WAGE: An Authenticated Cipher

Submission to the NIST LWC Competition

Submitters/Designers:
Mark Aagaard, Riham AlTawy[1], Guang Gong,
Kalikinkar Mandal*, Raghvendra Rohit, and Nusa Zidaric

*Corresponding submitter:
Email: kmandal@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/wage

September 27, 2019

[1]Currently with Department of Electrical and Computer Engineering, University of Victoria, 3800 Finnerty Rd, Victoria, BC, V8P 5C2, CANADA

Contents

List of Figures

List of Tables

Chapter 1

Introduction

WAGE is a 259-bit lightweight permutation based on the Welch-Gong (WG) stream cipher [22, 23]. It is designed to achieve an efficient hardware implementation for Authenticated Encryption with Associated Data (henceforth "AEAD"), while providing sufficient security margins. To accomplish this, the WAGE components and mode of operation are adopted from well known and analyzed cryptographic primitives. The design of WAGE, its security properties, and features are described as follows.

- WAGE **nonlinear layer:** WG permutation over $\mathbb{F}_{2^7}$ and a new 7-bit Sbox. The WG cipher, including the WG permutation, is a well-studied cryptographic primitive and has low hardware cost.
- WAGE **linear layer:** An LFSR with low hardware cost and good resistance against differential and linear cryptanalysis.
- WAGE **security:** Simple analysis and security bounds provided using automated tools such as CryptoSMT solver [17] and Gurobi [14].
- **Functionality:** Authenticated Encryption with Associated Data.
- WAGE **mode of operation:** Unified sponge duplex mode [3] that has a stronger keyed initialization and finalization phases.
- **Security claims:** Offers 128-bit security. Accepts a 128-bit key and 128-bit nonce.
- **Hardware performance:** Efficient in hardware. Achieves a throughput of 517 Mbps and has an area of 2900 GE in a 65 nm ASIC. Implementation results are presented for four ASIC libraries and two FPGAs along with parallel implementations.
- **Microcontroller performance:** WAGE is implemented on three different microcontroller platforms, namely ATmega128, MSP430F2370, and LM3S9D96 (Cotex M3). The best throughput for the permutation is achieved on LM3S9D96, which is 286.78 Kbps.

1.1 Notation

The following notation will be used throughout the document.

Notation	Description
$X \odot Y, X \oplus Y, X\|\|Y$	Bitwise AND, XOR and concatenation of X and Y
$X \otimes Y$	Finite field multiplication of X and Y
S	259 bit state of WAGE
S_j, $S_{j,k}$	stage j of state S and k-th bit of stage S_j, where $j \in \{0, \ldots, 36\}$ and $k \in \{0, \ldots, 6\}$
S_r, S_c	r-bit rate part and c-bit capacity part of S ($r = 64, c = 195$)
$\mathbb{F}_{2^7}$	Finite field $\mathbb{F}_{2^7}$
f, ω	Defining polynomial for $\mathbb{F}_{2^7}$ and its root, i.e., $f(\omega) = 0$
ℓ	LFSR feedback polynomial
WGP	Welch-Gong permutation over $\mathbb{F}_{2^7}$
SB	7-bit Sbox
rc_1^i, rc_0^i	7-bit round constants
K, N, T	key, nonce and tag
k, n, t	length of key, nonce and tag in bits ($k = n = t = 128$)
block	a 64-bit string
AD, M, C	associated data, plaintext and ciphertext (in blocks AD_i, M_i, C_i)
ℓ_X	length of X in blocks where $X \in \{AD, M, C\}$
$\widehat{K}_j, \widehat{N}_j, \widehat{T}_j$	7-bit tuple of key, nonce, and tag, $j = 0, \ldots, 17$
WAGE-$\mathcal{AE}$	WAGE authenticated encryption algorithm
WAGE-$\mathcal{E}$	WAGE encryption
WAGE-$\mathcal{D}$	WAGE decryption

1.2 Outline

The rest of the document is organized as follows. In Chapter 2, we present the complete specification of the WAGEand summarize the security claims of our submission in Chapter 3. In Chapter 4, we present the rationale of our design choices and provide the detailed security analysis in Chapter 5. 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 WAGE on microcontroller implementations. Finally, we conclude with references and test vectors in Appendix A.

Chapter 2

Specification of WAGE

2.1 WAGE AEAD Algorithm

WAGE is an iterative permutation with a state size of 259 bits inspired by the initialization phase of the Welch-Gong (WG) cipher [22, 23]. It operates in a unified duplex sponge mode [3] to offer authenticated encryption with associated data (AEAD) functionality. The AEAD algorithm (WAGE-$\mathcal{AE}$-k) processes an r-bit data per call of WAGE and is parameterized by the secret key size k. The WAGE-$\mathcal{AE}$-k consists of two algorithms, namely an authenticated encryption algorithm WAGE-$\mathcal{E}$ and a verified decryption algorithm WAGE-$\mathcal{D}$.

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

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

with

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

Decryption. 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 if the verification of tag is correct or $\perp$ (error symbol) if the tag verification fails. More formally,

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

2.2 Recommended Parameter Set

In Table 2.1, we list the recommended parameter set for WAGE-$\mathcal{AE}$-128. The length of each parameter is given in bits and d denotes the amount of allowed data (including both AD and M) before a re-keying is required.

Table 2.1: Recommended parameter set for WAGE-$\mathcal{AE}$-128

Functionality	Algorithm	r	k	n	t	$\log_2(d)$
AEAD	WAGE-$\mathcal{AE}$-128	64	128	128	128	64

2.3 Description of the WAGE Permutation

WAGE is an iterative permutation and its round function is constructed by tweaking the initialization phase of the WG cipher over $\mathbb{F}_{2^7}$ where an additional Welch-Gong permutation (WGP) and four 7-bit sboxes (SB) are added to achieve faster confusion and diffusion. We opt for a design based on a combination of an LFSR with WGP and SB, which provides a good trade-off between security and hardware efficiency. The core components of the round function are an LFSR, two WGPs and four SBs, which are described below in detail.

2.3.1 Underlying finite field

WAGE operates over the finite field $\mathbb{F}_{2^7}$, defined using the primitive polynomial $f(x) = x^7 + x^3 + x^2 + x + 1$. The elements of the finite field $\mathbb{F}_{2^7}$ are represented using the polynomial basis $\mathsf{PB} = \{1, \omega, \ldots, \omega^6\}$, and an element $a \in \mathbb{F}_{2^7}$ is given by

$$a = \sum_{i=0}^{6} a_i \omega^i, a_i \in \mathbb{F}_2$$

and its vector representation is

$$[a]_{\mathsf{PB}} = (a_0, a_1, a_2, a_3, a_4, a_5, a_6).$$

To represent a 7-bit finite field element as a byte, a 0 is appended on the left. For unambiguity, we include the conversion to binary as an intermediate step:

$$[a]_{\mathsf{PB}} = (a_0, a_1, a_2, a_3, a_4, a_5, a_6) \rightarrow [a]_b = (0, a_0, a_1, a_2, a_3, a_4, a_5, a_6) \rightarrow [a]_{hex} = (h_1, h_0)$$

Table 2.2 shows some examples of the conversion to HEX.

Table 2.2: Examples of conversion of the field elements to HEX

			2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	16^1	16^0
a	$\in$	$\mathbb{F}_{2^7}$	0	a_0	a_1	a_2	a_3	a_4	a_5	a_6	h_1	h_0
	1		0	1	0	0	0	0	0	0	4	0
	ω		0	0	1	0	0	0	0	0	2	0
1	+	ω	0	1	1	0	0	0	0	0	6	0
1	+	ω^6	0	1	0	0	0	0	0	1	4	1

2.3.2 The LFSR

The internal state S of the permutation is composed of 37 stages and given by $S = (S_{36}, \cdots, S_1, S_0)$, where each S_j holds a 7-bit word considered as an element from the finite field $\mathbb{F}_{2^7}$ represented using the PB, i.e., $S_j = (S_{j,0}, S_{j,1}, S_{j,2}, S_{j,3}, S_{j,4}, S_{j,5}, S_{j,6})$. The WAGE LFSR is defined by the feedback polynomial

$$\ell(y) = y^{37} + y^{31} + y^{30} + y^{26} + y^{24} + y^{19} + y^{13} + y^{12} + y^8 + y^6 + \omega,$$

which is primitive over $\mathbb{F}_{2^7}$. The linear feedback fb is computed as follows:

$$fb = S_{31} \oplus S_{30} \oplus S_{26} \oplus S_{24} \oplus S_{19} \oplus S_{13} \oplus S_{12} \oplus S_8 \oplus S_6 \oplus (\omega \otimes S_0).$$

2.3.3 The nonlinear components

In this subsection, we provide the details of the WGP and SB sboxes.

The Welch-Gong Permutation (WGP). The cryptographic properties of the WG permutation and transformation have been widely investigated in the literature [13]. We use a decimated WGP with low differential uniformity and high nonlinearity. Using the decimation $d = 13$, the differential uniformity for WGP is 6, and its nonlinearity is 42. The WGP7 over $\mathbb{F}_{2^7}$ is defined as

$$\mathsf{WGP7}(x) = x + (x+1)^{33} + (x+1)^{39} + (x+1)^{41} + (x+1)^{104}, x \in \mathbb{F}_{2^7}.$$

A decimated WG permutation with decimation d such that $\gcd(d, 2^m - 1) = 1$ is defined as

$$\mathsf{WGP7}(x^d) = x^d + (x^d+1)^{33} + (x^d+1)^{39} + (x^d+1)^{41} + (x^d+1)^{104}, x \in \mathbb{F}_{2^7}.$$

We use the decimation $d = 13$ and denote it by $\mathsf{WGP}(x) = \mathsf{WGP7}(x^{13})$. The maximum algebraic degree of its components is 6. An Sbox representation of WGP is given in Table 2.3 in a row-major order. The 7-bit finite field elements are represented in hex using the technique provided in Table 2.2.

Table 2.3: Hex representation of WGP

00	12	0a	4b	66	0c	48	73	79	3e	61	51	01	15	17	0e
7e	33	68	36	42	35	37	5e	53	4c	3f	54	58	6e	56	2a
1d	25	6d	65	5b	71	2f	20	06	18	29	3a	0d	7a	6c	1b
19	43	70	41	49	22	77	60	4f	45	55	02	63	47	75	2d
40	46	7d	5c	7c	59	26	0b	09	03	57	5d	27	78	30	2e
44	52	3b	08	67	2c	05	6b	2b	1a	21	38	07	0f	4a	11
50	6a	28	31	10	4d	5f	72	39	16	5a	13	04	3c	34	1f
76	1e	14	23	1c	32	4e	7b	24	74	7f	3d	69	64	62	6f

SBox (SB). We construct a lightweight 7-bit Sbox in an iterative way. Let the input be $x = (x_0, x_1, x_2, x_3, x_4, x_5, x_6)$. The nonlinear transformation Q is given by

$$Q(x_0, x_1, x_2, x_3, x_4, x_5, x_6) = (x_0 \oplus (x_2 \wedge x_3), x_1, x_2, \overline{x}_3 \oplus (x_5 \wedge x_6), x_4, \overline{x}_5 \oplus (x_2 \wedge x_4), x_6).$$

The bit permutation P is given by

$$P(x_0, x_1, x_2, x_3, x_4, x_5, x_6) = (x_6, x_3, x_0, x_4, x_2, x_5, x_1).$$

One-round R of the Sbox SB is obtained by composing the nonlinear transformation Q and the bit permutation P, and is given by $R = P \circ Q$ where

$$R(x_0, x_1, x_2, x_3, x_4, x_5, x_6) = (x_6, \overline{x}_3 \oplus (x_5 \wedge x_6), x_0 \oplus (x_2 \wedge x_3), x_4, x_2, \overline{x}_5 \oplus (x_2 \wedge x_4), x_1).$$

The 7-bit Sbox SB is constructed by iterating the function R 5 times, followed by applying Q once, and then complementing the 0th and 2nd components. Mathematically,

$$\begin{aligned}(x_0, x_1, x_2, x_3, x_4, x_5, x_6) &\leftarrow R^5(x_0, x_1, x_2, x_3, x_4, x_5, x_6)\\ (x_0, x_1, x_2, x_3, x_4, x_5, x_6) &\leftarrow Q(x_0, x_1, x_2, x_3, x_4, x_5, x_6)\\ x_0 &\leftarrow x_0 \oplus 1\\ x_2 &\leftarrow x_2 \oplus 1.\end{aligned}$$

SB has the differential uniformity of 8 and the nonlinearity of 44. The maximum algebraic degree of its components is 6.

Although SB is defined bit-wise, the interpretation of the 7 bits is identical to the interpretation of the coefficients of the finite field element represented in polynomial basis. The hex representation of SB is provided in Table 2.4 and the conversion to hex is the same as that of WGP.

2.3.4 Description of the core permutation

The WAGE permutation is a 259-bit permutation consisting of a 37-stage NLFSR defined over $\mathbb{F}_{2^7}$. It is based on the initialization phase of the WG cipher and utilizes 5 additional sboxes to update the internal state. At the i-th iteration, the internal state is denoted by $S^i = (S^i_{36}, S^i_{35}, \cdots, S^i_1, S^i_0)$. The round function that updates 6 stages of the register nonlinearly is viewed as

Table 2.4: Hex representation of SB

2e	1c	6d	2b	35	07	7f	3b	28	08	0b	5f	31	11	1b	4d
6e	54	0d	09	1f	45	75	53	6a	5d	61	00	04	78	06	1e
37	6f	2f	49	64	34	7d	19	39	33	43	57	60	62	13	05
77	47	4f	4b	1d	2d	24	48	74	58	25	5e	5a	76	41	42
27	3e	6c	01	2c	3c	4e	1a	21	2a	0a	55	3a	38	18	7e
0c	63	67	56	50	7c	32	7a	68	02	6b	17	7b	59	71	0f
30	10	22	3d	40	69	52	14	36	44	46	03	16	65	66	72
12	0e	29	4a	4c	70	15	26	79	51	23	3f	73	5b	20	5c

- Updating with initialization of the WG cipher:

  $S_{36}^{i+1} \leftarrow \mathsf{WGP}(S_{36}^i) \oplus S_{31}^i \oplus S_{30}^i \oplus S_{26}^i \oplus S_{24}^i \oplus S_{19}^i \oplus S_{13}^i \oplus S_{12}^i \oplus S_8^i \oplus S_6^i \oplus (\omega \otimes S_0^i)$

- Updating one stage with WGP:

  $S_{18}^{i+1} \leftarrow S_{19}^i \oplus \mathsf{WGP}(S_{18}^i)$

- Updating four stages with SB:

  $S_4^{i+1} \leftarrow S_5^i \oplus \mathsf{SB}(S_8^i)$

  $S_{10}^{i+1} \leftarrow S_{11}^i \oplus \mathsf{SB}(S_{15}^i)$

  $S_{23}^{i+1} \leftarrow S_{24}^i \oplus \mathsf{SB}(S_{27}^i)$

  $S_{29}^{i+1} \leftarrow S_{30}^i \oplus \mathsf{SB}(S_{34}^i)$.

A schematic diagram of the round function is presented in Figure 2.1. A pair of 7-bit round constants (rc_1, rc_0) is XORed with the pair of stages $(36, 18)$ to destroy similarity among state updates. On an input S^0, an output of the permutation is obtained by applying the round function, denoted by WAGE-StateUpdate, 111 times. An algorithmic description of WAGE is provided in Algorithm 1.

2.3.5 Round constants

We use two 7-bit round constants at each round of WAGE. The round constants are listed in Table 2.5. The interpretation of the hex values of round constants in terms of polynomial basis is the same as for SB, and hence details are omitted.

