASIC library than is power itself.

6.4.2 Implementation results

Figure 6.13, Table 6.9, and Table 6.10 present the hardware implementation results. Note that `ACE_module` performs all three functionalities (authenticated encryption, verified decryption and hashing) in a single module.

Figure 6.13 shows area² vs. throughput for ASICs with different degrees of parallelization, denoted by A-p (p = 1, 2, 3, 4, 8). The throughput axis is scaled as log(Tput) and the area axis is scaled as log(area²). The grey contour lines denote the relative optimality of the circuits using Tput/area². Throughput is increased by increasing the degree of parallelization (unrolling), which reduces the number of clock cycles per permutation round. Going from p = 1 to p = 8 results in a 1.72× area increase, and
optimality increases as parallelism increases from 1 to 8.

As can be seen by the relative constant size of the shaded rectangles enclosing the data points, the relative area increase with parallelization is relatively independent of implementation technology.

Throughput is measured in bits per clock cycle (bpc), and plotted on a log scale axis. The area axis is scaled as log(Area^2).

Figure 6.13: Area^2 vs Throughput

Table 6.9: ASIC implementation results

<table>
<thead>
<tr>
<th>Label</th>
<th>Tput [bpc]</th>
<th>ST Micro 65 nm</th>
<th>TSMC 65 nm</th>
<th>ST Micro 90 nm</th>
<th>IBM 130 nm</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1</td>
<td>0.5</td>
<td>4250</td>
<td>720</td>
<td>27.9</td>
<td>4600</td>
</tr>
<tr>
<td>A-2</td>
<td>1.0</td>
<td>4780</td>
<td>618</td>
<td>18.4</td>
<td>5290</td>
</tr>
<tr>
<td>A-4</td>
<td>2.0</td>
<td>5760</td>
<td>394</td>
<td>15.1</td>
<td>6260</td>
</tr>
<tr>
<td>A-8</td>
<td>4.0</td>
<td>7240</td>
<td>246</td>
<td>11.4</td>
<td>8090</td>
</tr>
</tbody>
</table>

Table 6.9 represents the same data points as Figure 6.13 with the addition of maximum frequency (f, MHz) and energy per bit (E, nJ). Energy is measured as the average
value while performing all cryptographic operations over 8192 bits of data at 10 MHz. As throughput increases, energy per bit decreases consistently, despite higher circuit area and, therefore, power consumption. Connecting modules in a combinational chain can result in an exponential increase of the number of glitches, which drastically increases power consumption. However, because of the small size of the ACE permutation, the actual increase in power is not that significant for parallelism up to degree 8.

Table 6.10: FPGA implementation results

<table>
<thead>
<tr>
<th>Module</th>
<th>Frequency [MHz]</th>
<th># of slices</th>
<th># of FFs</th>
<th># of LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE permutation</td>
<td>181</td>
<td>215</td>
<td>327</td>
<td>381</td>
</tr>
<tr>
<td>ACE_module</td>
<td>68</td>
<td>727</td>
<td>353</td>
<td>1410</td>
</tr>
<tr>
<td>Xilinx Spartan 3 (xc3s200-5ft256)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACE permutation</td>
<td>306</td>
<td>127</td>
<td>327</td>
<td>378</td>
</tr>
<tr>
<td>ACE_module</td>
<td>123</td>
<td>429</td>
<td>365</td>
<td>1272</td>
</tr>
<tr>
<td>Xilinx Spartan 6 (xc6slx9-3ftg256)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACE permutation</td>
<td>128</td>
<td>327</td>
<td>327</td>
<td>296</td>
</tr>
<tr>
<td>ACE_module</td>
<td>51</td>
<td>781†</td>
<td>354</td>
<td>781</td>
</tr>
</tbody>
</table>

Table 6.10: FPGA implementation results

Intel/Altera Stratix IV (EP4SGX70HF35M3)

† 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 constrained 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 \( S_i = s_0^i s_1^i s_2^i s_3^i s_4^i s_5^i s_6^i s_7^i \) 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.

