Sycon v1.0
Submission to Lightweight Cryptographic Standards

Designers and Submitters:
Sumanta Sarkar[†]
Kalikinkar Mandal[‡]
Dhiman Saha[††]

[†]TCS Innovation Labs, Hyderabad, INDIA
[‡]University of Waterloo, Waterloo, CANADA
[††]Indian Institute of Technology, Bhilai, INDIA

Corresponding submitter:
Sumanta Sarkar
TCS Innovation Labs
GS3-71, Tata Consultancy Services, Deccan Park
Madhapur, Hyderabad 500 081 INDIA
Email: sumanta.sarkar1@tcs.com
Telephone: +91-40-6667-3588

March 29, 2019

Contents

Chapter 1

Introduction

NIST has taken up the initiative to standardize lightweight cryptographic algorithms that are tailored for resource constrained devices. In this regard, NIST announces call for lightweight cryptographic algorithms [2].

We present SYCON as a competitor for this standardization process. SYCON offers two authenticated encryption algorithms with associated data, and one hash algorithm in sponge constructions [7, 6, 9]. For authenticated encryption, one instance offers 128-bit security for confidentiality, integrity and associated data, and the other instance offers 112-bit security. The hash algorithm accepts a message of any length and outputs a digest of length 256 bits, and offers 128-bit collision resistance security.

At the core of SYCON is a lightweight permutation of 320 bits, called SYCON *permutation*. The design of the SYCON permutation is based on the substitution permutation network, and its components are chosen in such a way that it is featured for efficient implementations both in hardware and software. Our design is simple, provides stronger security assurance with good performances on cross-platforms, and is suitable for resource-constrained devices such as RFID tags, sensor nodes, and industrial IoT devices.

Table 1.1: Notations

$x \in \{0,1\}^n$	Binary n-tuple
$x \oplus y$	Bitwise XOR of x and y
$x\|\|y$	Concatenation of bitstrings x and y
xy	Bitwise AND of x and y
$\lfloor x \rfloor_n$	x truncated to last (LSB) n-bits
$(x <<< n)$	Left circular shift by n-bits
SB	S-box layer
$\mathbf{PL}_1$	Bit permutation of 320 symbols
$\mathbf{IPL}_1$	Inverse of $\mathbf{PL}_1$
$\mathbf{PL}_2$	Bit permutation of 320 symbols
SD	Subblock Diffusion
RC	Add Round Constant Layer
rc_i	Round constant at round i
Π^ρ	An iterated permutation with ρ rounds over $\{0,1\}^{320}$

Roadmap. The rest of the document is organized as follows. In Section 2, we provide the specification of SYCON authenticated encryption and hash algorithms. Section 3 presents the security of the SYCON algorithms, and Section 4 describes the design rationale of the SYCON parameters. In Section 5, we assess the performances of the SYCON AEAD and hash algorithms in hardware and software.

Chapter 2

Specification of Sycon

In this chapter, we provide a complete specification of the Sycon permutation and of its authenticated encryption (AE) and hash algorithms. We first present the construction of the Sycon permutation, which is designed to achieve high throughput and efficient in both hardware and software implementations. It is plugged into the MonkeyDuplex sponge mode [6] to construct a family of authenticated encryption, and used in the unified sponge mode [9] to construct a hash algorithm.

2.1 The Sycon Permutation

Sycon is an iterative permutation of 320 bits. The permutation is constructed by iterating the round function

$$R : \{0,1\}^{320} \to \{0,1\}^{320},$$

ρ times. The design of R is based on the Substitution-Permutation Network (SPN). The permutation R is composed of a sequence of four distinct transformation, namely SBox (**SB**), PLayer ($\mathbf{PL}_1$ and $\mathbf{PL}_2$), SubBlockDiffusion (**SD**), and AddRoundConst (**RC**), and it is defined by

$$R = \mathbf{PL}_2 \circ \mathbf{RC} \circ \mathbf{SD} \circ \mathbf{PL}_1 \circ \mathbf{SB}.$$