2.4 WAGE-$\mathcal{AE}$-128 Algorithm

WAGE uses the unified sponge duplex mode to provide the AEAD functionality [3]. A WAGE instance is parametrized by a key of length k, denoted as WAGE-$\mathcal{AE}$-k. Algorithm

Algorithm 1 WAGE permutation

Input : $S^0 = (S^0_{36}, S^0_{35}, \cdots, S^0_1, S^0_0)$
Output : $S^{111} = (S^{111}_{36}, S^{111}_{35}, \cdots, S^{111}_1, S^{111}_0)$

for $i = 0$ to 110 **do**:
    $S^{i+1} \leftarrow \mathsf{WAGE\text{-}StateUpdate}(S^i, rc^i_1, rc^i_0)$
return S^{111}

Function WAGE-StateUpdate(S^i):
    $fb = S^i_{31} \oplus S^i_{30} \oplus S^i_{26} \oplus S^i_{24} \oplus S^i_{19} \oplus S^i_{13} \oplus S^i_{12} \oplus S^i_8 \oplus S^i_6 \oplus (\omega \otimes S^i_0)$
    $S^{i+1}_4 \leftarrow S^i_5 \oplus \mathsf{SB}(S^i_8)$
    $S^{i+1}_{10} \leftarrow S^i_{11} \oplus \mathsf{SB}(S^i_{15})$
    $S^{i+1}_{18} \leftarrow S^i_{19} \oplus \mathsf{WGP}(S^i_{18}) \oplus rc^i_0$
    $S^{i+1}_{23} \leftarrow S^i_{24} \oplus \mathsf{SB}(S^i_{27})$
    $S^{i+1}_{29} \leftarrow S^i_{30} \oplus \mathsf{SB}(S^i_{34})$
    $S^{i+1}_{36} \leftarrow fb \oplus \mathsf{WGP}(S^i_{36}) \oplus rc^i_1$
    $S^{i+1}_j \leftarrow S^i_{j+1}, j \in \{0, \cdots, 36\} \backslash \{4, 10, 18, 23, 29, 36\}$
    return S^{i+1}

2 presents a high-level overview of WAGE-$\mathcal{AE}$-128. The encryption (WAGE-$\mathcal{E}$) and decryption (WAGE-$\mathcal{D}$) of WAGE-$\mathcal{AE}$-128 are shown in Figure 2.2. In what follows, we first illustrate the rate and the capacity part of the state, and then the padding rule. We then describe each phase of WAGE-$\mathcal{E}$ and WAGE-$\mathcal{D}$.

2.4.1 Rate and capacity part of state

The internal state S of WAGE is divided into two parts, namely the rate part S_r and the capacity part S_c. The 0-th bit of stage S_{36}, i.e., $S_{36,0}$, and all bits of stages $S_{35}, S_{34}, S_{28}, S_{27}, S_{18}, S_{16}, S_{15}, S_9$ and S_8 constitute S_r (shaded orange in Figure 2.3),

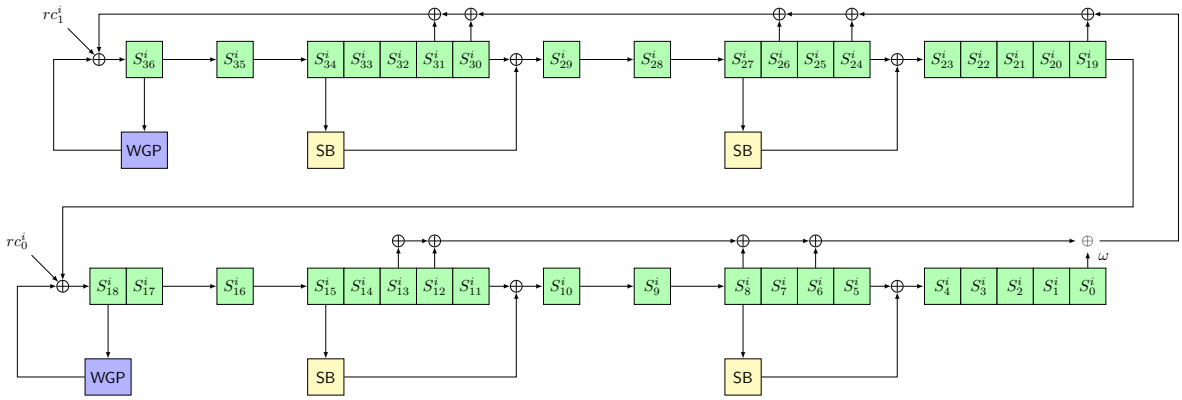


Figure 2.1: The state at i-th round of the WAGE permutation

Table 2.5: Round constants of WAGE

Round i	Round constant (rc_1^i, rc_0^i)								
0 - 9	(3f, 7f)	(0f, 1f)	(03, 07)	(40, 01)	(10, 20)	(04, 08)	(41, 02)	(30, 60)	(0c, 18)	(43, 06)
10 - 19	(50, 21)	(14, 28)	(45, 0a)	(71, 62)	(3c, 78)	(4f, 1e)	(13, 27)	(44, 09)	(51, 22)	(34, 68)
20 - 29	(4d, 1a)	(66, 73)	(5c, 39)	(57, 2e)	(15, 2b)	(65, 4a)	(79, 72)	(3e, 7c)	(2f, 5f)	(0b, 17)
30 - 39	(42, 05)	(70, 61)	(1c, 38)	(47, 0e)	(11, 23)	(24, 48)	(49, 12)	(32, 64)	(6c, 59)	(5b, 36)
40 - 49	(56, 2d)	(35, 6b)	(6d, 5a)	(7b, 76)	(5e, 3d)	(37, 6f)	(0d, 1b)	(63, 46)	(58, 31)	(16, 2c)
50 - 59	(25, 4b)	(69, 52)	(74, 3a)	(6e, 5d)	(3b, 77)	(4e, 1d)	(33, 67)	(4c, 19)	(53, 26)	(54, 29)
60 - 69	(55, 2a)	(75, 6a)	(7d, 7a)	(7f, 7e)	(1f, 3f)	(07, 0f)	(01, 03)	(20, 40)	(08, 10)	(02, 04)
70 - 79	(60, 41)	(18, 30)	(06, 0c)	(21, 43)	(28, 50)	(0a, 14)	(62, 45)	(78, 71)	(1e, 3c)	(27, 4f)
80 - 89	(09, 13)	(22, 44)	(68, 51)	(1a, 34)	(66, 4d)	(39, 73)	(2e, 5c)	(2b, 57)	(4a, 15)	(72, 65)
90 - 99	(7c, 79)	(5f, 3e)	(17, 2f)	(05, 0b)	(61, 42)	(38, 70)	(0e, 1c)	(23, 47)	(48, 11)	(12, 24)
100 - 109	(64, 49)	(59, 32)	(36, 6c)	(2d, 5b)	(6b, 56)	(5a, 35)	(76, 6d)	(3d, 7b)	(6f, 5e)	(1b, 37)
110	(46, 0d)									

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

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

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

Processing-Associated-Data(S, AD):
- $(AD_0||\cdots||AD_{\ell_{AD}-1}) \leftarrow \mathsf{pad}_r(AD)$
- **for** $i = 0$ to $\ell_{AD} - 1$ **do:**
  - $S \leftarrow (S_r \oplus AD_i, S_c \oplus 0^{c-7}||1||0^6)$
  - $S \leftarrow$ WAGE(S)
- **return** S

Encryption(S, M):
- $(M_0||\cdots||M_{\ell_M-1}) \leftarrow \mathsf{pad}_r(M)$
- **for** $i = 0$ to $\ell_M - 1$ **do:**
  - $C_i \leftarrow M_i \oplus S_r$
  - $S \leftarrow (C_i, S_c \oplus 0^{c-7}||0||1||0^5)$
  - $S \leftarrow$ WAGE(S)
- $C_{\ell_M-1} \leftarrow$ trunc-msb$(C_{\ell_M-1}, |M| \bmod r)$
- $C \leftarrow (C_0, C_1, \ldots, C_{\ell_M-1})$
- **return** (S, C)

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

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

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

Decryption(S, C):
- $(C_0||\cdots||C_{\ell_C-1}) \leftarrow \mathsf{pad}_r(C)$
- **for** $i = 0$ to $\ell_C - 2$ **do:**
  - $M_i \leftarrow C_i \oplus S_r$
  - $S \leftarrow (C_i, S_c \oplus 0^{c-7}||0||1||0^5)$
  - $S \leftarrow$ WAGE(S)
- $M_{\ell_C-1} \leftarrow S_r \oplus C_{\ell_C-1}$
- $C_{\ell_C-1} \leftarrow$ trunc-msb$(C_{\ell_C-1}, |C| \bmod r)||$trunc-lsb$(M_{\ell_C-1}, r - |C| \bmod r))$
- $M_{\ell_C-1} \leftarrow$ trunc-msb$(M_{\ell_C-1}, |C| \bmod r)$
- $M \leftarrow (M_0, M_1, \ldots, M_{\ell_C-1})$
- $S \leftarrow$ WAGE$(C_{\ell_C-1}, S_c \oplus 0^{c-7}||0||1||0^5)$
- **return** (S, M)

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

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

while all remaining bits in the state constitute S_c. The rationale for the choice of the S_r positions is explained in Section 4.7. The rate part S_r of the state is used for both

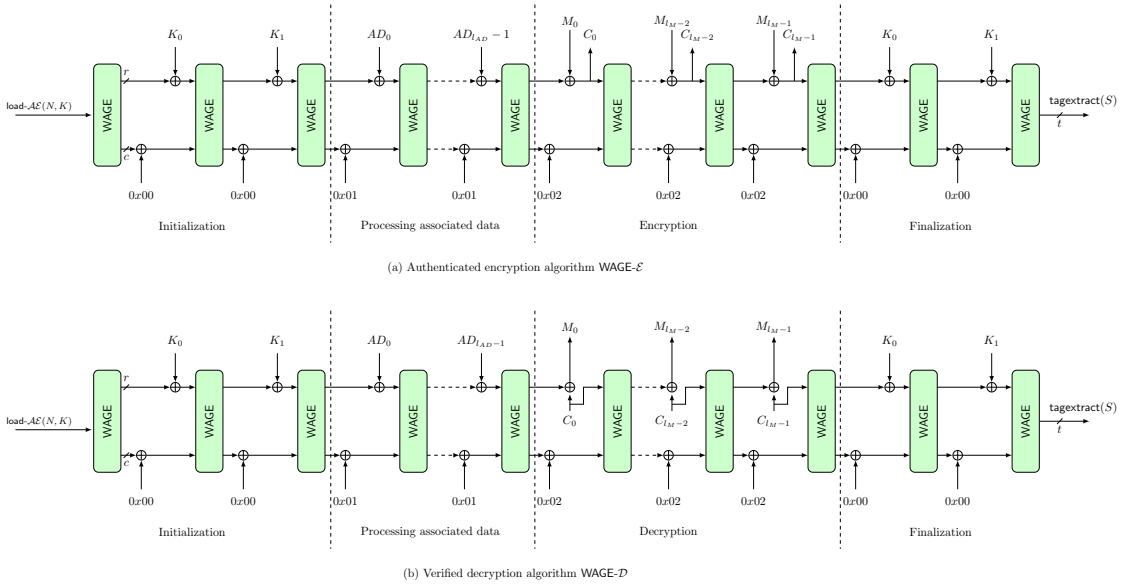


Figure 2.2: Schematic diagram of the WAGE-$\mathcal{AE}$-128 algorithm

absorbing and squeezing.

For example, the 64-bit bits of a message block are absorbed into the S_r as follows:

$$
\begin{array}{llll}
S_{36} \leftarrow (m_{63}, 0, \ldots, 0) \backsim D_9 & S_{18} \leftarrow (m_{28}, \ldots, m_{34}) \backsim D_4 \\
S_{35} \leftarrow (m_{56}, \ldots, m_{62}) \backsim D_8 & S_{16} \leftarrow (m_{21}, \ldots, m_{27}) \backsim D_3 \\
S_{34} \leftarrow (m_{49}, \ldots, m_{55}) \backsim D_7 & S_{15} \leftarrow (m_{14}, \ldots, m_{20}) \backsim D_2 \\
S_{28} \leftarrow (m_{42}, \ldots, m_{48}) \backsim D_6 & S_{9} \leftarrow (m_{7}, \ldots, m_{13}) \backsim D_1 \\
S_{27} \leftarrow (m_{35}, \ldots, m_{41}) \backsim D_5 & S_{8} \leftarrow (m_{0}, \ldots, m_{6}) \backsim D_0
\end{array}
$$

The tuples above labeled with D_k, $k = 0, \ldots, 9$, are used as data inputs to S_r; they carry the associated data bits, the message bits during encryption, the ciphertext bits during decryption, and the key bits during the initialization and finalization phases. Figure 2.3 shows the XORing of D_k to the appropriate stages S_j^{111}, $j \in \{36, 35, 34, 28, 27, 18, 16, 15, 9, 8\}$, shown in shaded orange. The two domain separator bits ds_1 and ds_0 are XORed to the first two bits of S_c, namely $S_{0,1}^{111}$ and $S_{0,0}^{111}$ respectively.

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 required number of 0's, is applied to the message M so that its length after

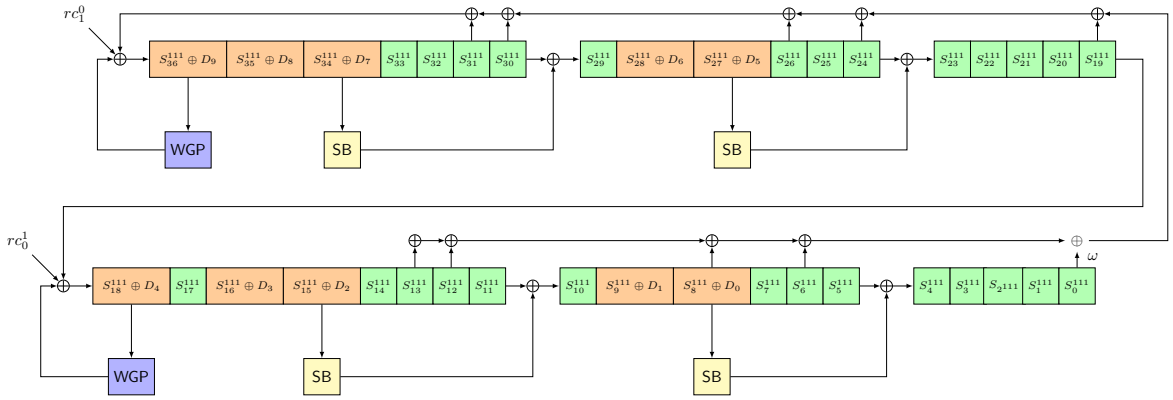


Figure 2.3: Rate (shaded orange) and capacity (green) part of WAGE-$\mathcal{AE}$-128.

padding is a multiple of r. The resulting padded message is divided into ℓ_M r-bit blocks $M_0 \| \cdots \| M_{\ell_M - 1}$. A similar procedure is carried out on the associated data AD which results in ℓ_{AD} r-bit blocks $AD_0 \| \cdots \| AD_{\ell_{AD}-1}$. In the case where no associated data is present, no processing is necessary. We summarize the padding rules for the message and associated data below.

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

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

2.4.3 Loading key and nonce

The state is loaded with 128-bit nonce $N = (n_0, \ldots, n_{127})$ and 128-bit key $K = (k_0, \ldots, k_{127})$. The remaining three bits of S are set to zero. Both the nonce and the key are divided into 7-bit tuples as follows:

- for $0 \leq i \leq 8$, $\widehat{N}_i = (n_{7i}, \ldots, n_{7i+6})$ and $\widehat{K}_i = (k_{7i}, \ldots, k_{7i+6})$
- for $9 \leq i \leq 17$, $\widehat{N}_i = (n_{7i+1}, \ldots, n_{7i+7})$ and $\widehat{K}_i = (k_{7i+1}, \ldots, k_{7i+7})$
- $\widehat{K}^*_{18} = (k_{63}, k_{127}, n_{63}, n_{127}, 0, 0, 0)$

The state S is initialized as follows:

$$\begin{aligned}
S_{36},S_{35},S_{34},S_{33},S_{32},S_{31},S_{30},S_{29},S_{28} &\leftarrow \widehat{N}_{16},\widehat{N}_{14},\widehat{N}_{12},\widehat{N}_{10},\widehat{N}_{8},\widehat{N}_{6},\widehat{N}_{4},\widehat{N}_{2},\widehat{N}_{0}\\
S_{27},S_{26},S_{25},S_{24},S_{23},S_{22},S_{21},S_{20},S_{19} &\leftarrow \widehat{K}_{17},\widehat{K}_{15},\widehat{K}_{13},\widehat{K}_{11},\hat{K}_{9},\widehat{K}_{7},\widehat{K}_{5},\widehat{K}_{3},\widehat{K}_{1}\\
S_{18},S_{17} &\leftarrow \widehat{K}^{*}_{18},\widehat{N}_{15}\\
S_{16},S_{15},S_{14},S_{13},S_{12},S_{11},S_{10},S_{9} &\leftarrow \widehat{N}_{17},\widehat{N}_{13},\widehat{N}_{11},\widehat{N}_{9}\ ,\ \widehat{N}_{7},\widehat{N}_{5},\widehat{N}_{3},\widehat{N}_{1}\\
S_{8},\ S_{7},\ S_{6},\ S_{5},\ S_{4},\ S_{3},\ S_{2},\ S_{1},\ S_{0} &\leftarrow \widehat{K}_{16},\widehat{K}_{14},\widehat{K}_{12},\widehat{K}_{10},\widehat{K}_{8},\widehat{K}_{6},\widehat{K}_{4},\widehat{K}_{2},\widehat{K}_{0}
\end{aligned}$$

This loading scheme is further discussed in Section 4.7. We use load-$\mathcal{AE}(N,K)$ to denote the process of loading the state with nonce N and key K in the positions described above.