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

Then the packing operation is defined as

\[
\text{PACK}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) :
\]

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

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 \( \text{UNPACK}() \), is the inverse of the packing operation, which we omit here. Both operations are implemented using \texttt{vpermcd} and \texttt{vpermm21128}, \texttt{vpuncklqdq} and \texttt{vpunpcchkqdq} 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 \( A = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 \) and \( B = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \). 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

\[
\text{PACK\_SB-64}(A, B) : \quad \text{UNPACK\_SB-64}(A, B) :
\]

\[
\begin{align*}
A &\leftarrow b_0 b_2 b_4 a_0 a_2 a_4 a_6 a_8; & A &\leftarrow b_0 b_2 b_4 a_0 a_2 a_4 a_6 a_8; \\
B &\leftarrow b_7 b_5 b_3 a_7 a_5 a_3 a_1 a_9; & B &\leftarrow b_7 b_5 b_3 a_7 a_5 a_3 a_1 a_9;
\end{align*}
\]

ROAX operation. We create an instruction for one round of execution of SB-64, denoted by ROAX, which is given by

\[
\text{ROAX}(A, B, q_1, q_2) :
\]

\[
\begin{align*}
tmp &\leftarrow A; & C &\leftarrow 0xffffffff; \\
A &\leftarrow (L^5(A) \oplus A) \oplus L^1(A); \\
A &\leftarrow A \oplus B \oplus (C \oplus q_1, C \oplus q_2, \ldots, C \oplus q_1, C \oplus q_2); \\
B &\leftarrow tmp;
\end{align*}
\]
Algorithm 4 Eight parallel instances of the ACE permutation

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

3: \((R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) \leftarrow \text{PACK}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9);\)
4: for \(i = 0\) to \(15\) do:
5: \(R_0, R_1 \leftarrow \text{PACK}_\text{SB-64}(R_0, R_1);\)
6: \(R_2, R_3 \leftarrow \text{PACK}_\text{SB-64}(R_2, R_3);\)
7: \(R_8, R_9 \leftarrow \text{PACK}_\text{SB-64}(R_8, R_9);\)

8: for \(j = 0\) to \(7\) do:
9: \(R_0, R_1 \leftarrow \text{ROAX}(R_0, R_1, r_c^0[j], r_c^1[j]);\)
10: \(R_2, R_3 \leftarrow \text{ROAX}(R_2, R_3, r_c^0[j], r_c^1[j]);\)
11: \(R_8, R_9 \leftarrow \text{ROAX}(R_8, R_9, r_c^0[j], r_c^1[j]);\)
12: end for
13: \(R_0, R_1 \leftarrow \text{UNPACK}_\text{SB-64}(R_0, R_1);\)
14: \(R_2, R_3 \leftarrow \text{UNPACK}_\text{SB-64}(R_2, R_3);\)
15: \(R_8, R_9 \leftarrow \text{UNPACK}_\text{SB-64}(R_8, R_9);\)
16: \(C \leftarrow \text{0xffff00000}; D \leftarrow \text{0xffffffff};\)
17: \(\text{tmp0} \leftarrow (D, C \oplus sc^0_0, D, C \oplus sc^1_0, D, C \oplus sc^0_1, D, C \oplus sc^1_1);\)
18: \(\text{tmp1} \leftarrow (D, C \oplus sc^0_2, D, C \oplus sc^1_2, D, C \oplus sc^0_3, D, C \oplus sc^1_3);\)
19: \(R_4 \leftarrow R_4 \oplus \text{tmp0}; R_5 \leftarrow R_5 \oplus \text{tmp0};\)
20: \(R_6 \leftarrow R_6 \oplus \text{tmp0}; R_7 \leftarrow R_7 \oplus \text{tmp0};\)
21: \(R_8 \leftarrow R_8 \oplus \text{tmp1}; R_9 \leftarrow R_9 \oplus \text{tmp1};\)
22: \((R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) \leftarrow \text{SWAPBLKS}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9);\)
23: end for
24: \((R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9) \leftarrow \text{UNPACK}(R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9);\)
25: return \((R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9);\)

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.