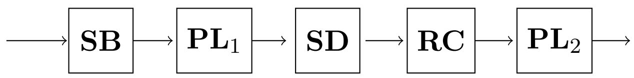

Figure 2.1: The round function

Then, a ρ-round permutation, denoted by Π^ρ, is constructed as

$$\Pi^\rho = \mathbf{PL}_1 \circ \underbrace{R \circ \cdots \circ R}_{\rho\ \text{times}} \circ \mathbf{IPL}_1,$$

where $\mathbf{IPL}_1$ is the inverse of $\mathbf{PL}_1$. Below we describe the components of R in detail.

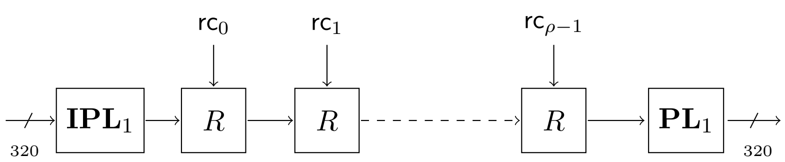

Figure 2.2: An overview of the permutation Π^ρ

2.1.1 Substitution Box: SBox (SB)

The substitution box (**SB**) provides confusion in the permutation, and is applied to the state $\mathbf{a} = a_0 a_1 \cdots a_{320}$ as follows. It arranges the 320-bit state into 64 5-bit words as

$\mathbf{b}_i = a_{5i}a_{5i+1}a_{5i+2}a_{5i+3}a_{5i+4}$, $0 \le i \le 63$, $b_i \in \{0,1\}^5$, that is $\mathbf{a} = \mathbf{b}_0\mathbf{b}_1\cdots\mathbf{b}_{63}$. Then a 5×5 S-box S, given in Table 2.1, is applied to each 5-bit word as

$$\mathbf{SB}(\mathbf{a}) = \mathrm{S}[\mathbf{b}_0]\mathrm{S}[\mathbf{b}_1]\mathrm{S}[\mathbf{b}_2]\cdots\mathrm{S}[\mathbf{b}_{63}].$$

The cryptographic properties of the S-box are summarized in Table 2.2.

Table 2.1: The 5-bit S-box

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S[x]	8	19	30	7	6	25	16	13	22	15	3	24	17	12	4	27
x	16	17	18	19	20	21	22	23	24	25	26	27	28	19	30	31
S[x]	11	0	29	20	1	14	23	26	28	21	9	2	31	18	10	5

Table 2.2: Cryptographic properties of the S-box

Differential uniformity	Algebraic degree	Fixed point	Nonlinearity	Differential branch number	Linear branch number
8	2	No	8	3	3

2.1.2 First Permutation Layer: PLayer ($\mathbf{PL}_1$)

After applying the SBox layer, we apply the permutation layer PLayer on the 320-bit state. The bit-permutation over 320 symbols is defined in Table A.1. Basically, it is obtained from $\hat{P}_j(i) = 5 * i + j$ where $0 \le i \le 63$ and $0 \le j \le 4$. For the state

$$\mathbf{a} = a_0a_1\cdots a_5a_6a_7\cdots a_{10}a_{11}a_{12}\cdots a_{160}a_{161}a_{162}\cdots a_{315}a_{316}a_{317}\cdots a_{319},$$

the permutation layer $\mathbf{PL}_1$ changes it to

$$\begin{aligned}\mathbf{PL}_1(\mathbf{a}) &= a_{\mathbf{PL}_1(0)}a_{\mathbf{PL}_1(1)}\cdots\cdots\cdots a_{\mathbf{PL}_1(319)}\\ &= a_0a_5\cdots a_{315}a_1a_6\cdots a_{316}a_2a_7\cdots a_{317}a_3a_8\cdots a_{318}a_4a_9\cdots a_{319}.\end{aligned}$$