2.4.4 Initialization

The goal of this phase is to initialize the state S with the public nonce N and the secret key K. The state is first loaded using load-$\mathcal{AE}(N,K)$ as described above, and then the two key blocks K_0 and K_1, with $K = K_0||K_1$, are absorbed into the state, with the WAGE permutation applied each time. The steps of the initialization are described as follows.

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

2.4.5 Processing associated data

If there is associated data, then for each absorbed block of AD, a domain separator bit is XORed to the current value of $S_{0,0}$. Then the WAGE permutation is applied to the whole state. This phase is defined in Algorithm 2.

$$S \leftarrow \mathsf{WAGE}(S_r \oplus AD_i, S_c \oplus 0^{c-7}||1||0^6),\ i = 0,\ldots,\ell_{AD}-1$$

2.4.6 Encryption

This phase is similar to the processing of associated data, however, the domain separator bit is XORed to the current value of $S_{0,1}$. In addition, each message block M_i, $i = 0,\ldots,\ell_M - 1$, is XORed to S_r part of the internal state as described in Section 2.4.1, which gives the corresponding ciphertext block C_i, which is extracted from the S_r part of the state as well. After that, the WAGE permutation is applied to the internal state S.

$$\begin{aligned}
C_i &\leftarrow S_r \oplus M_i,\\
S &\leftarrow \mathsf{WAGE}(C_i, S_c \oplus 0^{c-7}||0||1||0^5),\ i = 0,\cdots,\ell_M - 1
\end{aligned}$$

The last ciphertext block is truncated so that its length is equal to that of the last unpadded message block. The details of this phase are given in Algorithm 2.

2.4.7 Finalization

After the extraction of the last ciphertext block, the domain separator is reset to zero. First, the two 64-bit key blocks $K = K_0||K_1$ are absorbed into the state, with the WAGE permutation applied each time. Then, the tag is extracted from the positions of state which are used for loading the nonce during load-$\mathcal{AE}(N, K)$. The finalization steps are mentioned below and illustrated in Algorithm 2.

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

The function tagextract(S) extracts the 128-bit tag $T = \widehat{T}_0||\widehat{T}_1||\ldots||\widehat{T}_{17}||\widehat{T}^*_{18}$ from the positions that were used to load the 7-bit tuples of the nonce N during load-$\mathcal{AE}(N, K)$, namely stages S_{36} ,..., S_{28} and S_{18} ... S_9. The 7-bit $\widehat{T}_i$ tuples are given by:

$$\begin{aligned} \widehat{T}_{16},\widehat{T}_{14},\widehat{T}_{12},\widehat{T}_{10},\widehat{T}_8,\widehat{T}_6,\widehat{T}_4,\widehat{T}_2,\widehat{T}_0 &\leftarrow S_{36},S_{35},S_{34},S_{33},S_{32},S_{31},S_{30},S_{29},S_{28} \\ \widehat{T}_{15},\widehat{T}_{13},\widehat{T}_{11},\widehat{T}_9,\widehat{T}_7,\widehat{T}_5,\widehat{T}_3,\widehat{T}_1 &\leftarrow S_{16,}S_{15},S_{14},S_{13},S_{12},S_{11},S_{10},S_9 \\ \widehat{T}^*_{18},\widehat{T}_{17} &\leftarrow S_{18},S_{17} \end{aligned}$$

where

$$\begin{aligned} \widehat{T}_i &= (t_{7i}, \ldots, t_{7i+6}),\ \text{for } 0 \le i \le 17,\ \text{and} \\ \widehat{T}^*_{18} &= (-, -, t_{126}, t_{127}, -, -, -). \end{aligned}$$

Note that for $\widehat{T}^*_{18}$, only the second two bits of stage S_{18} are used, the remaining stage bits are discarded, as indicated by the sign "–".

2.4.8 Decryption

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

Chapter 3

Security Claims

WAGE is designed to provide authenticated encryption with associated data functionality. We assume a nonce-respecting adversary and do not claim security in the event of nonce reuse. If the verification procedure fails, the decrypted ciphertext and the new tag should not be given as output. Moreover, we do not claim security for the reduced-round versions of WAGE-$\mathcal{AE}$-128. The security claims of WAGE-$\mathcal{AE}$-128 are summarized in Table 3.1. Note that the security for integrity in Table 3.1 includes the integrity of nonce, plaintext and associated data.

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

Confidentiality	**Integrity**	**Authenticity**	**Data limit**
128	128	128	2^{64}

Chapter 4

Design Rationale

WAGE is a hardware-oriented $\mathcal{AE}$ scheme. Our design philosophy for the WAGE permutation is to reuse and adopt the initialization phase of the well-studied WG cipher. More specifically, we use the initialization phase of the WG cipher over $\mathbb{F}_{2^7}$. Feedback shift registers (FSR) are widely used as basic building blocks in many cryptographic designs, due to their simple architecture and efficient implementations. We choose a design for a lightweight permutation based on word-oriented shift registers and substitution boxes (sboxes).

Our parameter selection was aimed at reducing the hardware implementation cost. First, we exhaustively collected pre place-and-route (pre-PAR) synthesis results for the CMOS 65 nm area of the WGP for $\mathbb{F}_{2^m}$, $m \in \{5, 7, 8, 10, 11, 13, 14, 16\}$, and all polynomial bases, to find the balance between security and hardware implementation area. Once the field was set, we searched for the sboxes based on their hardware cost, differential uniformity and nonlinearity, and exhaustively searched for symmetric feedback polynomials with a low number of nonzero terms, and with good security properties.

4.1 Mode of Operation

WAGE adopts the sLiSCP sponge mode [3] as its mode of operation. The adopted mode is a slight variation of well analyzed traditional sponge duplex mode [4] and offers the following features.

- Provable security bounds when instantiated with an ideal permutation [5, 15].
- No key scheduling is required.
- Inverse free as the permutation is always evaluated in the forward direction.
- Encryption and decryption functionalities are identical and can be implemented with the same hardware circuit (only r-bit MUXs are required to replace the rate part of state).

- The length of processed data is not required beforehand.
- Keyed initialization and finalization phases, where the key is absorbed in the state using the XORs of the rate part. This ensures that key recovery is hard, even if the internal state is recovered. Universal forgery with the knowledge of the internal state is not practical.
- Domain separators are used for each processed data block and they are changed with each new phase, rather than with last data block in the previous phase. This leads to a more efficient hardware implementation. This method was shown to be secure in [15].

4.2 WAGE State Size

Our main aim is to choose b (state size) that provides 128-bit AE security. For a b-bit permutation with $b = r + c$ (r-bit rate and c-bit capacity), operating in sponge duplex mode, the best known bound is $\min\{2^{b/2}, 2^c, 2^k\}$ [15]. This implies that, for $k = 128$, the state size $b \geq 256$. In Section 4.4.1 we choose the operating finite field as $\mathbb{F}_{2^7}$ and accordingly $b = 259$. The values of $r = 64$ and $c = 195$ are chosen to have an efficient and low-cost hardware implementation. Our choice of (b, r, c) satisfies the NIST-LWC requirements [21] and 2^{64} bits of data can be processed per key.

4.3 Choice of Linear Layer

The linear layer of WAGE is composed of 1) $\mathcal{L}_1$: a feedback polynomial of degree 37, which is primitive over $\mathbb{F}_{2^7}$ and 2) $\mathcal{L}_2$: input and output tap positions of WGP and SB sboxes. There exist many choices for $\mathcal{L}_1$ and $\mathcal{L}_2$, which results in a tradeoff between security and efficient implementations. Thus, we restrict our search to ones which are lightweight and offer good security bounds. Note that we can not have only $\mathcal{L}_1$ or only $\mathcal{L}_2$ as the linear layer, because that would result in slower diffusion. The required criteria for $\mathcal{L}_1$ and $\mathcal{L}_2$ are:

1. To have a lightweight $\mathcal{L}_1$ we look for a feedback polynomial of the form

$$\ell(y) = y^{37} + \sum_{j=1}^{36} c_j y^j + \omega, \qquad c_j \in \mathbb{F}_2,$$

where ω is the root of the chosen field polynomial $f(x)$, which is also a primitive element of $\mathbb{F}_{2^7}$. Including ω, we chose feedback polynomials with 10 nonzero tap positions ($c_j = 1$) that are symmetric and need only 70 XOR gates to implement in hardware. In order to allow hardware optimizations in the future, e.g., parallelization, we prefer polynomials that minimize the position j of the biggest non-zero c_j. This pushes the taps as far to the right as possible, therefore we fixed the highest coefficients to zero.

2. A combination of $\mathcal{L}_1$ and $\mathcal{L}_2$ for which computing the minimum number of active sboxes is feasible and enable us to provide bounds for differential/linear distinguishers.

We found 23 symmetric polynomials with 10 non-zero taps (Table 5.1 in Section 5)[12]. The first column shows the candidate polynomials listed with their nonzero coefficients c_j. We chose the one that provides the maximum resistance against cryptanalytic attacks, such as differential and linear attacks. More precisely, we have:

$$\mathcal{L}_1 : y^{37} + y^{31} + y^{30} + y^{26} + y^{24} + y^{19} + y^{13} + y^{12} + y^8 + y^6 + \omega,$$
$$\mathcal{L}_2 : \{(36, 36), (34, 30), (27, 24), (18, 19), (15, 11), (8, 5)\}$$

where $(a, b) \in \mathcal{L}_2$ denotes the (input, output) position of an Sbox (Figure 2.1).

4.4 Nonlinear Layer of WAGE

We now justify the choices for the components in the nonlinear layer of the WAGE permutation. The nonlinear layer consists of two WGPs and four sboxes SB, specified in Section 2.3.3. The number of WGPs and sboxes was chosen to achieve faster confusion and diffusion.

4.4.1 The Welch-Gong permutation (WGP)

The natural choice of the finite field for low-cost hardware, while maintaining ease of software implementations, is $\mathbb{F}_{2^8}$. However, the pre-PAR hardware area for the WGP defined over $\mathbb{F}_{2^8}$, averaged over all irreducible polynomials, is 546 GE, which is bigger than two $\mathbb{F}_{2^7}$ WGP hardware modules. Hence we choose the finite field $\mathbb{F}_{2^7}$ for WAGE.

The polynomial basis $\mathsf{PB}_i = \{1, \omega_i, \ldots, \omega_i^6\}$ was chosen for the representation of the field elements, where ω_i is a root of the defining polynomial $f_i(x)$, i.e., $f_i(\omega_i) = 0$. The polynomial $f_i(x)$ was chosen to minimize the hardware implementation area of WGP with a decimation exponent 13 and of multiplication with the constant term of the LFSR feedback polynomial. As we use the polynomial basis, the smallest area constant term is ω_i. To estimate the area of the constant term multiplier, we used the Hamming weight of the matrix for multiplication by ω_i w.r.t. to the basis PB_i. The pre-PAR results for CMOS 65 nm implementations of the WGP modules and the constant terms are listed in Table 4.1: they show 18 primitive polynomials of degree 7, denoted $f_i(x)$. Each of the f_i has a different root ω_i, which in turn gives a different PB_i. Thus, the implementation results change with the field defining polynomial. The smallest area for WGP and constant term multiplier was found for the defining polynomial $x^7 + x^3 + x^2 + x + 1$.

4.4.2 The 7-bit sbox (SB)

The search for lightweight 7-bit sboxes varies with nonlinearity, differential uniformity and the number of rounds, balancing with small hardware cost; the sboxes explored were

in the range of 55 – 65 GE for their pre-PAR implementation area. While constructing the 7-bit sboxes, we chose the nonlinear transformations Q that have efficient hardware implementation and varied all 5040 (= 7!) bit permutations (P). The chosen Sbox SB, described in Section 2.3.3, has differential uniformity 8 and nonlinearity 44, and can be implemented with just 58 GE.

Table 4.1: Area implementation results for the defining polynomials $f_i(x)$ for $\mathbb{F}_{2^7}$

Defining polynomial $f_i(x)$	constant term area [GE]	WGP area [GE]	sum† [GE]
$x^7 + x + 1$	2	258	260
$x^7 + x^3 + 1$	16	247	263
$x^7 + x^3 + x^2 + x + 1$	10	245	255
$x^7 + x^4 + 1$	23	243	266
$x^7 + x^4 + x^3 + x^2 + 1$	22	255	277
$x^7 + x^5 + x^2 + x + 1$	24	258	282
$x^7 + x^5 + x^3 + x + 1$	6	261	267
$x^7 + x^5 + x^4 + x^3 + 1$	16	264	280
$x^7 + x^5 + x^4 + x^3 + x^2 + x + 1$	19	251	270
$x^7 + x^6 + 1$	14	270	284
$x^7 + x^6 + x^3 + x + 1$	28	248	276
$x^7 + x^6 + x^4 + x + 1$	29	261	290
$x^7 + x^6 + x^4 + x^2 + 1$	27	265	292
$x^7 + x^6 + x^5 + x^2 + 1$	16	257	273
$x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$	26	257	283
$x^7 + x^6 + x^5 + x^4 + 1$	31	259	290
$x^7 + x^6 + x^5 + x^4 + x^2 + x + 1$	20	254	274
$x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1$	14	255	269

†Combined area of constant term and WGP implementation.

4.5 Number of Rounds

Our rationale for selecting the number of rounds (say n_r) is to choose a value such that the WAGE permutation is indistinguishable from a random permutation. We now justify our choice of $n_r = 111$ as follows.

1. WAGE adopts a shift register based structure with 37 7-bit words, and hence $n_r \geq 37$, otherwise the words will not be mixed among themselves properly, which leads to meet/miss-in-the-middle attacks.

2. For $n_r = 74$, the MEDCP of WAGE equals $2^{-4\times59} = 2^{-236} > 2^{-259}$. Thus, to push the MEDCP value below 2^{-259}, $n_r \geq 74$. However, it is infeasible to compute the value of MEDCP for $n_r \geq 74$. Thus, we expect that for $n_r = 111$, MEDCP $<< 2^{-259}$ (see Section 5.1.1).

4.6 Round Constants

The round constants are added to mitigate the self-symmetry based distinguishers as mentioned in Section 2.3.5. We use a single 7-stage LFSR to generate a pair of constants at each round. Our choice of the utilized LFSR polynomial ensures that each pair of such constants does not repeat, due to the periodicity of the 8-tuple sequence constructed from the decimated m-sequence of period 127. Below we provide the details of how to generate the round constants.

4.6.1 Generation of round constants

We use an LFSR of length 7 with feedback polynomial $x^7 + x + 1$ to generate the round constants of WAGE. To construct these constants, the same LFSR is run in a 2-way parallel configuration, as illustrated in Figure 4.1. Let $\underline{a}$ denote the sequence generated by the initial state $(a_0, a_1, \ldots, a_6)$ of the LFSR without parallelization. The parallel version of this LFSR outputs two sequences, both of them using decimation exponent 2. More precisely,

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

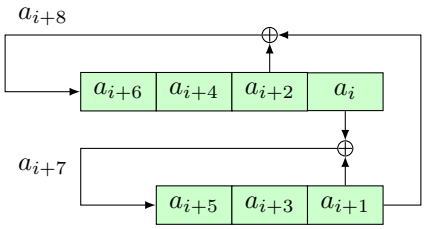


Figure 4.1: The LFSR for generating WAGE round constants.

The computation of round constants does not need any extra circuitry, but rather uses a feedback value a_{i+7} together with all 7 state bits, annotated in Figure 4.1. In

Figure 4.2 we show how the 8 consecutive sequence elements are used to generate round constants. The round constants are given by:

$$rc_0^i = a_{i+6} \| a_{i+5} \| a_{i+4} \| a_{i+3} \| a_{i+2} \| a_{i+1} \| a_i$$
$$rc_1^i = a_{i+7} \| a_{i+6} \| a_{i+5} \| a_{i+4} \| a_{i+3} \| a_{i+2} \| a_{i+1}$$

$$\overbrace{a_{i+7}, \underbrace{a_{i+6}, a_{i+5}, a_{i+4}, a_{i+3}, a_{i+2}, a_{i+1}}_{rc_0^i}, a_i}^{rc_1^i}$$

Figure 4.2: Two 7-bit round constants, generated from 8 consecutive sequence elements

We provide an example of the hex conversion of constants from LFSR sequence in Appendix A.3. The first five round constant pairs are shown in Table A.1.