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

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Speed [cpb]</th>
<th>Instruction Set</th>
<th>CPU Name Spec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>15.66</td>
<td>SSE2</td>
<td>Skylake</td>
</tr>
<tr>
<td></td>
<td>9.97</td>
<td>AVX2</td>
<td>Intel i7-6700</td>
</tr>
<tr>
<td></td>
<td>16.96</td>
<td>SSE2</td>
<td>Haswell</td>
</tr>
<tr>
<td></td>
<td>10.56</td>
<td>AVX2</td>
<td>Intel i7-4790</td>
</tr>
<tr>
<td>ACE-(\mathcal{AE})-128</td>
<td>110.29</td>
<td>SSE2</td>
<td>Skylake</td>
</tr>
<tr>
<td></td>
<td>68.53</td>
<td>AVX2</td>
<td>Intel i7-6700</td>
</tr>
<tr>
<td></td>
<td>128.66</td>
<td>SSE2</td>
<td>Haswell</td>
</tr>
<tr>
<td></td>
<td>89.10</td>
<td>AVX2</td>
<td>Intel i7-4790</td>
</tr>
<tr>
<td>ACE-(\mathcal{H})-256</td>
<td>95.65</td>
<td>SSE2</td>
<td>Skylake</td>
</tr>
<tr>
<td></td>
<td>58.81</td>
<td>AVX2</td>
<td>Intel i7-6700</td>
</tr>
<tr>
<td></td>
<td>108.15</td>
<td>SSE2</td>
<td>Haswell</td>
</tr>
<tr>
<td></td>
<td>66.12</td>
<td>AVX2</td>
<td>Intel i7-4790</td>
</tr>
</tbody>
</table>

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-_AES-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 \((5 + \ell)(\text{resp. } (4 + \ell))\) executions of the permutation are needed to evaluate the AE (resp. hash) mode where \(\ell\) 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 7.2: Performance of **ACE** on microcontrollers at a clock frequency of 16 MHz

<table>
<thead>
<tr>
<th>Platform</th>
<th>Primitive</th>
<th>#AD blocks</th>
<th>#M blocks</th>
<th>Memory (bytes)</th>
<th>Throughput [Kbps]</th>
<th>Energy/bit [nJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>ACE</strong> permutation</td>
<td>-</td>
<td>-</td>
<td>304</td>
<td>1456</td>
<td>69440</td>
</tr>
<tr>
<td></td>
<td>ACE-_AES-128</td>
<td>0</td>
<td>16</td>
<td>330</td>
<td>1740</td>
<td>1445059</td>
</tr>
<tr>
<td></td>
<td>ACE-_AES-128</td>
<td>2</td>
<td>16</td>
<td>330</td>
<td>1786</td>
<td>1582892</td>
</tr>
<tr>
<td></td>
<td>ACE-H-256</td>
<td>-</td>
<td>2</td>
<td>330</td>
<td>1682</td>
<td>413056</td>
</tr>
<tr>
<td></td>
<td>ACE-H-256</td>
<td>-</td>
<td>16</td>
<td>330</td>
<td>1684</td>
<td>1375672</td>
</tr>
<tr>
<td>16-bit MSP430F2370</td>
<td><strong>ACE</strong> permutation</td>
<td>-</td>
<td>-</td>
<td>523</td>
<td>1598</td>
<td>13003</td>
</tr>
<tr>
<td></td>
<td>ACE-_AES-128</td>
<td>0</td>
<td>16</td>
<td>559</td>
<td>1790</td>
<td>269341</td>
</tr>
<tr>
<td></td>
<td>ACE-_AES-128</td>
<td>2</td>
<td>16</td>
<td>559</td>
<td>1822</td>
<td>294988</td>
</tr>
<tr>
<td></td>
<td>ACE-H-256</td>
<td>-</td>
<td>2</td>
<td>559</td>
<td>1822</td>
<td>77114</td>
</tr>
<tr>
<td></td>
<td>ACE-H-256</td>
<td>-</td>
<td>16</td>
<td>559</td>
<td>1822</td>
<td>256524</td>
</tr>
<tr>
<td>32-bit Cortex M3 LM359D96</td>
<td><strong>ACE</strong> permutation</td>
<td>-</td>
<td>-</td>
<td>523</td>
<td>1598</td>
<td>13003</td>
</tr>
<tr>
<td></td>
<td>ACE-_AES-128</td>
<td>0</td>
<td>16</td>
<td>559</td>
<td>1790</td>
<td>269341</td>
</tr>
<tr>
<td></td>
<td>ACE-_AES-128</td>
<td>2</td>
<td>16</td>
<td>559</td>
<td>1822</td>
<td>294988</td>
</tr>
<tr>
<td></td>
<td>ACE-H-256</td>
<td>-</td>
<td>2</td>
<td>559</td>
<td>1822</td>
<td>77114</td>
</tr>
<tr>
<td></td>
<td>ACE-H-256</td>
<td>-</td>
<td>16</td>
<td>559</td>
<td>1822</td>
<td>256524</td>
</tr>
</tbody>
</table>

59
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


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 [6] 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 65 nm ASIC.