2.1.3 Diffusion Layer: SubBlockDiffusion (SD)

The diffusion layer is a linear transformation that is applied independently on five 64-bit subblocks constructed from the state. Given a state

$$\mathbf{a} = a_0a_1\cdots a_5a_6a_7\cdots a_{10}a_{11}a_{12}\cdots a_{160}a_{161}a_{162}\cdots a_{315}a_{316}a_{317}\cdots a_{319},$$

it first divides the state into five 64-bit blocks as $\mathbf{a} = Y_0\|Y_1\|Y_2\|Y_3\|Y_4$ and then five distinct diffusion transformations are applied on Y_i's. The diffusion layer on $\mathbf{a} = Y_0\|Y_1\|Y_2\|Y_3\|Y_4$ is defined as

$$\begin{aligned}Z_0 &\leftarrow Y_0 \oplus (Y_0 \lll 11) \oplus (Y_0 \lll 22)\\ Z_1 &\leftarrow Y_1 \oplus (Y_1 \lll 13) \oplus (Y_1 \lll 26)\\ Z_2 &\leftarrow Y_2 \oplus (Y_2 \lll 31) \oplus (Y_2 \lll 62)\\ Z_3 &\leftarrow Y_3 \oplus (Y_3 \lll 56) \oplus (Y_3 \lll 60)\\ Z_4 &\leftarrow Y_4 \oplus (Y_4 \lll 6) \oplus (Y_4 \lll 12).\end{aligned}$$

where $\mathbf{SD}(\mathbf{a}) = Z_0\|Z_1\|Z_2\|Z_3\|Z_4\|$ and $\lll$ is the left cyclic shift operation.

2.1.4 Round Constant Layer: AddRoundConst (RC)

We add the round constants to each round of the permutation to destroy the structural symmetry in the permutation. We generate the round constants by a 5-bit LFSR defined by the primitive feedback polynomial $x^5 + x^3 + 1$ over $\mathbb{F}_2$. We start with the initial state $(1, 0, 1, 0, 1)$ to generate 14 round constants where each state of the LFSR is served as distinct constants. For example, the 5-bit LFSR state $(1, 0, 1, 0, 1)$ is converted to a byte $(0, 0, 0, 1, 0, 1, 0, 1)=$ `0x15`, and then a 64-bit round constant is constructed as `0xaaaaaaaaaaaaaa`$\|$`0x15` $=$ `0xaaaaaaaaaaaaaa15`. The round constants are given in Table 2.3.

Table 2.3: The round constants $\{\mathsf{rc}_i\}$

Round #	Constants	Round #	Constants
0	0xaaaaaaaaaaaaaa15	7	0xaaaaaaaaaaaaaa06
1	0xaaaaaaaaaaaaaa1a	8	0xaaaaaaaaaaaaaa03
2	0xaaaaaaaaaaaaaa1d	9	0xaaaaaaaaaaaaaa11
3	0xaaaaaaaaaaaaaa0e	10	0xaaaaaaaaaaaaaa18
4	0xaaaaaaaaaaaaaa17	11	0xaaaaaaaaaaaaaa1c
5	0xaaaaaaaaaaaaaa1b	12	0xaaaaaaaaaaaaaa1e
6	0xaaaaaaaaaaaaaa0d	13	0xaaaaaaaaaaaaaa1f

2.1.5 Second Permutation Layer: PLayer ($\mathbf{PL}_2$)

`FIST` permutation construction. First we construct a bit permutation P over 320 symbols, which we call `FIST` permutation[1]. Suppose $P_i, 0 \le i \le 4$ is a bit permutation over 64 symbols, i.e., $P_i = [P_i(0), \cdots, P_i(63)]$. The `FIST` permutation ($P$) over 320 symbols is constructed from P_i's as follows:

$$P = [P_0(0), \cdots, P_0(63), 64 + P_1(0), \cdots, 64 + P_1(63), 128 + P_2(0), \cdots, 128 + P_2(63),$$
$$192 + P_3(0), \cdots, 192 + P_3(63), 256 + P_4(0), \cdots, 256 + P_4(63)].$$