4.7 Loading and Tag Extraction

The 128-bit key K and 128-bit nonce N are divided into 7-bit tuples. In software we work with bytes, and since WAGE is using 7-bit tuples, we have "left-over" bits k_{63} and n_{63}; instead of shifting all remaining key and nonce bits by 1, the bits n_{63} and k_{63} are put into the last key block $\widehat{K}_{18}^*$, which makes the loading phase and key absorption efficient for the software implementation.

Loading regions. Recall the data inputs $D_k, k = 0, \ldots 9$, in the shift register as shown in Figure 2.1. In order to minimize the hardware overhead, we reuse the data inputs D_k for loading. However, instead of XORing the D_k with previous stage content, the D_k data is fed directly into the corresponding stage. We have 10 D_k inputs, but must load the entire state, i.e., 37 stages. The stages without D_k inputs are loaded by shifting. We divide the stages without D_k inputs into loading regions, e.g., the loading region $S_8, \ldots, S_0$ can be loaded through the data input D_0 and has length 9, hence will require 9 shifts for loading. The loading region $S_8, \ldots, S_0$ is the last part of the register in Figure 2.3, and has a nonlinear input from the SB, which are disconnected during the loading. The remaining 3 SB are grounded. By inspecting the shift register, we find two other loading regions of length 9, namely region $S_{27}, \ldots, S_{19}$ (loaded through D_5) and region $S_{36}, \ldots, S_{28}$ (loaded through D_9). We decided to split the remaining 10 consecutive stages into two regions, one of length 8 and another of length 2. The region of length 8 are the stages $S_{16}, \ldots, S_9$, loaded through D_3, while the region of length 2 consists of stages S_{18}, S_{17} and are loaded through D_4. Note that there is no need to disconnect the two WGPbecause they are automatically disabled by loading through D_9 and D_4.

Loading sequence. The five loading regions, annotated with D_k used for loading, are listed below in a way that reflects their respective lengths. The $\widehat{K}_i$ and $\widehat{N}_i$ tuples

on the right show the contents of the stages S_j after the loading is complete. The notations on the top denote the 64-bit loading blocks KN_t. They are the formed by lumping together tuples appearing in the same column. For example, during the first shift we load the 64-bit block $KN_0 = \widehat{N}_0||\widehat{K}_1||0^7||0^7||\widehat{K}_0$ and during the last shift the block $KN_8 = \widehat{N}_{16}||\widehat{K}_{17}||\widehat{K}^*_{18}||\widehat{N}_{17}||\widehat{K}_{16}$.

		KN_8	KN_7	KN_6	KN_5	KN_4	KN_3	KN_2	KN_1	KN_0
$S_{36},S_{35},S_{34},S_{33},S_{32},S_{31},S_{30},S_{29},S_{28}$	$\leftarrow^{D_9}$	$\widehat{N}_{16}$,	$\widehat{N}_{14}$,	$\widehat{N}_{12}$,	$\widehat{N}_{10}$,	$\widehat{N}_8$,	$\widehat{N}_6$,	$\widehat{N}_4$,	$\widehat{N}_2$,	$\widehat{N}_0$
$S_{27},S_{26},S_{25},S_{24},S_{23},S_{22},S_{21},S_{20},S_{19}$	$\leftarrow^{D_5}$	$\widehat{K}_{17}$,	$\widehat{K}_{15}$,	$\widehat{K}_{13}$,	$\widehat{K}_{11}$,	$\widehat{K}_9$,	$\widehat{K}_7$,	$\widehat{K}_5$,	$\widehat{K}_3$,	$\widehat{K}_1$
S_{18},S_{17}	$\leftarrow^{D_4}$	$\widehat{K}^*_{18}$,	N_{15}							
$S_{16},S_{15},S_{14},S_{13},S_{12},S_{11},S_{10},S_9$	$\leftarrow^{D_3}$	$\widehat{N}_{17}$,	$\widehat{N}_{13}$,	$\widehat{N}_{11}$,	$\widehat{N}_9$,	$\widehat{N}_7$,	$\widehat{N}_5$,	$\widehat{N}_3$,	$\widehat{N}_1$	
S_8, S_7, S_6, S_5, S_4, S_3, S_2, S_1, S_0	$\leftarrow^{D_0}$	$\widehat{K}_{16}$,	$\widehat{K}_{14}$,	$\widehat{K}_{12}$,	$\widehat{K}_{10}$,	$\widehat{K}_8$,	$\widehat{K}_6$,	$\widehat{K}_4$,	$\widehat{K}_2$,	$\widehat{K}_0$

The entire loading process for regions $S_{18}, \ldots, S_9$ and $S_8, \ldots, S_0$ is shown in Table 4.2. The table shows the shifting of data through the registers in 9 shifts. The first column shows which KN_t is sent to the D_k inputs during the shift $t+1$. The stages are shown in the second row of Table 4.2, and the values "-" in the table denote the old, unknown values, which will be overwritten by the specified $\widehat{K}_i$ and $\widehat{N}_i$ blocks by the time the loading is finished. The state of stages $S_{18}, \ldots, S_0$ after shifting 9 times, i.e., after the loading is finished, is visible from the last row.

Table 4.2: Loading into the shift register through data inputs D_4, D_3 and D_0

KN_t block	shift count	D_4 S_{18},S_{17}	D_3 $S_{16}, S_{15}, S_{14}, S_{13},S_{12},S_{11},S_{10},S_9$	D_0 S_8, S_7, S_6, S_5, S_4, S_3, S_2, S_1, S_0
KN_0	1	- -	- - - - - - - -	$\widehat{K}_0$ - - - - - - - -
KN_1	2	- -	$\widehat{N}_1$ - - - - - - -	$\widehat{K}_2$, $\widehat{K}_0$ - - - - - - -
KN_2	3	- -	$\widehat{N}_3$, $\widehat{N}_1$ - - - - - -	$\widehat{K}_4$, $\widehat{K}_2$, $\widehat{K}_0$ - - - - - -
KN_3	4	- -	$\widehat{N}_5$, $\widehat{N}_3$, $\widehat{N}_1$ - - - - -	$\widehat{K}_6$, $\widehat{K}_4$, $\widehat{K}_2$, $\widehat{K}_0$ - - - - -
KN_4	5	- -	$\widehat{N}_7$, $\widehat{N}_5$, $\widehat{N}_3$, $\widehat{N}_1$ - - - -	$\widehat{K}_8$, $\widehat{K}_6$, $\widehat{K}_4$, $\widehat{K}_2$, $\widehat{K}_0$ - - - -
KN_5	6	- -	$\widehat{N}_9$, $\widehat{N}_7$, $\widehat{N}_5$, $\widehat{N}_3$, $\widehat{N}_1$ - - -	$\widehat{K}_{10}$,$\widehat{K}_8$, $\widehat{K}_6$, $\widehat{K}_4$, $\widehat{K}_2$,$\widehat{K}_0$ - - -
KN_6	7	- -	$\widehat{N}_{11}$,$\widehat{N}_9$, $\widehat{N}_7$, $\widehat{N}_5$, $\widehat{N}_3$, $\widehat{N}_1$ - -	$\widehat{K}_{12}$,$\widehat{K}_{10}$,$\widehat{K}_8$, $\widehat{K}_6$, $\widehat{K}_4$,$\widehat{K}_2$,$\widehat{K}_0$ - -
KN_7	8	$\widehat{N}_{15}$ -	$\widehat{N}_{13}$,$\widehat{N}_{11}$,$\widehat{N}_9$, $\widehat{N}_7$, $\widehat{N}_5$, $\widehat{N}_3$, $\widehat{N}_1$ -	$\widehat{K}_{14}$,$\widehat{K}_{12}$,$\widehat{K}_{10}$,$\widehat{K}_8$, $\widehat{K}_6$,$\widehat{K}_4$,$\widehat{K}_2$,$\widehat{K}_0$ -
KN_8	9	$\widehat{K}_{18}\widehat{N}_{15}$	$\widehat{N}_{17}$,$\widehat{N}_{13}$,$\widehat{N}_{11}$,$\widehat{N}_9$, $\widehat{N}_7$, $\widehat{N}_5$, $\widehat{N}_3$, $\widehat{N}_1$	$\widehat{K}_{16}$,$\widehat{K}_{14}$,$\widehat{K}_{12}$,$\widehat{K}_{10}$,$\widehat{K}_8$,$\widehat{K}_6$,$\widehat{K}_4$,$\widehat{K}_2$,$\widehat{K}_0$

Tag extraction regions. The tag is extracted in a similar fashion, from the positions that were loaded with nonce tuples. For example, the state region $S_{16}, \ldots, S_9$, which was loaded through D_3, is extracted through the output that belongs to the D_1 input. Similarly, the state region S_{18}, S_{17} is extracted through the output belonging to the D_3 input and the region $S_{36}, \ldots, S_{28}$ through the output belonging to the D_6 input. The

longest tag extraction region is also of length 9. Similar to KN_t for the loading, the 7-bit tuples extracted during shift $t+1$ are lumped into a tag-extract block TE_t .

4.8 Choice of Rate Positions

The internal state constitutes of a rate part and a capacity part in which the adversary has freedom to inject messages into the state through the rate part. The rate positions in the state, as given in Section 2.4.1, are chosen by considering the security and efficient hardware implementation. From a security point of view, the chosen rate positions allow the input bits to be processed by the six sboxes and diffused by the feedback polynomial as soon as possible after absorbing the message into the state, thus faster confusion and diffusion is achieved. Moreover, our choice ensures that any injected differences will activate at least two sboxes in the first two rounds. This enhances resistance to differential and linear cryptanalysis.

Exploiting the shifting property, the length of the process of updating the rate positions is minimized. The current choice of rate positions also allows an efficient loading and tag extraction within 9 consecutive clock cycles.

4.9 Relationship to WG ciphers

The WG cipher is a family of word-oriented stream ciphers based on an LFSR, a WG transformation and a WG permutation module over an extension field. The first family member, WG-29 [22], proceeded to Phase 2 of the eSTREAM competition [8]. Later, the lightweight variants WG-5 [1], WG-7 [19] and WG-8 [10] were proposed for constrained environments, e.g., RFID, and WG-16 [26, 11, 9] was proposed for 4G LTE.

We adopt the initialization phase of the WG cipher where we chose a decimated WG permutation with good cryptographic properties and tweak it to construct the round function of WAGE. Our proposed tweak brings faster confusion and diffusion in the state update. We choose the decimated WG permutation with decimation $d = 13$ for which its differential uniformity is 6 and nonlinearity 42 [20].

We make the tweak hardware efficient so that by disconnecting the second WGP module and all four SB modules, and keeping the domain separator 0, the round function of WAGE becomes identical to the WG initialization phase. So, the original WG stream cipher can be enabled for certain applications which require guaranteed randomness properties.

4.10 Statement

The authors declare that there are no hidden weaknesses in WAGE-$\mathcal{AE}$-128.

Chapter 5

Security Analysis

5.1 Security of WAGE Permutation

In this section, we analyze the security of the WAGE permutation against generic distinguishers. Formally, we show that WAGE with 111 rounds is indistinguishable from a random permutation. In the following, we denote the nonzero coefficients $c_i \in \{0, 1\}$ of a degree 37 primitive polynomial $l(y) = y^{37} + \sum_{i=1}^{36} c_i y^i + \omega \in \mathbb{F}_{2^7}$ by the vector $\vec{c}$.

5.1.1 Differential distinguishers

In WAGE, we use two distinct 7-bit sboxes namely, WGP and SB as the nonlinear components. The differential probabilities of the sboxes are $2^{-4.42}$ and 2^{-4}, respectively. To evaluate the maximum expected differential characteristic probability (MEDCP), we bound the minimum number of active sboxes using a Mixed Integer Linear Programming (MILP) model that takes as input $\vec{c}$, the position of sboxes and the number of rounds r. It then computes the minimum number of active sboxes denoted by $n_r(\vec{c})$. In Table 5.1, we list the values of $n_r(\vec{c})$ for varying $\vec{c}$ and $r \in \{37, 44, 51, 58, 74\}$.

The MEDCP is then given by:

$$\text{MEDCP} = \max(2^{-4.42}, 2^{-4})^{n_r(\vec{c})} = 2^{-4 \times n_r(\vec{c})}.$$

Note that for $r = 74$ and $\vec{c} = (31, 30, 26, 24, 19, 13, 12, 8, 6)$, we have MEDCP $= 2^{-4\times 59} = 2^{-236} > 2^{-259}$. Since, the MILP solver [14] is unable to finish for $r > 74$, we expect that for our choice of $\vec{c}$, $n_{111}(\vec{c}) \geq 65$. This is because for each additional 7 rounds, the number of active sboxes increases by at least 6 (see row 10 in Table 5.1) which implies MEDCP $\leq 2^{-260} < 2^{-259}$.

5.1.2 Diffusion behavior

To achieve full bit diffusion, i.e., each output bit of the permutation depends on all the input bits, we need at least 21 rounds. This is because the 7 bits of S_{36} is shifted to S_0 in 21 clock cycles. However, as the feedback function consists of 10 taps and all six sboxes

Table 5.1: Minimum number of active sboxes $n_r(\vec{c})$ for different primitive polynomials. Here − denotes that MILP optimization was too long and can not finish.

Primitive poly. coefficients	Rounds r				
$\vec{c}$	37	44	51	58	74
24, 23, 22, 21, 19, 6, 5, 4, 3	18	26	30	35	51
29, 27, 24, 23, 19, 11, 9, 6, 5	23	31	36	41	54
29, 28, 23, 22, 19, 11, 10, 5, 4	21	28	34	40	54
29, 28, 24, 20, 19, 11, 10, 6, 2	21	27	34	40	54
30, 28, 27, 21, 19, 12, 10, 9, 3	22	30	34	39	54
30, 29, 28, 26, 19, 12, 11, 10, 8	20	30	37	44	57
31, 25, 23, 21, 19, 13, 7, 5, 3	20	29	33	38	54
31, 26, 23, 20, 19, 13, 8, 5, 2	20	26	34	39	54
31, 28, 23, 21, 19, 13, 10, 5, 3	19	27	33	39	53
31, 30, 26, 24, 19, 13, 12, 8, 6	24	30	38	44	59
32, 25, 24, 21, 19, 14, 7, 6, 3	19	28	34	39	54
32, 29, 25, 22, 19, 14, 11, 7, 4	19	28	36	41	57
32, 29, 27, 22, 19, 14, 11, 9, 4	23	31	37	41	57
32, 29, 27, 24, 19, 14, 11, 9, 6	23	31	37	39	55
32, 30, 28, 24, 19, 14, 12, 10, 6	23	29	38	44	58
32, 31, 21, 20, 19, 14, 13, 3, 2	21	26	30	36	47
33, 27, 26, 20, 19, 15, 9, 8, 2	21	30	35	39	55
33, 29, 28, 21, 19, 15, 11, 10, 3	22	27	35	39	53
33, 30, 29, 26, 19, 15, 12, 11, 8	21	31	38	44	57
33, 31, 23, 22, 19, 15, 13, 5, 4	23	31	36	41	55
33, 31, 28, 23, 19, 15, 13, 10, 5	23	30	36	41	-
33, 31, 29, 22, 19, 15, 13, 11, 4	22	32	37	44	-
33, 31, 30, 25, 19, 15, 13, 12, 7	23	34	39	44	-

(2 WGP and 4 SB) individually have the full bit diffusion property, WAGE achieves the full bit diffusion in at most 37 rounds. Accordingly, we claim that meet/miss-in-the middle distinguishers may not cover more than 74 rounds as 74 rounds guarantee full bit diffusion in both the forward and backward directions.

5.1.3 Algebraic degree

The WGP and SB sboxes have an algebraic degree of 6. Note that if we only have WGP sbox at position S_{36} along with the feedback polynomial and exclude all other sboxes and intermediate XORs, then we get the original WG stream cipher [22]. Such a stream cipher is resistant to attacks exploiting the algebraic degree if non-linear feedback used in the initialization phase is also used in the key generation phase [25, 24].

Given that WAGE has 6 sboxes and we use nonlinear feedback for all of them, we expect that 111-round WAGE is secure against integral attacks.

5.1.4 Self-symmetry based distinguishers

WAGE employs two 7-bit round constants, rc_0 and rc_1, which are XORed to S_{36} and S_{18}, respectively. The round constant tuple is distinct for each round, i.e., $(rc_0^i, rc_1^i) \neq (rc_0^j, rc_1^j)$ for $0 \leq i, j \leq 110$ and $i \neq j$. This property ensures that all the rounds of WAGE are distinct and thwart attacks which exploit the symmetric properties of the round function [7, 18].