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Permutation</th>
<th>Functionality</th>
<th>Parameters (in bits)</th>
<th>Mode of operation</th>
<th>Area [GE]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE, AE-128 and ACE-H-256</td>
<td>sLiSCP-light</td>
<td>AEAD &amp; Hash</td>
<td>320 64 128</td>
<td>Unified sLiSCP sponge</td>
<td>4250</td>
</tr>
<tr>
<td>SPIX [4]</td>
<td>sLiSCP-light-256</td>
<td>AEAD</td>
<td>256 64 128</td>
<td>Monkey Duplex</td>
<td>2611</td>
</tr>
<tr>
<td>SPOC-64 [3]</td>
<td>sLiSCP-light-192</td>
<td>AEAD</td>
<td>192 64 128</td>
<td>SPOC</td>
<td>2329</td>
</tr>
<tr>
<td>SPOC-128 [3]</td>
<td>sLiSCP-light-256</td>
<td>AEAD</td>
<td>256 128 128</td>
<td>SPOC</td>
<td>3051</td>
</tr>
</tbody>
</table>
Appendix B

Test Vectors

B.1 Simeck sbox

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

<table>
<thead>
<tr>
<th>Round</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>FFFFFFFF00000000</td>
</tr>
<tr>
<td>2</td>
<td>FFFFFFFF0000000000</td>
</tr>
<tr>
<td>3</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>4</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>5</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>6</td>
<td>FFFFFFF9A0000000000</td>
</tr>
<tr>
<td>7</td>
<td>000000000000000000</td>
</tr>
<tr>
<td>8</td>
<td>000000000000000000</td>
</tr>
</tbody>
</table>
## B.2 ACE Permutation

Test vector for ACE with all zero state.

Table B.1: Test vector for ACE permutation

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000</td>
</tr>
<tr>
<td>1</td>
<td>FFFFF7CAFFFE776 0000000000000000 0000000000000000 0000000000000000 0000000000000000</td>
</tr>
<tr>
<td>2</td>
<td>1A00151D9C4D9D9F0 25ACDA124884ECC 86A59002FB4A4BDD D9528A37DDC179A3 D4531AC03D77ADAA</td>
</tr>
<tr>
<td>3</td>
<td>446BF17FEAC5A5F 683F3428F9655013 68637D8AD32A691 F559A911A848501 E29C12724212F44E</td>
</tr>
<tr>
<td>4</td>
<td>5453CAE4EC2F4442 1229976DB44E931 62A32D34BF8A33A4 C3AAEBC56636242 38E965CBAC25E3AF</td>
</tr>
<tr>
<td>5</td>
<td>598621C1B175FD21 2D427182784D02B9 7067E7DE7730C1F EA4B2DD90656936FD C9041910726C64B</td>
</tr>
<tr>
<td>6</td>
<td>F082FCB61529AA71 7BCFD42D4AC35E52 2D5F057B73ACCE3 37961D8A276F5D05 A97265A073D111B</td>
</tr>
<tr>
<td>7</td>
<td>FEEB1A90FC6D467 F9577671E158260 8E92C0A5D6B0006F 47FF0534B9A9F53 1B675DA36BA63A5</td>
</tr>
<tr>
<td>8</td>
<td>0F9308AF36928397 C4FDAE6EBCA67C9 1F00F5716C4AFCA2 DAFC0E6A21D24D4A A4552F857DB0146</td>
</tr>
<tr>
<td>9</td>
<td>4C21336992E4423 0114A6EDC0C54B43 66FEB6383B4199ES F02194B58877636CA 3A1B1F81F1063C</td>
</tr>
<tr>
<td>10</td>
<td>2610F06C195D5056 17AE4F7D8D09471B FFBDD5A7EA4B6BCE 29C8B19379E00DB EB94BF8C4A343C5</td>
</tr>
<tr>
<td>11</td>
<td>C0DC927D4DB070E3 CF776397E9890CB5 18539159A987CD1C FCD382859F9B089 2726D3BB3C7F307</td>
</tr>
<tr>
<td>12</td>
<td>EB0021C196A1BD2A 0430040EF558DB7D BC9CEE20225F90CF AB4F7D652B79198 34B896272E3EE3E</td>
</tr>
<tr>
<td>13</td>
<td>6665AC0D976887B80 5930C950DDEB737 A1B9647843F87266 AC9EEBE137FC980E A2FF33F7DD4CEFF</td>
</tr>
<tr>
<td>14</td>
<td>A8B18A05461271D 8E5353F0E2295CA48 3A1D7E3EDDC0DEEB 3DB2755BF66616D 289C6891908FFCC9</td>
</tr>
<tr>
<td>15</td>
<td>9FEB7E5E3421167A 637EA3CF65E16677 A7C2A0F4D71079A3 05973F456E70E1C1 12D203D0B82AD26</td>
</tr>
<tr>
<td>16</td>
<td>5C39657A0560935 DC19CE947EAD5550 AC12BD91A48670E F516EB81E1DA60DA 409892A4E4CCBC15</td>
</tr>
</tbody>
</table>