We now describe the construction of P_i. Let $V = V_0\|V_1\|V_2\|V_3$, $V_i \in \{0,1\}^{16}$ be the 16-bit representation of a 64-bit word V. The 16-bit rotation function on V with rotation constant u, denoted by $\texttt{ROT16}(V, u), 0 \le u \le 15$ is defined as

$$\texttt{ROT16}(V, u) = (V_0 \lll u)\|(V_1 \lll u)\|(V_2 \lll u)\|(V_3 \lll u).$$

Let $W = w_0w_1w_2w_3w_4w_5w_6w_7$ be the 8-bit representation of W where $w_i \in \{0,1\}^8$. Let π be a permutation over $\{0, \cdots, 7\}$. A byte-shuffling transformation on W with respect to π, denoted by $\texttt{ByteShuffle}(W, \pi)$, is defined as

$$\texttt{ByteShuffle}(W, \pi) = w_{\pi(0)}w_{\pi(1)}w_{\pi(2)}w_{\pi(3)}w_{\pi(4)}w_{\pi(5)}w_{\pi(6)}w_{\pi(7)}.$$

We define a bit permutation over 64 symbols using $\texttt{ROT16}(,)$ and $\texttt{ByteShuffle}(,)$ as

$$Y \leftarrow \texttt{ROT16}(Y, u)$$
$$Y \leftarrow \texttt{ByteShuffle}(Y, \pi).$$

This bit permutation is uniquely determined by the parameters u and π. Table 2.4 presents the parameters for five permutations on 64 symbols constituting the `FIST` permutation used in the permutation.

[1] As five fingers make a fist, similarly five bit permutations over 64 symbols come together to create the bit permutation P.

Table 2.4: Parameters for five permutations on 64 symbols for the `FIST` permutation

Permutation	**Parameters** (u, π_i)
P_0	$(11, [7, 0, 3, 5, 4, 6, 2, 1])$
P_1	$(4, [0, 6, 1, 7, 3, 4, 2, 5])$
P_2	$(10, [7, 2, 4, 5, 1, 0, 6, 3])$
P_3	$(7, [2, 4, 5, 3, 0, 7, 6, 1])$
P_4	$(5, [3, 6, 1, 0, 5, 7, 2, 4])$

Inverse of $\mathbf{PL}_1$. The inverse permutation of $\mathbf{PL}_1$ is denoted by $\mathbf{IPL}_1$, which is given by $\hat{IP}_i(j) = 64j + i$, $0 \leq i \leq 63$ and $0 \leq j \leq 4$. For a state $\mathbf{a} = a_0a_1a_2 \cdots a_{318}a_{319}$, the inverse of $\mathbf{PL}_1$ is given by

$$\mathbf{IPL}_1(a) = a_0a_{64}a_{128}a_{192}a_{256}a_1a_{65}a_{129}a_{193}a_{257} \cdots a_{63}a_{127}a_{191}a_{255}a_{319}.$$

Construction of $\mathbf{PL}_2$. The $\mathbf{PL}_2$ is constructed by composing the `FIST` permutation P and $\mathbf{IPL}_1$, meaning first P is applied on the input and then $\mathbf{IPL}_1$ is applied on the output of P. Symbolically, $\mathbf{PL}_2$ is written as $\mathbf{PL}_2 = \mathbf{IPL}_1 \circ P$. Note that it is not a typical permutation composition. For the parameter in Table 2.4, the full description of $\mathbf{PL}_2$ is given in Table A.2.