5.2 Security of WAGE-$\mathcal{AE}$-128

The security proofs of modes based on the sponge construction rely on the indistinguishability of the underlying permutation from a random one [4, 6, 5, 15]. In previous sections, we have shown that for 111 rounds the WAGE permutation is indistinguishable from a random permutation. Thus, the security bounds of the sponge duplex mode are applicable to WAGE-$\mathcal{AE}$-128. Moreover, we assume a nonce-respecting adversary, i.e, for a fixed K, nonce N is never repeated during encryption queries. Then, considering a data limit of 2^d, the k-bit security is achieved if $c \geq k + d + 1$ and $d \ll c/2$ [5]. The parameter set of WAGE (see Table 2.1) with actual effective capacity 193 (2 bits are lost for domain separation) satisfies this condition, and hence WAGE-$\mathcal{AE}$-128 provides 128-bit security for confidentiality, integrity and authenticity.

Chapter 6

Hardware Design And Analysis

In this chapter, we describe the hardware implementation of WAGE_module, which is a single module that supports both functionalities: authenticated encryption and verified decryption 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 WAGE_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

In this section, we describe the design principles and assumptions that we follow while implementing WAGE and WAGE_module.

1. **Multi-functionality module.** The system should support all operations, namely authenticated encryption and verified decryption for WAGE, in a single module (Figure 6.1), because lightweight applications generally cannot afford the extra area for separate modules.

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). This is agreed with the proposed lightweight cryptography hardware API's [16] 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 system's input port. We use a single-phase valid-bit protocol, where each input or output data signal is paired with a valid bit to denote when the data is valid. The receiving entity must capture the data in a single clock cycle (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, WAGE hardware implementations will be updated to match the proposed lighweight 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. It is tempting to use scan-cell flip-flops to reduce area, because these cells include a 2:1 multiplexer in the flip-flop which incurs less area than using a separate multiplexer. However, using scan cells as part of the design would prevent their insertion for fault-detection and hence, prevent the circuit from being tested for manufacturing faults. Clock gating can save area by replacing a flip-flop that has a chip-enable with a regular flip-flop and using a gated clock. Clock gating was done with ASICs through the synthesis script, no change was made to the hardware design.

6.2 Interface and Top-level Module

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

Table 6.1: Interface signals

Input signal	**Meaning**
reset	resets the state machine
i_mode	mode of operation
i_dom_sep	domain separator
i_padding	the last block is padded
i_data	input data
i_valid	valid data on i_data
Output signal	**Meaning**
o_ready	hardware is ready
o_data	output data
o_valid	valid data on o_data

Table 6.2: Modes of operation

i_mode	**Mode**	**Datapath operation**
0	WAGE-$\mathcal{E}$	Encryption
1	WAGE-$\mathcal{D}$	Decryption

WAGE-$\mathcal{AE}$-128 performs two operations, namely authenticated encryption (WAGE-$\mathcal{E}$) and verified decryption (WAGE-$\mathcal{D}$). We use the i_mode input signal to distinguish between the two operations.

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

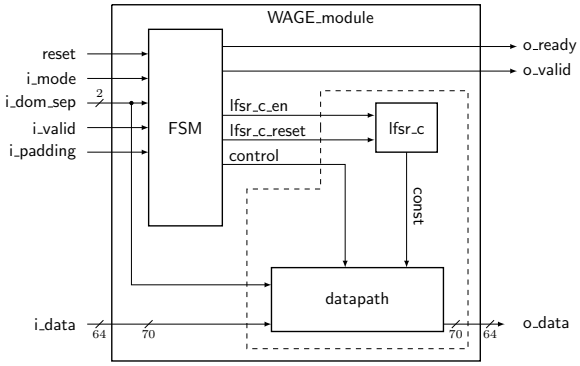


Figure 6.1: Top-level WAGE_module and the interface with the environment

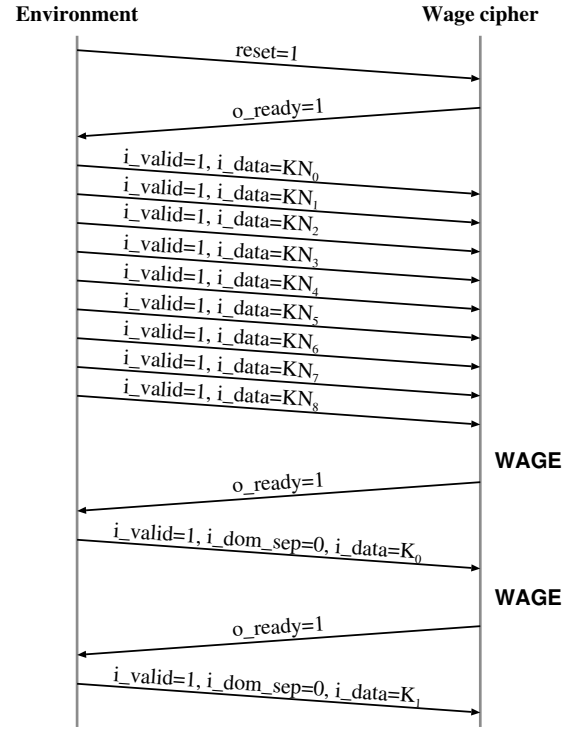
i_padding is used to indicate that the last i_data block is padded. The hardware is unaware of the lengths of individual phases, hence no internal counters for the number of processed blocks are needed. The domain separators are provided by the environment and serve as an indication of the phase change, i.e., whether the input data for WAGE-$\mathcal{AE}$-128 is the key, associated data or plaintext/ciphertext.

WAGE is specified to operate on 64-bit blocks, hence 64-bit i_data and o_data interface signals. However, since WAGE is operating over $\mathbb{F}_{2^7}$, i_data is internally fragmented into 7-bit tuples, with the last one 0-padded to form 70-bits, for the D_k, $k = 0, \ldots, 9$, signals that carry the data into the S_r stages of the internal state, as specified in Section 4.7.

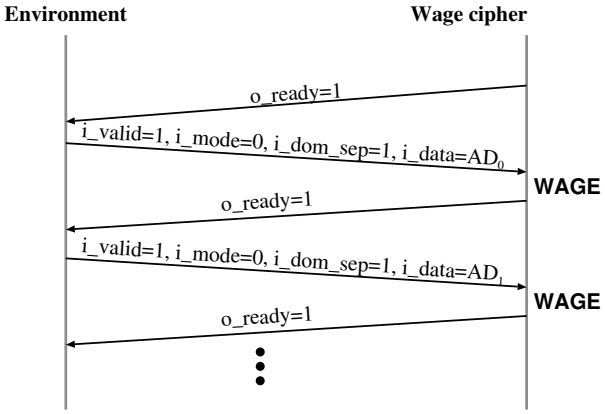
6.2.1 Interface protocol

The top-level WAGE_module is in constant interaction with the environment. We show this interaction in a form of protocol in Figures 6.2a-6.2d for the WAGE-$\mathcal{E}$ example, using the signal names from Figure 6.1. The environment is only allowed to send data to the hardware when it is in the `idle` mode, which is indicated by the hardware using the `o_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_ready` signal. Figure 6.2a shows the environment (on the left) resetting the WAGE_module (on the right), waiting for the `o_ready` to be asserted, then sending , as specified in load-$\mathcal{AE}(N, K)$ in Sections 2.4 and 4.7. The loading messages are annotated with KN_t, $t = 0, \ldots, 8$, values, the correctly fragmented and ordered key and the nonce tuples $\widehat{K}_i$ and $\widehat{N}_i$, specified in Section 4.7. Then WAGE_module starts with the initial WAGE permutation, as indicated on the right. After completing the permutation, the hardware enters idle state and asserts `o_ready` to signal to the environment that it can accept new data. The environment proceeds with key block K_0, accompanied by the proper values for `i_mode` and `i_dom_sep`. Loading and initialization is the same for both WAGE-$\mathcal{E}$ and WAGE-$\mathcal{D}$, which is why the interface signal `i_mode` is omitted from Figure 6.2a (see Table 6.2). The WAGE_module performs the WAGE permutation and asserts `o_ready`, and the environment responds with the second key block K_1.

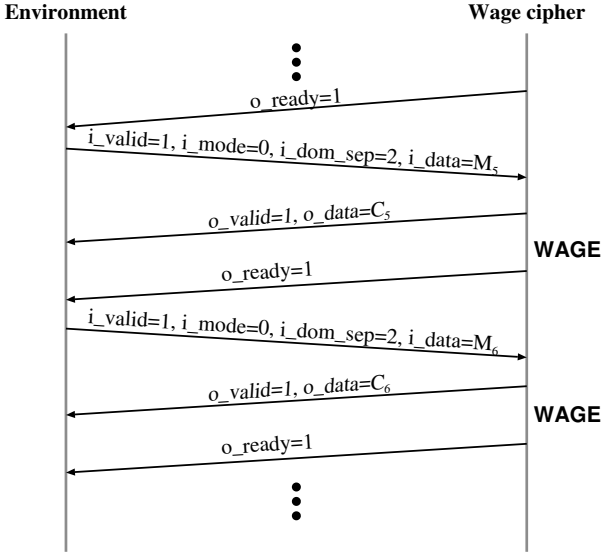
Figure 6.2b shows the communication between the environment and the hardware for



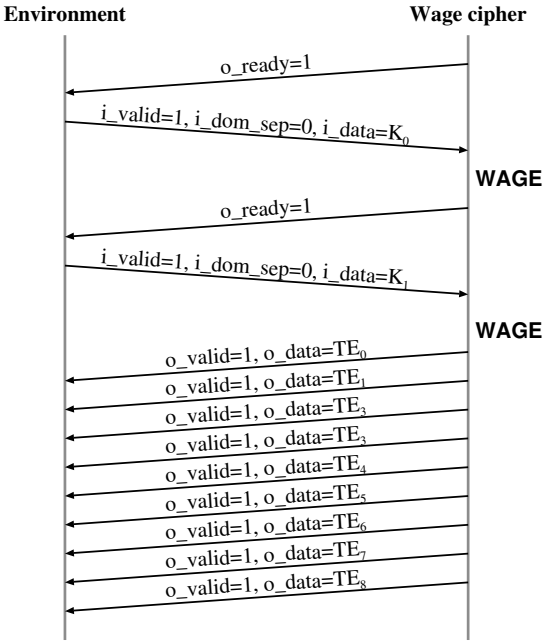
(a) Loading and initialization



(b) Processing associated data



(c) Encryption



(d) Finalization and tag generation

Figure 6.2: Interface protocol

the first two blocks of associated data AD_0 and AD_1. The environment sets `i_dom_sep` to 1 and signals the arrival of a new AD block with `i_valid`. After completing the WAGE permutation, WAGE_module asserts `o_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_dom_sep`=2 and `i_mode`=0. The WAGE_module receives M_5, encrypts it and immediately returns C_5 together with asserted `o_valid`, then starts the WAGE permutation and asserts `o_ready` upon its completion. With the exception of the tag generation, encryption (decryption) is the only phase in which WAGE_module sends data to the environment.

Each time a data block is transmitted between the environment and WAGE_module, the valid-bit protocol is used: the environment asserts `i_valid`, and WAGE_module asserts `o_valid`. This naming convention is centred around the hardware module. Another important part of the handshake between the environment and WAGE_module is the `o_ready` signal: WAGE_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 have been encrypted, finalization and tag generation begins (Figure 6.2d). The environment changes `i_dom_sep` to 0 and starts sending the key blocks. After receiving K_0, WAGE_module performs a WAGE permutation and asserts `o_ready`. After receiving K_1, WAGE_module again performs a WAGE permutation, but this time replies to the environment with 9 messages containing the tag extract blocks TE_t, $t = 0, \ldots, 8$, as specified in tagextract(S) in Sections 2.4 and 4.7. Each of the messages is accompanied by `o_valid`. Afterwards, the WAGE_module returns to idle and asserts `o_ready`.

As was mentioned before, the WAGE_module is unaware of the number of AD and M blocks, and relies on the environment to set proper values for `i_dom_sep`. However, the WAGE 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, initialization and finalization phase and the tag extraction. Another counter is needed to keep track of the WAGE permutation, which requires 111 clock cycles to complete. More details follow in Section 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 mode signal to denote which operation is to be performed (Encryption, Decryption) and the two-bit domain separator to denote the type of data being processed (associated data, message, ciphertext). The hardware uses `o_ready` to signal that it is ready to receive new data, and the environment uses `i_valid` to signal that it is sending data to the hardware.

6.2.2 Protocol timing

A more detailed representation of the events between the environment and WAGE_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 `i_mode` and `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 WAGE-$\mathcal{AE}$-128. Figure 6.3 shows the loading and the initialization up to the beginning of the second WAGE permutation; it corresponds to the upper half of the protocol in 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 WAGE state machine. The counter `count` is needed to keep track of the number of key/nonce blocks KN_t. 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 messages containing the tag-extract blocks TE_t. The counter `pcount` keeps track of the 111 clock cycles needed for one WAGE permutation. After the environment deasserts `reset`, WAGE_module enters the `Load` phase and asserts `o_ready`. The environment responds with the first message KN_0, which contains the first key and nonce tuple, accompanied with asserted `i_valid`. The WAGE_module stores the new data into its internal state and increments `count`. While `count` is running, the WAGE LFSR is shifting. Figure 6.3 shows an example when the response of the environment varies, e.g., the delay between KN_0 and KN_1 is bigger than delay between KN_1 and KN_2. After receiving KN_8, WAGE_module performs the first WAGE permutation, denoted `LoadPerm`: `o_ready` is deasserted and `pcount` increments every clock cycle. After `LoadPerm` is finished, the state machine enters the `Init` phase and `o_ready` is asserted while waiting for the first key block. The arrival of the next `i_valid` and K_0 triggers the second WAGE permutation.

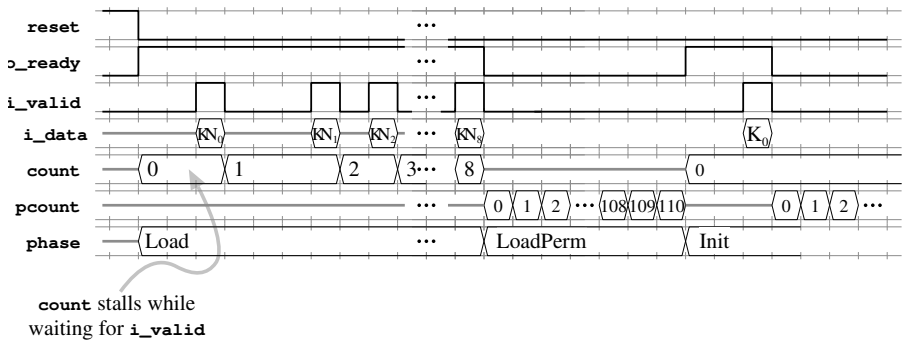


Figure 6.3: Timing diagram: loading and initialization during WAGE-$\mathcal{AE}$-128

Encryption during WAGE-$\mathcal{AE}$-128. Figure 6.4 shows the timing diagram during

the encryption of message blocks M_5 and M_6, corresponding to the 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`. After completing the previous permutation, WAGE_module asserts `o_ready`. The environment replies with a new message block M_5 accompanied by `i_valid`. The hardware immediately encrypts, returns C_5 and asserts `o_valid`. In this clock cycle the value M_5 is stored into the S_r stages of the WAGE LFSR but the LFSR is not shifted. The next clock cycle is the first round of a new WAGE permutation and `o_ready` is deasserted, indicating that the hardware is busy. When `pcount` is running, the WAGE LFSR is shifting with every clock cycle. Figure 6.4 shows WAGE_module remaining busy (`o_ready = 0`) for the duration of one WAGE permutation, then becoming idle and ready to receive new input, in this case M_6. The counter `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_data` and `o_valid`, we do not show a separate timing diagram.

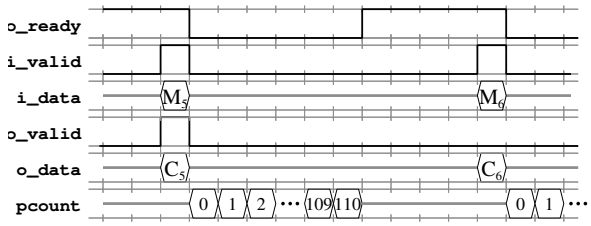


Figure 6.4: Timing diagram: encryption during WAGE-$\mathcal{AE}$-128

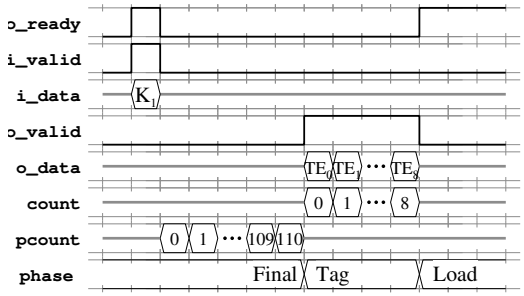


Figure 6.5: Timing of tag phase during WAGE-$\mathcal{AE}$-128

Tag phase during WAGE-$\mathcal{AE}$-128. Figure 6.5 shows a part of finalization, tag extraction, and the return of the state machine into the loading phase, which corresponds to the lower part of the protocol in Figure 6.2d. The timing diagram starts with the completion of the WAGE permutation after block K_0 was received, followed by K_1 immediately, which triggers the second WAGE permutation during finalization. This is also the last WAGE permutation of WAGE-$\mathcal{AE}$-128. After that, WAGE_module sends 9 messages with tag extract blocks TE_t to the environment. The counter `count` is used

to return WAGE_module to the `Load` phase, where it asserts `o_ready` and awaits the new key/nonce blocks KN_t.