## B.3 ACE-Æ-128

### Key
00111122335588DD 00111122335588DD

### Nonce
11122335588DD00 11122335588DD00

### Associated data
1122335588DD0011 1122335588DD00

### Plaintext
335588DD0011122 335588DD001111

### Ciphertext
F9362385DC213A07 CEF3F83C34CEFF

### Tag
AE85154F0242F0E4 0F9ECA3FE6967C6

## B.4 ACE-Æ-256

### Message
335588DD0011122 335588DD001111

### Hash
1676336AB5C04A1D 9225FB283172A757 A0637A6523127B83 EFC3E990BABBBD2E6
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 \textsc{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:

\[
\begin{align*}
rc_0^0 &= 00000111 = 0x07 & sc_0^0 &= 01010000 = 0x50 \\
rc_1^0 &= 01010011 = 0x53 & sc_1^0 &= 00101000 = 0x28 \\
rc_2^0 &= 01000011 = 0x43 & sc_2^0 &= 00010100 = 0x14
\end{align*}
\]
Table C.1: Generation of round and step constants for $i = 0$

<table>
<thead>
<tr>
<th>clk. cycle</th>
<th>LFSR state</th>
<th>(current) subsequence bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1 1</td>
<td>$a_9$ $a_8$ $a_7$ $a_6$ $a_5$ $a_4$ $a_3$ $a_2$ $a_1$ $a_0$</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
<td>0 0 0 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>0 1 1</td>
<td>$a_{12}$ $a_{11}$ $a_{10}$ $a_9$ $a_8$ $a_7$ $a_6$ $a_5$ $a_4$ $a_3$</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0 0 0 0 0 0 0 0 1 1 1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1</td>
<td>$a_{15}$ $a_{14}$ $a_{13}$ $a_{12}$ $a_{11}$ $a_9$ $a_8$ $a_7$ $a_6$</td>
</tr>
<tr>
<td></td>
<td>0 0</td>
<td>0 0 1 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0</td>
<td>$a_{18}$ $a_{17}$ $a_{16}$ $a_{15}$ $a_{14}$ $a_{13}$ $a_{12}$ $a_{11}$ $a_9$</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>0 0 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>4</td>
<td>0 0 0</td>
<td>$a_{21}$ $a_{20}$ $a_{19}$ $a_{18}$ $a_{17}$ $a_{16}$ $a_{15}$ $a_{14}$ $a_{13}$ $a_{12}$</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>0 1 1 0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>5</td>
<td>0 0 0</td>
<td>$a_{24}$ $a_{23}$ $a_{22}$ $a_{21}$ $a_{20}$ $a_{19}$ $a_{18}$ $a_{17}$ $a_{16}$ $a_{15}$</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
<td>0 0 0 0 0 1 1 0 0 0 0</td>
</tr>
<tr>
<td>6</td>
<td>0 0 0</td>
<td>$a_{27}$ $a_{26}$ $a_{25}$ $a_{24}$ $a_{23}$ $a_{22}$ $a_{21}$ $a_{20}$ $a_{19}$ $a_{18}$</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
<td>1 0 1 0 0 0 0 0 0 1 1 0</td>
</tr>
<tr>
<td>7</td>
<td>1 0 0</td>
<td>$a_{30}$ $a_{29}$ $a_{28}$ $a_{27}$ $a_{26}$ $a_{25}$ $a_{24}$ $a_{23}$ $a_{22}$ $a_{21}$</td>
</tr>
<tr>
<td></td>
<td>0 0</td>
<td>0 0 0 1 0 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

$\leftarrow s_{c_0}^{\phi_2}, s_{c_1}^{\phi_1}, s_{c_0}^{\phi_0}$

$rc_0^{\phi_2}$ $rc_1^{\phi_1}$ $rc_0^{\phi_0}$