2.2 Sycon Modes: Authenticated Encryption

An authenticated encryption with associated data (AEAD) consists of a tuple of two algorithms, namely the authenticated encryption algorithm and the decryption and verification of tags. The authenticated encryption algorithm accepts as input a secret key K, a public nonce N, an (optional) associated data A and plaintext message M and output a ciphertext C and a tag T. Symbolically,

$$\mathcal{E}(K, N, A, M) = (C, T).$$

The decryption algorithm accepts as input a secret key K, a public nonce N, an (optional) associated data A, a ciphertext message C, and a tag T and computes a tag T'. It outputs the plaintext message M if the verification of the tag succeeds, otherwise, outputs $\perp$.

$$\mathcal{D}(K, N, A, C, T) = \begin{cases} M & T = T' \\ \perp & T \neq T'. \end{cases}$$

The entire encryption process consists of four distinct phases/algorithms, namely the initialization phase, processing associated data, encrypting message and generation of the tag. Figure 2.4 presents an overview of the authenticated encryption algorithms of Sycon. Like encryption, the decryption process also has four algorithms that are same except encryption. All five phases are described in detail in Sections 2.2.3 - 2.2.5.

2.2.1 Parameter Sets and Security Claims

The Sycon family of authenticated encryption has two instances that are constructed from the Sycon permutation using the MonkeyDuplex mode [6]. Each instance is determined by the key length, the nonce length, the tag length, the rate length, and an initial vector (IV). The first instance has a rate of 64 bits, denoted by Sycon_AEAD_128_r64, and the second one has a rate of 96 bits, denoted by Sycon_AEAD_128_r96. Each instance of Sycon supports variable length plaintexts and associated data. We shall recommend the first instance where applications require lightweight authenticated encryption and hash functionality, and the second instance is for lightweight applications where speed matters.

Table 2.5: Recommended parameters for SYCON_AEAD instances

Algorithms	Key (κ)	Nonce (n)	Tag (τ)	Rate (r)	Rounds	
					ρ_1	ρ_2
SYCON_AEAD_128_r64	128	128	128	64	14	7
SYCON_AEAD_128_r96	128	128	128	96	14	9

- ρ_1 is used in the initialization and finalization phases.
- ρ_2 is used in the AD and encryption/decryption phases.

Table 2.6: The initial vectors for the SYCON_AEAD instances

Algorithms	Initial vector (IV)	Const. identity
SYCON_AEAD_128_r64	0x0000000000000000	iv0
SYCON_AEAD_128_r96	0x5980A92AFC5D9D2C	iv1

We emphasize that two SYCON AE instances provide no security if two plaintexts are encrypted using the same key and the same nonce. In the decryption phase, the decrypted plaintext is only output if the tag verification is successful, otherwise, it outputs $\perp$. We also limit the data usage (2^a) for each key, which is the number of plaintexts and associated data used per key. We set the exponent of the data usage to 64.

Table 2.7: Security claims for SYCON_AEAD instances

Algorithms	Confidentiality	Integrity	Authenticity	Data usages (a)
SYCON_AEAD_128_r64	128	128	128	64
SYCON_AEAD_128_r96	128	128	128	64

2.2.2 Description of Mode Components

Padding. When the length of the plaintext or associated data is not a multiple of r, padding is mandatory to make the message block multiple of r. For empty associated data, no padding is applied, otherwise, the padded associated data A is $\textsc{pad}_r(A) = A\|1\|0^{r-1-|A| \bmod r}$. For the plaintext message M, the padding is applied as $\textsc{pad}_r(M) = M\|1\|0^{r-1-|M| \bmod r}$. The padding rules for the plaintext and associated data are summarized below.

$$\textsc{pad}_r(A) = \begin{cases} A\|1\|0^{r-1-|A| \bmod r} = A_0 \cdots A_{t-1} & |A| > 0 \\ \phi & |A| = 0. \end{cases}$$

$$\textsc{pad}_r(M) = M\|1\|0^{r-1-|M| \bmod r}, |A| \geq 0.$$

For $\kappa = 128$ and $r = 64$, when the key is absorbed into the state, no padding is required, i.e., whereas, for $r = 96$, the padding for the key is required, which is described below.