6.2.3 Control phases

In the previous subsection, we touched on the different phases of the control circuitry. The phases are categorizations of the active connections between the WAGE LFSR and the interface signals. For simplicity we show only stages $S_4, \ldots, S_9$ of the LFSR. These stages are enough to capture all possible interactions between the environment and datapath operations. A very important notion is the shifting nature of the WAGE LFSR. The six categories, of datapath operations are shown in Figure 6.6 and described below. The roman numerals used to identify the different operations also appear in the state machine (Figure 6.9) to denote the datapath operation that is done in each transition between states.

- The first row shows `Load` (**I**). The domain separator is not used.
  - `Load` phase: the WAGE LFSR is set to shifting and the non-linear inputs, e.g., the SB, are disconnected. The `i_data` is fragmented into 7-bit tuples and loaded through the data ports D_0, D_3, D_4, D_5, D_9 as specified in Section 4.7. The data $\texttt{i\_data}_{0\ldots6}$ is loaded into stage S_8, and shifting is enabled for stages $S_8 \rightarrow S_4$. Stage S_9 is disconnected from S_8, because S_9 belongs to a different loading region — Activating the data ports D_0, D_3, D_4, D_5, D_9 has the effect of cutting the LFSR into loading regions. The data port D_9 disconnects the LFSR feedback in addition to the non-linear WGP. The values on the `i_data` port during `Load` are the key/nonce blocks KN_t, $t = 0, \ldots, 8$.

- The second row shows `Permutation` (**II**) and `Init/ProcAD/Final` (**III**). During permutation, the domain separator is not used, and otherwise its value is set to 1 for `ProcAD` and to 0 for `Init` and `Final`.
  - `Permutation` phase: the WAGE LFSR is set to shifting, but the non-linear inputs are active, e.g., the SBinput is XORed to the content of S_5 and the sum stored into S_4, as shown in **II**. All data ports D_k, $k = 0, \ldots, 9$ are disconnected from the LFSR while all the non-linear inputs, as well as the LFSR feedback, are active. Functinally, the `Permutation` phase corresponds to one round of the WAGE permutation.
  - `Init/ProcAD/Final` phase: the `i_data` is *absorbed* (XORed) to the S_r portion of the internal state S before entering the WAGE permutation, i.e., between two consecutive WAGE permutations. In **III**, we show the 7-bit tuple $\texttt{i\_data}_{0\ldots6}$ being absorbed into stage S_8. The permutation phase is treated as a single category of datapath operation from a hardware perspective, because the state machine drives the control signals with the same value, but from an algorithmic perspective this operation captures the behaviour during the initialization, processing associated data and finalization phases.

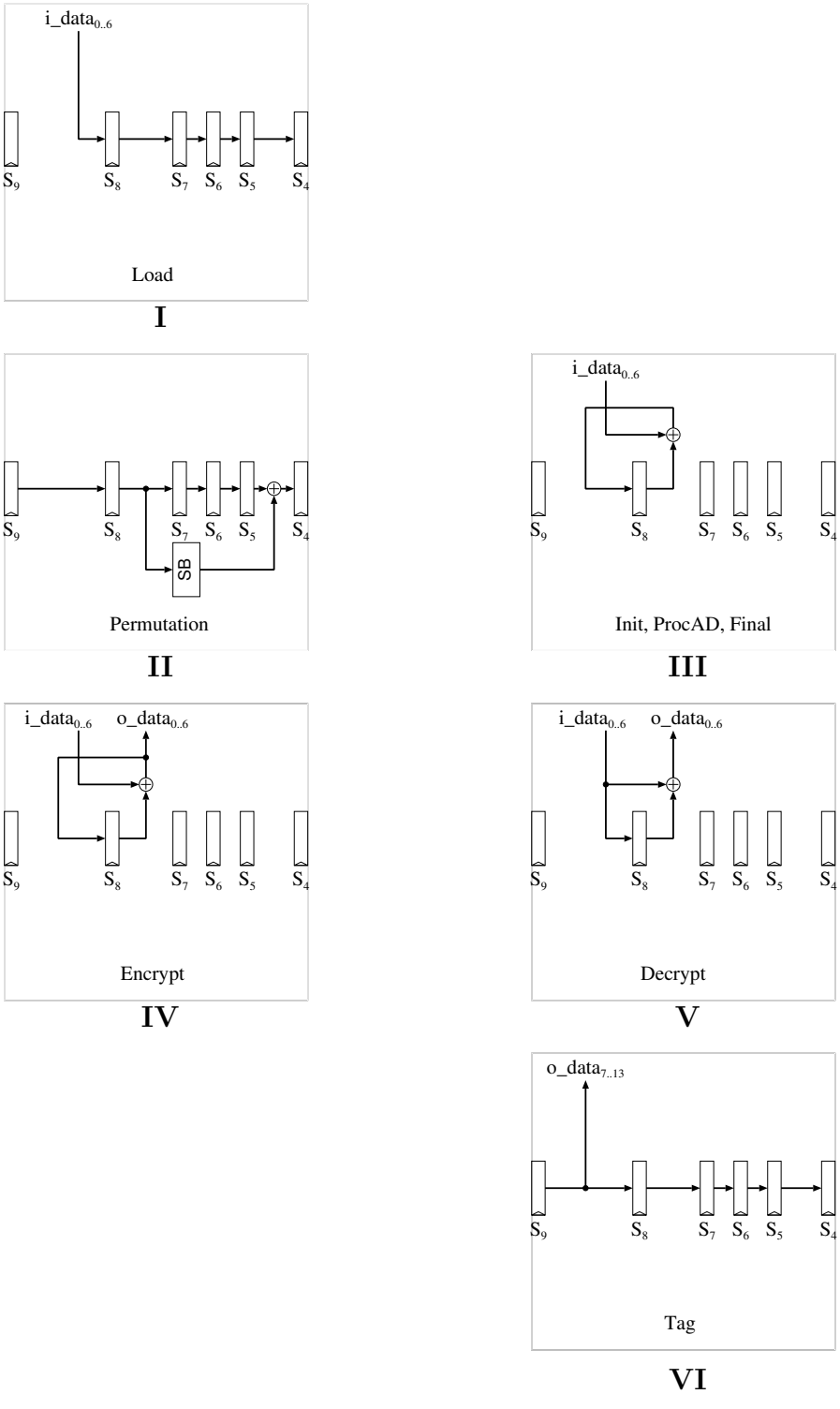


Figure 6.6: Phases and datapath operations

Only the S_r stages of the LFSR are updated during this phase, all other stages hold their previous value, as indicated by the lack of (shifting) arrows between stages.

- The third row shows `Encrypt` (**IV**) and `Decrypt` (**V**). For both phases, the domain separator is set to 2.
  - textttEncrypt phase: received `i_data` (plaintext) is XORed to the S_r stages of the internal state S and the result of this operation (ciphertext) is passed to the `o_data` output. The `i_data` (plaintext) is also *absorbed* (XORed) into the S_r stages, i.e., the resulting ciphertext is stored, and becomes a part of the internal state S when the next WAGE permutation begins. Only the S_r stages of the LFSR are updated during this phase, all other stages hold their previous value, as indicated by the lack of (shifting) arrows.
  - `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 WAGE permutation. Instead, the ciphertext from `i_data` is used to *replace* the S_r portion of the internal state before the next WAGE permutation begins. Again, only the S_r stages of the LFSR are updated during this phase, all other stages hold their previous value, as indicated by the lack of (shifting) arrows.

- The last row shows the `Tag` phase (**VI**). The domain separator is not used.
  - `Tag` phase: during tag extract, the `o_data` is extracted through outputs O_1, O_3, O_6 that belong to inputs D_1, D_3, D_6, see Section 4.7 for details. The values passed on to the `o_data` port during `Tag` are the tag-extract blocks TE_t, $t = 0, \ldots, 8$. The WAGE LFSR is shifting and the non-linear inputs, e.g., SB, are disconnected.

Note that in the datapath operations of **III, IV** and **V**, the LFSR in *not* shifting, and only the S_r stages are updated (clocked). These datpath operations require interaction with the environment. For example, in the timing diagram for encryption (Figure 6.4), the clock cycle with blocks M_5 and C_5 corresponds to the `Encrypt` phase (**III**). During the phases `Load` (**I**) and `Tag` (**VI**), WAGE LFSR is behaving as a simple shift register, with all non-linear inputs and the LFSR feedback disconnected. Only during the `Permutation` phase (**II**), is WAGE LFSR shifting and using all the non-linear and linear elements that compose the WAGE permutation. In the timing diagram for encryption (Figure 6.4), the permutation phase begins one clock cycle after the M_5 and C_5 exchange, and repeats a total of 111 times.

Table 6.3 summarizes the datapath operations (Dp op) shown in Figure 6.6, and specifies the exact input to the S_r stages of the WAGE LFSR and the output of WAGE_module for the environment, i.e., the `o_data` value.

Table 6.3: Control table for datapath based on phases from Figure 6.6

Function	i_mode	i_dom_sep	Phase	Dp op	Input to S_r stages	Output to environment
–	–	– –	Load	**I**	×	×
–	–	– –	Permutation	**II**	S_r	×
En/De-crypt	–	00	Init	**III**	$S_r \oplus$ i_data	×
En/De-crypt	–	01	ProcAD			
En/De-crypt	–	00	Final			
Encrypt	0	10	Encrypt	**IV**	$S_r \oplus$ i_data	$S_r \oplus$ i_data
Decrypt	1	10	Decrypt	**V**	i_data	$S_r \oplus$ i_data
En/De-crypt	–	– –	Tag	**VI**	×	S_{28}, S_{17}, S_9 of S

– stands for "don't care" × stands for "not used"

6.3 Hardware Implementation Details

In this section, we describe the implementation details of WAGE_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 phases from Figure 6.6. After reset is asserted, we first enter Load, followed by a single WAGE permutation in LoadPerm and initialization in Init. The choice of the phase after Init depends on the value of i_dom_sep. When i_dom_sep=1, we begin processing associated data (ProcAD), and there as long as i_dom_sep=1. Regardless of whether the current state is Init or ProcAD, i_dom_sep=2 will trigger a transition to either Encrypt or Decrypt, depending on the value of i_mode. The FSM will begin finalization (Final) when the domain separator changes to 0. The phase Final is followed by tag extraction (Tag) and then an unconditional return to the reset state.
Optimized control flow between phases. Figure 6.8 shows the optimized control flow from Figure 6.7. The optimization reduces the number of transitions in the system, which is reflected the structure of the actual state machine that is implemented (Figure 6.9). The optimized figure is annotated with transient states (small solid circles) in which the paths split or join. Transient states are syntactic sugar to reduce the number of transitions in the system. Transient states do not indicate a clock cycle boundary, while all other states do.
Summary of control flow. The high-level algorithm for WAGE (Figure 2.2) was 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

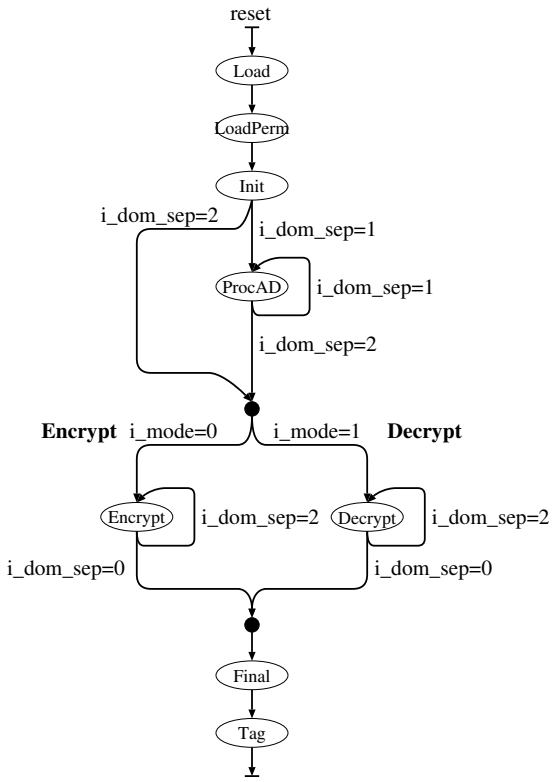


Figure 6.7: Control flow between phases

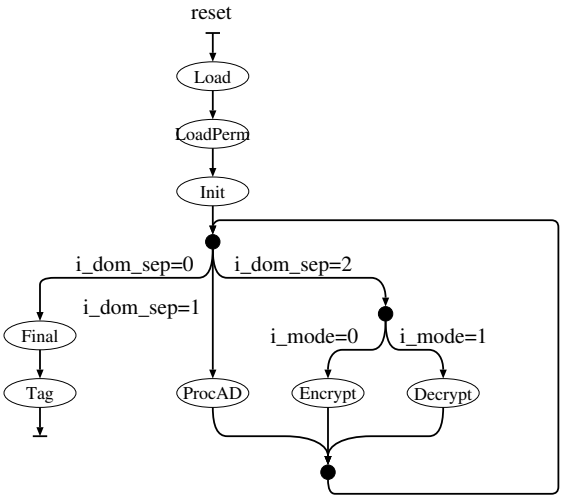


Figure 6.8: Optimized control flow between phases

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.

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., ℓ_X, $X \in \{AD, M, C\}$, which is indicated by the value change of interface signal i_mode or i_dom_sep, we show the transition to itself. While the phases Load, Init, Final and Tag also take more than one iteration, the number of iterations is fixed by the WAGE-$\mathcal{AE}$-128 algorithm 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 Figure 6.9.

Derivation of state machine from control flow. The implementation details for the control flow from Figure 6.8 are shown as a state machine in Figure 6.9. Each phase from the control flow is split into three states: iStateName, aStateName or rStateName, and pStateName, where the prefix "i" stands for "idle", "a" stands for "absorb", "r" stands for "replace" and "p" for "permutation", and the state name corresponds to the phase name. In the "idle" states, WAGE_module is waiting for new input from the environment, i.e., for i_valid=1. The "absorb" and "replace" refer to different interaction with the environment, e.g., the state aInit corresponds directly to datapath operation **III** for the initialization phase in Figure 6.6. Similarly, rDecrypt corresponds to Decryption (**VI**). In the "permutation" states, the WAGE permutation is running and WAGE_module is busy, i.e., o_ready=0. The "permutation" states directly correspond to the Permutation phase (**II**). The normal structure can be seen in the iInit, aInit and pInit states in Figure 6.9. There are a few exceptions to the normal structure:

- The Load phase receives multiple key/nonce blocks consecutively without running a permutation. Hence, in addition to iLoad and pLoad states, there is a plain Load state .
- The phase Tag can transmit multiple blocks consecutively without running a permutation. Hence, there is no need for a p-state.
- The states iProcAD, iEncrypt and iDecrypt have the same behaviour for idling, and so their idle states are merged into iAED.
- The states pProcAD, pEncrypt and pDecrypt have the same behaviour for idling, and so their idle states are merged into pAED.