$$\textsc{pad}_r(K) = \begin{cases} K_0\|K_1 & \text{if } r = 64 \\ K\|1\|0^{r-1-|K| \bmod r} = K_0\|K_1 & \text{if } r = 96. \end{cases}$$

Positions for the rate and capacity. The state of the permutation is divided into two parts, called rate part and capacity part. Any input that is absorbed into the state is done through the rate part. Given the state of the permutation $S = (s_0, s_1, \cdots, s_{319})$, for

$r = 64$, the rate part of the state, denoted by S_r, is given by $S_r = (s_0, s_1, \cdots, s_{63})$. The capacity part of the state, denoted by S_c, is given by $S_c = (s_{64}, s_{65}, \cdots, s_{318}, s_{319})$. For $r = 96$, the rate part of the state is given by $S_r = (s_0, s_1, \cdots, s_{94}, s_{95})$, and the capacity part is given by $S_c = (s_{96}, s_{97}, \cdots, s_{318}, s_{319})$.
Domain separation. For each phase of the encryption/decryption algorithm, a distinct (3-bit) domain separation constant is used. A domain separation constant is XORed with the last three bits of the capacity, i.e., XORed with state bits $(s_{317}, s_{318}, s_{319})$.

2.2.3 Initialization and Processing Associated Data

Initialization. The initialization phase consists of a loading phase that loads the key and nonce to the state and absorbing the key into the state. The key, nonce and initial vector (iv) loading mechanism into the state S, denoted as $\texttt{LOAD}(K, N, \mathsf{iv})$, is given by

$$S \leftarrow \texttt{LOAD}(K, N, iv) = (k_0, k_1, \cdots, k_{127}, n_0, n_1, \cdots, n_{127}, iv_0, \cdots, iv_{63})$$

where $K = (k_0, \cdots, k_{127})$ is the key, $N = (n_0, \cdots, n_{127})$ is the nonce, and $iv = (iv_0, \cdots, iv_{63})$ is the initial vector. Algorithm 1 presents the steps of the initialization.

Processing Associated Data. This algorithm is applied after the initialization phase. It accepts the associated data (AD) and the current state as input and returns the state of the permutation. Note that the padding rule is applied on the associated data if it is nonempty. The steps of the algorithm is described in Algorithm 2.

Table 2.8: The SYCON initialization and associated data processing algorithms

Algorithm 1 Proc. initialization

Input: Key K, nonce N, and IV iv
Output: State S
$S \leftarrow \texttt{LOAD}(N, K, iv)$
$\text{PAD}_r(K) = K_0 \| K_1$
$S \leftarrow \Pi^{14}((S_r \oplus K_0), S_c)$
$S \leftarrow \Pi^{14}((S_r \oplus K_1), S_c \oplus (0^{c-2} \| 01))$
return S

Algorithm 2 Proc. associated data

Input: State S, and AD A
Output: State S
$\text{PAD}_r(A) = A_0 \| \cdots \| A_{\ell_A - 1}$
for i **from** 0 **to** $\ell_A - 2$ **do**
    $S \leftarrow \Pi^{\rho}((S_r \oplus A_i), S_c \oplus (0^{c-2} \| 01))$
end for
$S \leftarrow \Pi^{\rho}((S_r \oplus A_{\ell_A - 1}), S_c \oplus (0^{c-2} \| 10))$
return S

2.2.4 Encryption and Decryption

Encryption. After processing the associated data, the encryption algorithm is applied on the plaintext M of length $m = |M|$. First, the padding rule (10^*) is applied on the plaintext M, and the padding on M returns a padded message which is a multiple of r, i.e., $M_0 \| \cdots \| M_{\ell_M - 1}$ where ℓ_M is the number blocks for the padded message. The encryption algorithm produces a ciphertext of length m corresponds to the input plaintext. The detailed steps of encryption are given in Algorithm 3.
Decryption. Like encryption, the decryption algorithm is applied to the ciphertext C after processing the associated data. The detailed steps of the decryption algorithm are provided in Algorithm 4.

2.2.5 Tag Generation

After the encryption or decryption algorithm, the tag generation algorithm is executed. The tag generation algorithm accepts the state after encryption and the key again and outputs a tag of length $\tau = 128$. A tag is constructed by concatenating 128 bits from

Table 2.9: The SYCON encryption $\mathcal{E}()$ and decryption $\mathcal{D}()$ algorithms

Algorithm 3 Encryption algorithm $\mathcal{E}()$

Input: Plaintext M and state S
Output: Ciphertext C and state S
$\text{PAD}_r(M) = M_0\|\cdots\|M_{\ell_M-2}$
for i **from** 0 **to** $\ell_M - 2$ **do**
  $C_i \leftarrow M_i \oplus S_r$
  $S \leftarrow \Pi^\rho(C_i, S_c \oplus (0^{c-2}\|10))$
end for
$C_{\ell_M-1} \leftarrow M_{\ell_M-1} \oplus S_r$
$S \leftarrow \Pi^\rho(C_{\ell_M-1}, S_c \oplus (0^{c-3}\|100))$
$C_{\ell_M-1} \leftarrow \lfloor C_{\ell_M-1} \rfloor_{m\%r}$
return $C = C_0\|\cdots\|C_{\ell_M-1}$ and S

Algorithm 4 Decryption algorithm $\mathcal{D}()$

Input: Ciphertext C and state S
Output: Plaintext M and state S
$\text{PAD}_r(C) = C_0\|\cdots\|C_{\ell_M-1}$
for i **from** 0 **to** $\ell_C - 2$ **do**
  $M_i \leftarrow C_i \oplus S_r$
  $S \leftarrow \Pi^\rho(C_i, (S_c \oplus (0^{c-2}\|10))$
end for
$M_{\ell_M-1} \leftarrow C_{\ell_M-1} \oplus S_r$
$S \leftarrow \Pi^\rho(C_{\ell_M-1}, S_c \oplus (0^{c-3}\|100))$
$M_{\ell_M-1} \leftarrow \lfloor M_{\ell_M-1} \rfloor_{m\%r}$
return $M = M_0\|\cdots\|M_{\ell_M-1}$ and S

the indices 128 to 191 in the state. Given the state $S = (s_0, s_1, \cdots, s_{318}, s_{319})$, the tag extraction function, denoted by $\texttt{ExtTag}(S)$, extracts the tag as follows:

$$\texttt{ExtTag}(S) = s_{128}s_{129}\cdots s_{191}\|s_{192}s_{193}s_{13}\cdots s_{254}s_{255}.$$

Table 2.10: The tag extraction algorithm

Algorithm 5 Finalization algorithm

Input: State S and key K
Output: Tag T
$\text{PAD}_r(K) = K_0\|K_1$
$S \leftarrow \Pi^{14}((S_r \oplus K_0), S_c)$
$S \leftarrow \Pi^{14}((S_r \oplus K_1), S_c)$
return $T \leftarrow \texttt{ExtTag}(S)$

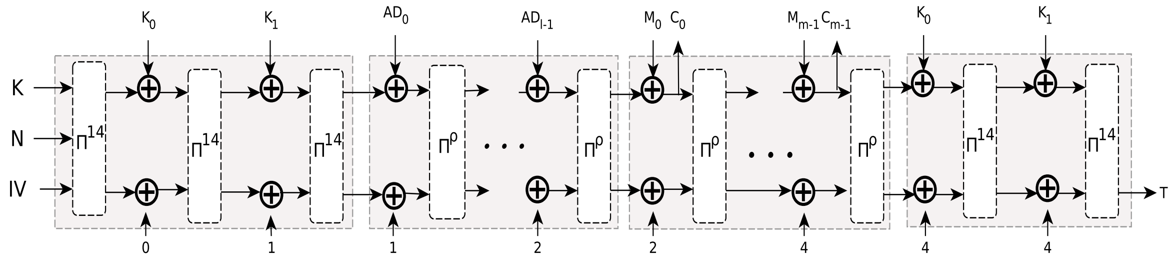

Figure 2.4: Modes for authentication encryption with associated data, $\rho = 7$ and 9 for SYCON_AEAD_128_r64 and SYCON_AEAD_128_r96, respectively