Each state is annotated with three circles to denote the three bits encoding the current values on the interface signals i_mode and i_dom_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

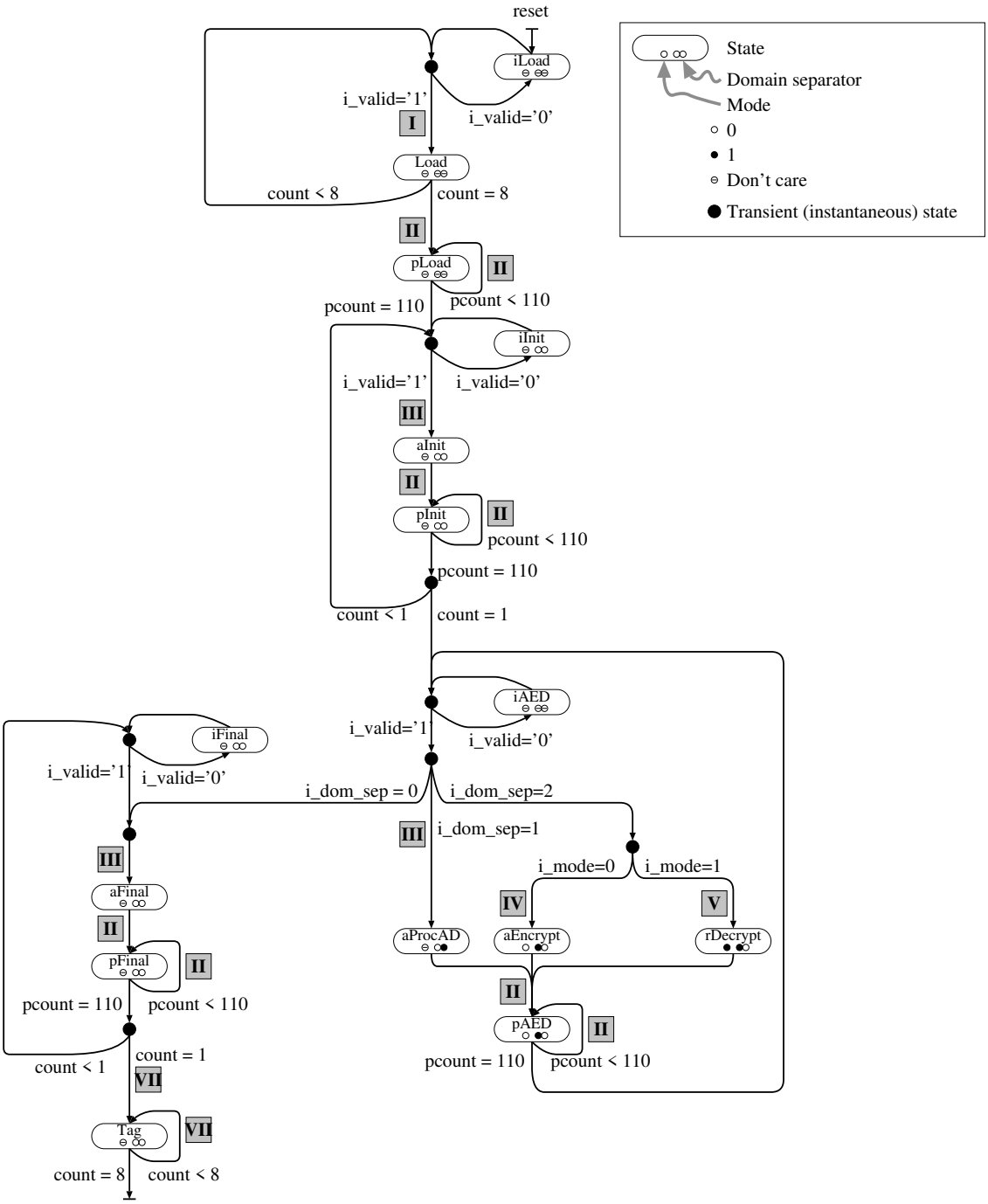


Figure 6.9: State machine

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 on the top of Figure 6.9. Asserting `reset` places the state machine into the `iLoad` idle state, where it awaits the first key/nonce block KN_0. The environment sends 9 key/nonce blocks, and the state machine uses an internal counter `count` to keep track of the loading, i.e., to keep track of iterations back to state `Load`. This behaviour was explained in the timing diagram for loading (Figure 6.3). After the key/nonce blocks are received, the state machine enters the permutations state `pLoad`, which is controlled by the counter `pcount`. After 111 rounds of `pLoad`, we first enter the `iInit` state, where we observe the normal structure: at the arrival of the next `i_valid`, we transition to `aInit`. This transition is annotated with **III**, meaning that new data was absorbed into S_r (see the datapath operations in Figure 6.6). Then we enter `pInit` 111 times (counter `pcount`). The counter `count` is used to keep track of the number of received key blocks, K_0 and K_1.

State machine: processing associated data, encryption and decryption. After the second WAGE permutation in the `Init` phase is completed, we proceed to `iAED`, a merged idle state for `iProcAD`, `iEncrypt` and `iDecrypt`. The next `i_valid` triggers the transition to `aProcAD` if `i_dom_sep=01` or to the transient state on the right if `i_dom_sep=10`, for either `aEncrypt` or `rDecrypt`, depending on `i_mode`. The transitions to the two "absorbing" states `aProcAD` and `aEncrypt` correspond to the datapath operations (**III**) and (**IV**) in Figure 6.6 respectively. The transition to the "replacing" state `rDecrypt` corresponds to (**V**). After any of these three states, the WAGE permutation runs for 111 rounds, which is shown as a merged state `pAED`. After completing a WAGE permutation, the state machine returns to the idle state `iAED`.

As an example of the correspondence between the timing diagrams and state machine behaviour, the encryption behaviour explained in Figure 6.4 corresponds to the following state transitions:

$$\texttt{iAED} \rightarrow \texttt{aEncrypt} \rightarrow \texttt{pAED} \rightarrow \ldots \rightarrow \texttt{pAED} \rightarrow \texttt{iAED}$$

The first `iAED` state in the sequence above corresponds to the clock cycles to the left of M_5 and C_5 where `o_ready`=1. When `i_valid`=1, we move to the `aEncrypt` state: this transition is annotated with the roman numeral **IV**, which tells us what the datapath does just before absorbing M_5 into S_r. Once in `aEncrypt`, we can go only to `pAED`. This transition is annotated with **II**, indicating round 0 (`pcount`=0) of the permutation, i.e., the clock cycle immediately after M_5 and C_5 column in the timing diagram (Figure 6.4). When `pcount` reaches 110, we return to `iAED`, this corresponds to the clock cycle in which `o_ready` is set anew.

State machine: finalization. When we observe `i_dom_sep=00`, the state machine will transition into `aFinal` and `pFinal` state and run the first WAGE permutation in `Final` phase. Again, counter `count` is used to keep track of the two iterations. The idle state `iFinal` is entered only once. Finally, the state machine enters the `Tag` state, and WAGE_module transmits 9 tag-extract blocks TE_t to the environment. Again,

counter `count` is used, but this time, no WAGE permutation is required. This was also explained in the protocol (Figure 6.2d) and timing diagram (Figure 6.5) for finalization.
Summary of state machine. The state machine is responsible for the `o_valid` and `o_ready` interface signals. It is also tasked with control signals for the multiplexers in the WAGE datapath, which accommodate different interactions between WAGE_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_valid='1'`. This reduces the number of idle states and facilitates combinational logic optimizations due to the uniform structure of the control flow. Loading, initialization, finalization and tag extraction all use the same one-hot counter to count their iterations. Also, all 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 The WAGE datapath

In this Section we descrbe the implementation details of the WAGE datapath.

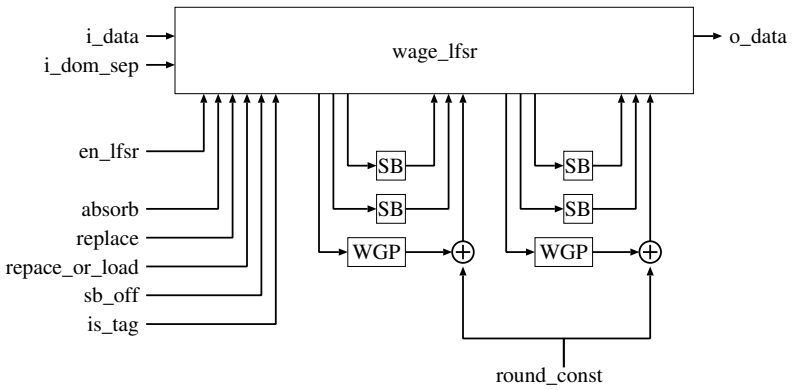


Figure 6.10: WAGE datapath

Components of WAGE datapath. Because of *the shifting nature of the LFSR*, which in turn affects loading, absorbing, replacing and tag-extraction, the WAGE datapath is explained in two levels:

1. The wage_lfsr treated as a black box in Figure 6.10.

   - wage_lfsr: The LFSR has 37 stages with 7 bits per stage, a feedback with 10 taps and a module for multiplication with ω. The internal state of wage_lfsr is also the internal state S of WAGE.
   - WGP module implementing WGP7 (Section 2.3.3): For small fields like $\mathbb{F}_{2^7}$, the WGP area, when implemented as a constant array in VHDL/Verilog, i.e., as a look-up table, is smaller than when implemented using components

such as multiplication and exponentiation to powers of two [1]. However, the WGP is not stored in hardware as a memory array, but rather as a net of AND, OR and NOT gates, derived and optimized by the synthesis tools.
- SB module (Section 2.3.3): The SB is implemented in an unrolled fashion, i.e., as purely combinational logic, composed of 5 copies of R, followed by a Q and the final two NOT gates.
- lfsr_c(Section 2.3.5): The lfsr_c for generating the round constants was implemented in a 2-way parallel fashion. It has only 7 1-bit stages and two XOR gates for the two feedback computations.

2. The extra hardware for wage_lfsr in sponge mode, i.e., the hardware allowing us to switch between different phases in Figure 6.6. Figure 6.11 shows details for stages $S_0, \ldots, S_{10}$. A smaller and less detailed portion of Figure 6.11 was shown in Figure 6.6. The grey line represents the path for normal operations during the WAGE permutation, i.e., the `Permutation` phase **II** in Figure 6.6. The additional hardware for the entire wage_lfsr is listed below, with examples referring to Figures 6.11 and datapath operations from Figure 6.6.

   - The 64-bit signal `i_data` is padded with zeros to 70 bits, then fragmented into 7-bit wage_lfsr inputs D_k, $k = 0, \ldots, 9$, corresponding to the rate stages S_r. For each data input D_k there is a corresponding 7-bit data output O_k. In Figure 6.11 we show D_1, O_1 and D_0, O_0. The input tuple $\texttt{i\_data}_{0\ldots6}$ in Figure 6.6 is loaded through the input port D_0 in Figure 6.11. Outputs O_0 and O_1 in Figure 6.11 correspond to $\texttt{o\_data}_{0\ldots6}$ and $\texttt{o\_data}_{7\ldots13}$ in Figure 6.6. The data moves to all of the S_c registers through shifting and non-linear updates.
   - 10 XOR gates must be added to the S_r stages to accommodate absorbing, encryption and decryption (**III**, **IV** and **V**). These XORs are located at stages S_9,S_8 in Figure 6.11).
   - 10 multiplexers to switch between absorbing and normal operation. In Figure 6.11, we show `Amux1` at S_9 and `Amux0` at S_8. They are needed to choose whether to shift in data from the previous stage (**I, II, VI**) or to absorb new data into the S_r stages, while the remaining stages hold their previous values (**III, IV**).
   - An XOR and a multiplexer are needed to add the domain separator into the internal state (`Amux` at S_0 in Figure 6.11).
   - To replace the contents of the S_r stages, 10 multiplexers are added. They allow us to switch between replacing (**V**) and all other datapath operations. An example of a replace multiplexer is `Rmux1` at stage S_9 in Figure 6.11.
   - Instead of additional multiplexers for loading, the existing `Rmux`k, $k = 9, 5, 4, 3, 0$, multiplexers are now controlled by `replace or load` and labelled `RLmux`k. An example is `RLmux0` on S_8 in Figure 6.11, which corresponds to the $\texttt{i\_data}_{o\ldots6}$ path in **I** and **V**.

- During the datapath operations **III**, **IV** and **V**, all non-input stages must keep their previous values, hence an enable signal `lfsr_en` is needed. It is set to 0 for phases **III**, **IV** and **V**, and to 1 in the other operations.
- Three 7-bit AND gates to turn off the inputs D_6, D_3 and D_1 (see AND at D_1 in Figure 6.11). The output O_1 is used for both tags (**VII**) as well as ciphertext (**IV**) and plaintext (**V**). The AND gates allow to turn off the input for tag extraction (**VII**) and turn on the input for encryption/decryption (**IV**/**V**).
- 4 multiplexers are needed to turn off the SB during loading and tag extraction (`SBmux` at S_4). While the non-linear inputs are used only during the `Permutation` phase (**II**), there is no need to disconnect them during phases when only the S_r stages are updated (**III, IV, V**), since `lfsr_en` prevents any other register from shifting.

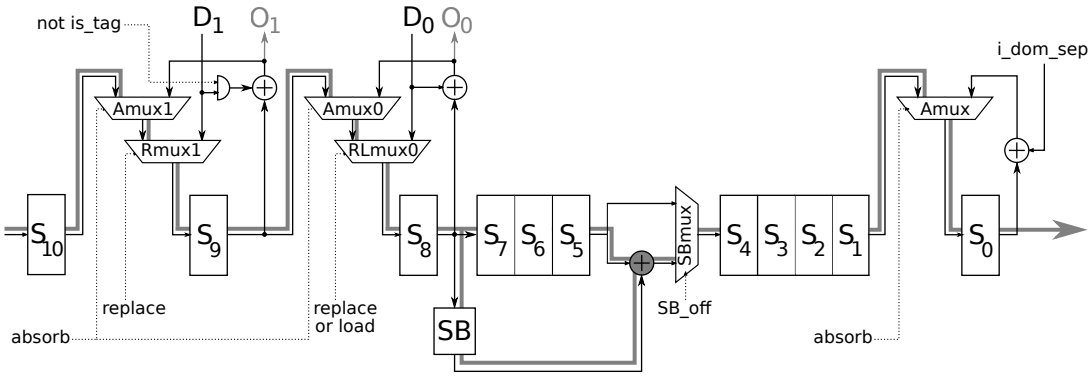


Figure 6.11: The wage_lfsr stages $S_0, \ldots, S_{10}$ with multiplexers, XOR and AND gates for the sponge mode

Control signals for multiplexers. The extra circuitry described above (and shown in Figure 6.11) needs the following control signals, which are set by the FSM:

- For `Rmux`k, `RLmux`k and `Amux`k multiplexer control: `load`, `absorb`, and `replace`. The control signal is always interpreted as follows: a value of 0 denotes the left input to the mux and a value of 1 the right input.
- For `SBmux` multiplexer control: `sb_off`. A signal value of 0 selects the bottom mux input, and value 1 the top mux input.
- For AND gate: `is_tag`

These signals are listed in Table 6.4. A final multiplexer is needed to decide whether or not the O_k outputs are sent to the environment. Instead of showing this mux, we include an extra "generate output" column in the table. The value of the control signals is determined by the interface signals `i_mode` and `i_dom_sep`, and the datapath operation. The most common operation is permutation, so for simplicity all control

signals are set to 0 in this situation. The signal `sb_off`, which turns off the non-linear S-boxes, is asserted twice: during loading and tag extraction. The S-Boxes are not used in datapath operations **III**, **IV**, and **V**, but **sb_off** is not asserted, because, as described above for `SBmux`, wage_lfsr does not shift and so the output of the S-boxes does not affect the stages.

Interface signals		**State**	**D Op**	**Generate**	**Datapath control**				
`i_mode`	`i_dom_sep`	(Fig. 6.9)	(Fig 6.6)	**output**	`load`	`sb_off`	`is_tag`	`absorb`	`replace`
-	–	`Load`	**I**	no	1	1	0	0	0
-	–	p*State*	**II**	no	0	0	0	0	0
-	00	`aInit`							
-	01	`aProcAD`	**III**	no	0	0	0	1	0
-	00	`aFinal`							
0	10	`aEncrypt`	**IV**	yes	0	0	0	1	0
1	10	`rDecrypt`	**V**	yes	0	0	0	0	1
-	–	`Tag`	**VI**	yes	0	1	1	0	0

Table 6.4: Control table for WAGE

Estimated and synthesized cost of WAGE permutation. Table 6.5 provides the estimated and actual area of the WAGE permutation for the ST Micro 65 nm ASIC library. We use an estimate of 3.75 GE for a 1-bit register and 2.00 GE for a 2:1 mux and 2-input XOR gate. We first calculate the area for the permutation without using any multiplexers or additional XORs to load inputs or support the sponge mode. The actual area for this circuit is just 2% smaller than the estimate. Next, we add the circuity to support inputs, outputs, and spong-mode. We estimated that this would require 536 GE, but the actual required area is approximately 200 GE greater, as can be seen in the relative areas for the complete datapath. 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 6.7, 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).

6.4 Hardware Implementation Results

In this section, we provide the ASIC and FPGA implementation results of WAGE permutation and WAGE_module. We first give the details of the synthesis and simulation tools and then present the implementation results.

Table 6.5: WAGE permutation hardware area estimate and implementation results

Component	**Estimate per unit** [GE]	**Count**	**Estimate per component** [GE]
State registers	3.75	259	971
Feedback XORs	2.00	70	140
Feedback multiplier			6
WGP†	258	2	516
SB†	58	4	227
constant LFSR†			45
other XORs	56	2	112
Permutation without muxes (estimate)			**2022**
Permutation without muxes (synthesized)†			1984
Absorb muxes	2.00	70	140
AED XORs	2.00	70	140
Dom-sep muxes	2.00	7	14
Dom-sep XORs	2.00	7	14
Replace muxes	2.00	70	140
Input-enable AND	1.50	21	32
Non-linear muxes	2.00	28	56
Extra circuitry for sponge mode			536
Complete datapath (estimate)			**2557**
Complete datapath (synthesized)			2753
FSM (synthesized)			228
Complete cipher (estimate)			**2786**
Complete cipher (synthesized)†			2981

† pre-PAR implementation results

Table 6.6: Tools and implementation technologies

Tools and libraries for ASICs

Logic synthesis	Synopsys Design Compiler vN-2017.09
Physical synthesis	Cadence Encounter 2014.13-s036_1
Simulation	Mentor Graphics QuestaSim 10.5c
ASIC cell libraries	65 nm STMicroelectronics CORE65LPLVT, 1.25V
	TSMC 65 nm tpfn65gpgv2od3 200c and tcbn65gplus 200a at 1.0V
	ST Microelectronics 90 nm CORE90GPLVT and CORX90GPLVT at 1.0V
	IBM 130nm CMRF8SF LPVT with SAGE-X v2.0 standard cells at 1.2V

Synthesis tools for FPGAs

Logic synthesis	Mentor Graphics Precision 64-bit 2016.1.1.28 (for Intel/Altera)
	ISE (for Xilinx)
Physical synthesis	Altera Quartus Prime 15.1.0 SJ (for Intel/Altera)
	ISE (for Xilinx)

6.4.1 Tool configuration and implementation technologies

Table 6.6 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.

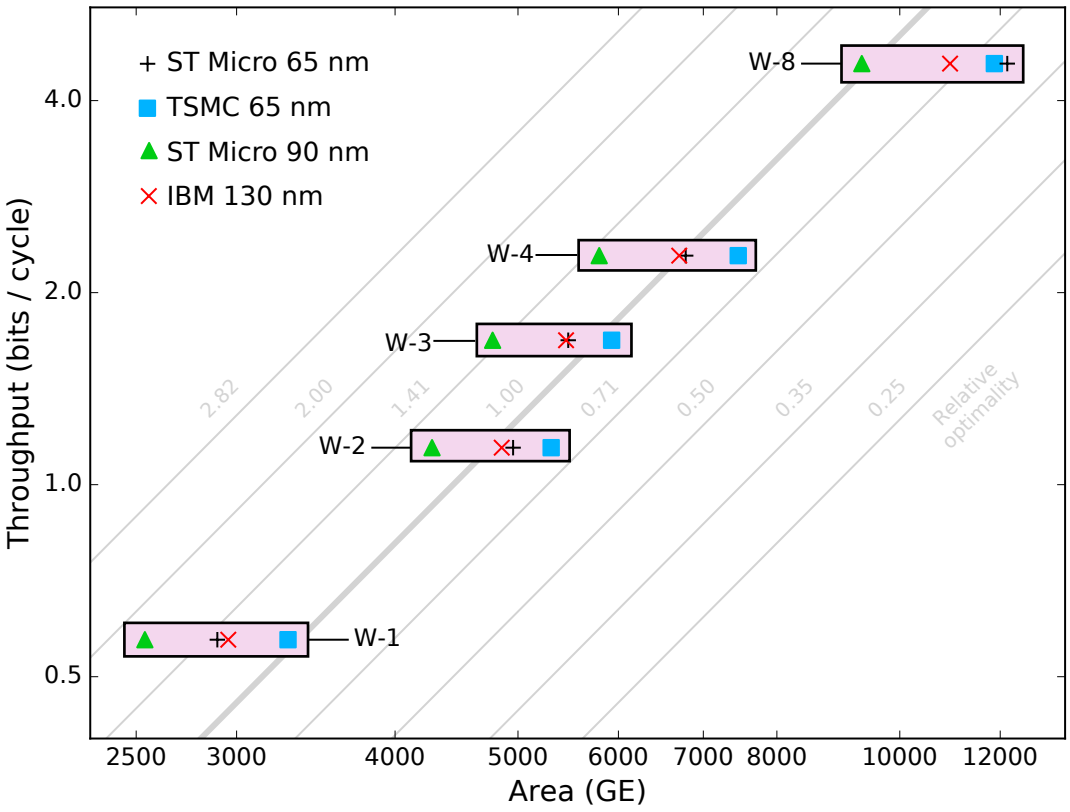
For ASICs, logic synthesis was done using the `compile_ultra` command and clock gating; and physical synthesis (place-and-route) was done with a density of 95%. By selecting a target clock speed, synthesis for ASICs can exhibit a significant range in tradeoffs between speed and area for the same RTL 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 6.12, Table 6.7, and Table 6.8 present the hardware implementation results. More details on the hardware implementation and results are available at [2].

Figure 6.12 shows area^2 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(area^2). The grey contour lines denote the relative optimality of the circuits using Tput/area^2. 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 $\log(\text{Area}^2)$.

Figure 6.12: Area^2 vs Throughput

Table 6.7 represents the same data points as Figure 6.12 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 the WAGE throughput increases, energy per bit increases, because connecting more WGPs in a combinational chain results in an exponential increase of the number of glitches, which drastically increases power consumption.

Table 6.7: ASIC implementation results

Label	Tput	ST Micro 65 nm			TSMC 65 nm			ST Micro 90 nm			IBM 130 nm		
		A	f	E*	A	f	E*	A	f	E*	A	f	E*
	[bpc]	[GE]	[MHz]	[nJ]	[GE]	[MHz]	[nJ]	[GE]	[MHz]	[nJ]	[GE]	[MHz]	[nJ]
W-1	0.57	2900	907	20.0	3290	1120	13.0	2540	940	39.2	2960	153	30.4
W-2	1.14	4960	590	19.1	5310	693	10.6	4280	493	34.4	5520	75.4	26.3
W-3	1.68	5480	397	20.4	5930	527	10.7	4770	414	31.2	5460	79.6	26.5
W-4	2.29	6780	307	24.0	7460	387	12.1	5790	277	32.9	6700	51.9	33.4
W-8	4.57	12150	192	38.5	11870	204	19.9	9330	137	49.9	10960	34.5	59.9

∗ Energy results done with timing simulation at 10 Mhz.

Table 6.8: FPGA implementation results

Module	**Extract† attribute**	**Frequency** [MHz]	**# of Slices**	**# of FFs**	**# of LUTs**
Xilinx Spartan 3 (xc3s200-5ft256)					
WAGE permutation	yes	145	139	161	168
	no	160	282	237	313
WAGE_module	yes	96	326	212	531
	no	92	455	284	699
Xilinx Spartan 6 (xc6slx9-3ftg256)					
WAGE permutation	yes	214	42	161	134
	no	218	89	237	211
WAGE_module	yes	129	144	232	367
	no	134	149	281	431

Module	**Frequency** [MHz]	**# of LC**	**# of FFs**	**# of LUTs**
Intel / Altera Stratix IV (EP4SGX70HF35M3)				
WAGE permutation	92	195	195	129
WAGE_module	73	372	372	259

† WAGE_module includes a shift register wage_lfsr and two constant array modules (WGPs) We set the attributes SHREG_EXTRACT, ROM_EXTRACT and RAM_EXTRACT to (dis)allow optimizations to shift-register configuration LUTs and Block RAMs, hence there are two sets of implementation results. When memory is inferred, 1 RAMB16 is used for Spartan 3, and 1 RAMB8BWER for Spartan 6.

Chapter 7

Software Efficiency Analysis

The WAGE permutation is designed to be efficient on heterogeneous resource constrained devices, which imposes the primitive to be efficient in hardware as well as in software. We assess the efficiency of the WAGE permutation and its modes on three different microcontroller platforms.

7.1 Software: Microcontroller

We implemented the WAGE permutation and WAGE-$\mathcal{AE}$-128 on three distinct microcontroller platforms. For WAGE-$\mathcal{AE}$-128, we implement only encryption, because decryption is the same as encryption, except updating the state with ciphertext. Our codes are written in assembly language to achieve optimal performance. We choose: 1) the Atmel ATmega128, an 8-bit microcontroller with 128 Kbytes of programmable flash memory, 4.448 Kbytes of RAM, and 32 general purpose registers of 8 bits, 2) 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 3) 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, code size (Kbytes), energy (nJ), and RAM (Kbytes) consumption.

The scheme WAGE-$\mathcal{E}$ is instantiated with a random 128-bit key and a 128-bit nonce. Note that the throughput of the WAGE-$\mathcal{E}$, which includes processing of AD/M blocks, is smaller than that of the WAGE permutation. For producing a ciphertext and a tag, $(5 + \ell)$ executions of the permutation are required where ℓ is the total number of the 64-bit data blocks including the padded associated data and plaintext. We chose two combinations of the numbers of the AD block (ℓ_{AD}) and the message block (ℓ_M), which are: 1) $(\ell_{AD}, \ell_M) = (0, 16)$, meaning empty AD and 1024-bit plaintext; and 2) $(\ell_{AD}, \ell_M) = (2, 16)$, meaning 128-bit AD and 1024-bit plaintext. Table 7.1 presents the performance of the WAGE permutation and its modes for these two choices of AD and message.

Table 7.1: Performance of WAGE on microcontrollers

Cryptographic primitive	**Platform**		**Clock freq.**	**Memory usage** [Bytes]		**Setup**	**Throughput**	**Energy/bit**
	Device	Bit	[MHz]	SRAM	Flash	[Cycles]	[Kbps]	[nJ]
WAGE Permutation	ATmega128	8	16	802	4132	19011	217.98	568
WAGE Permutation	MSP430F2370	16	16	4	5031	23524	176.16	135
WAGE Permutation	LM3S9D96	32	16	3076	5902	14450	286.78	1162
WAGE-$\mathcal{E}$ ($l_{AD} = 0, l_M = 16$)	ATmega128	8	16	808	4416	362888	45.15	2741
WAGE-$\mathcal{E}$ ($l_{AD} = 0, l_M = 16$)	MSP430F2370	16	16	46	5289	433105	37.83	628
WAGE-$\mathcal{E}$ ($l_{AD} = 0, l_M = 16$)	LM3S9D96	32	16	3084	6230	278848	58.76	5673
WAGE-$\mathcal{E}$ ($l_{AD} = 2, l_M = 16$)	ATmega128	8	16	808	4502	397260	41.24	3001
WAGE-$\mathcal{E}$ ($l_{AD} = 2, l_M = 16$)	MSP430F2370	16	16	46	5339	474067	34.56	687
WAGE-$\mathcal{E}$ ($l_{AD} = 2, l_M = 16$)	LM3S9D96	32	16	3084	6354	305284	53.67	6210

Acknowledgment

The submitters would like to thank Marat Sattarov for his help in the part of hardware implementation and Yunjie Yi for the microcontroller implementation.

Bibliography

[1] Aagaard, M. D., Gong, G., and Mota, R. K. Hardware implementations of the WG-5 cipher for passive RFID tags. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on* (2013), IEEE, pp. 29–34.

[2] Aagaard, M. D., Sattarov, M., and Zidarič, N. Hardware design and analysis of the ACE and WAGE ciphers. To appear in NIST LWC Workshop 2019. Also available at https://arxiv.org.

[3] AlTawy, R., Rohit, R., He, M., Mandal, K., Yang, G., and Gong, G. sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. In *SAC* (2017), C. Adams and J. Camenisch, Eds., Springer, pp. 129–150.

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

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

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

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

[8] eSTREAM: the ecrypt stream cipher project. http://www.ecrypt.eu.org/stream/.

[9] El-Razouk, H., Reyhani-Masoleh, A., and Gong, G. New hardware implementationsof WG($\mathbf{29}, \mathbf{11}$) and WG-$\mathbf{16}$ streamciphers using polynomial basis. *IEEE Transactions on Computers 64*, 7 (July 2015), 2020–2035.

[10] Fan, X., Mandal, K., and Gong, G. WG-8: A lightweight stream cipher for resource-constrained smart devices. In *Quality, Reliability, Security and Robustness in Heterogeneous Networks* (Berlin, Heidelberg, 2013), K. Singh and A. K. Awasthi, Eds., Springer Berlin Heidelberg, pp. 617–632.

[11] Fan, X., Zidaric, N., Aagaard, M., and Gong, G. Efficient hardware implementation of the stream cipher WG-16 with composite field arithmetic. In *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices* (New York, NY, USA, 2013), TrustED '13, ACM, pp. 21–34.

[12] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.10.0*, 2018.

[13] Gong, G., and Youssef, A. M. Cryptographic properties of the Welch-Gong transformation sequence generators. *IEEE Transactions on Information Theory 48*, 11 (Nov 2002), 2837–2846.

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

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

[16] Kap, J., Diehl, W., Tempelmeier, M., Homsirikamol, E., and Gaj, K. Hardware API for lightweight cryptography, 2019.

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

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

[19] Luo, Y., Chai, Q., Gong, G., and Lai, X. A lightweight stream cipher WG-7 for RFID encryption and authentication. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010* (Dec 2010), pp. 1–6.

[20] Mandal, K., Gong, G., Fan, X., and Aagaard, M. Optimal parameters for the WG stream cipher family. *Cryptography Commun. 6*, 2 (June 2014), 117–135.

[21] McKay, K., Bassham, L., Sönmez Turan, M., and Mouha, N. Report on lightweight cryptography (NISTIR8114), 2017.

[22] Nawaz, Y., and Gong, G. The WG stream cipher. *ECRYPT Stream Cipher Project Report 2005 33* (2005).

[23] Nawaz, Y., and Gong, G. WG: A family of stream ciphers with designed randomness properties. *Inf. Sci. 178*, 7 (Apr. 2008), 1903–1916.

[24] Rohit, R., AlTawy, R., and Gong, G. MILP-based cube attack on the reduced-round WG-5 lightweight stream cipher. In *Cryptography and Coding* (Cham, 2017), M. O'Neill, Ed., Springer International Publishing, pp. 333–351.

[25] Rønjom, S. Improving algebraic attacks on stream ciphers based on linear feedback shift register over f_{2^k}. *Des. Codes Cryptography 82*, 1-2 (2017), 27–41.

[26] Zidaric, N., Aagaard, M., and Gong, G. Hardware optimizations and analysis for the WG-16 cipher with tower field arithmetic. *IEEE Transactions on Computers 68*, 1 (Jan 2019), 67–82.

Appendix A

Test Vectors

A.1 WAGE Permutation

Input:000000000000000000000000000000000000000000000000000000000000000000000000
Output:0FA82908FEA670F1B8609F00420FC3376A52DCA922061FED7C568F785C22B4A4C

A.2 WAGE-$\mathcal{AE}$-128

Key	: 00111122335588DD 00111122335588DD
Nonce	: 111122335588DD00 111122335588DD00
Associated data	: 1122335588DD0011 1122335588DD00
Plaintext	: 335588DD00111122 335588DD001111
Ciphertext	: 4B7CD23D07D75575 5EA2ADEC4FEFF3
Tag	: D03CF7894D6D3697 C2B1758D41E78344

A.3 Round Constants Conversion

Table A.1: Generation of the first five round constant pairs (rc_1^i, rc_0^1)

clk. cycle	(current) LFSR state				(current) subsequence bits								HEX			
					a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0				
0	1	1	1	1		1	1	1	1	1	1	1	7	F	$\leftarrow$	rc_0^0
		1	1	1	0	1	1	1	1	1	1		3	F	$\leftarrow$	rc_1^0
					a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2				
1	0	1	1	1		0	0	1	1	1	1	1	1	F	$\leftarrow$	rc_0^1
		0	1	1	0	0	0	1	1	1	1		0	F	$\leftarrow$	rc_1^1
					a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4				
2	0	0	1	1		0	0	0	0	1	1	1	0	7	$\leftarrow$	rc_0^2
		0	0	1	0	0	0	0	0	1	1		0	3	$\leftarrow$	rc_1^2
					a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6				
3	0	0	0	1		0	0	0	0	0	0	1	0	1	$\leftarrow$	rc_0^3
		0	0	0	1	0	0	0	0	0	0		4	0	$\leftarrow$	rc_1^3
					a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8				
4	0	0	0			0	1	0	0	0	0	0	2	0	$\leftarrow$	rc_0^4
		1	0	0	0	0	1	0	0	0	0		1	0	$\leftarrow$	rc_1^4

The round constants are translated to HEX values as shown in Table 2.2.