2.3 SYCON Mode: Hash Algorithm

A hash function accepts a message of an arbitrary length as input and outputs a message digest of a fixed length. Mathematically, $H : \{0,1\}^* \to \{0,1\}^{\ell_h}$ where ℓ_h is the length of the digest. We use our SYCON permutation in the unified sponge mode [9] to achieve the hash function. The parameters for the hash function and the security claims are provided in Tables 2.11 and 2.12, respectively. The hash algorithm consists of three steps, namely loading the initial vector (iv2) into the state, absorbing the message and squeezing the hash value. The 64-bit iv2 is loaded into the state bits $(s_{128}, s_{129}, \cdots, s_{191})$

positions, and the remaining state bits are set to zero, which is denoted as `LOADIV`(iv2), i.e., `LOADIV`(iv2) $= 0^{128} \| \text{iv2} \| 0^{128}$. The description of the steps for the hashing are given in Algorithm 6. Figure 2.5 depicts an overview of the hash algorithm.

Table 2.11: Recommended parameters for SYCON_HASH

Algorithm	**IV** (iv2)	**Digest** (ℓ_h)	**Rate** r	**Capacity** c	**Rounds** (Π^ρ) ρ
SYCON_HASH_256	`0x1C0A80D42C6E63C5`	256	64	256	14

Table 2.12: Security claims for the SYCON hash algorithm (in bits)

Algorithm	**Preimage resistance**	**2nd preimage resistance**	**Collision resistance**
SYCON_HASH_256	192	128	128

Algorithm 6 The SYCON hash algorithm

Input: State S, iv2, and message M
Output: Message digest D
$\text{PAD}_r(M) = M_0 \| \cdots \| M_{\ell_M - 1}$
$S \leftarrow$ `LOADIV`(iv2)
for i **from** 0 **to** $\ell_M - 1$ **do**
    $S \leftarrow \Pi^{14}((S_r \oplus M_i), S_c)$
end for
$D_0 \leftarrow S_r$
$S = (S_r, S_c) \leftarrow \Pi^{14}(S)$
$D_1 \leftarrow S_r$
$S = (S_r, S_c) \leftarrow \Pi^{14}(S)$
$D_2 \leftarrow S_r$
$S = (S_r, S_c) \leftarrow \Pi^{14}(S)$
$D_3 \leftarrow S_r$
return $D = D_0 \| D_1 \| D_2 \| D_3$.

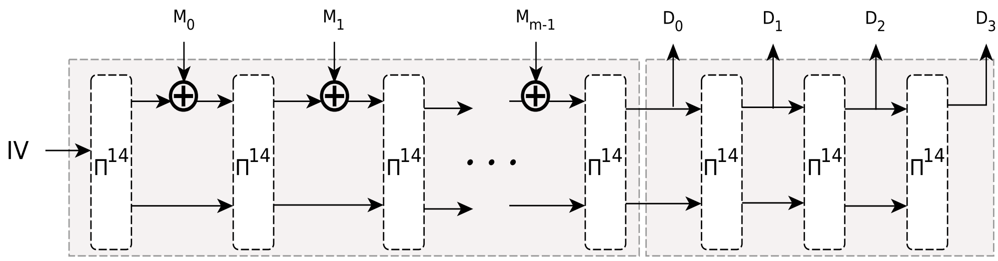

Figure 2.5: A block diagram of the SYCON_HASH_256 algorithm

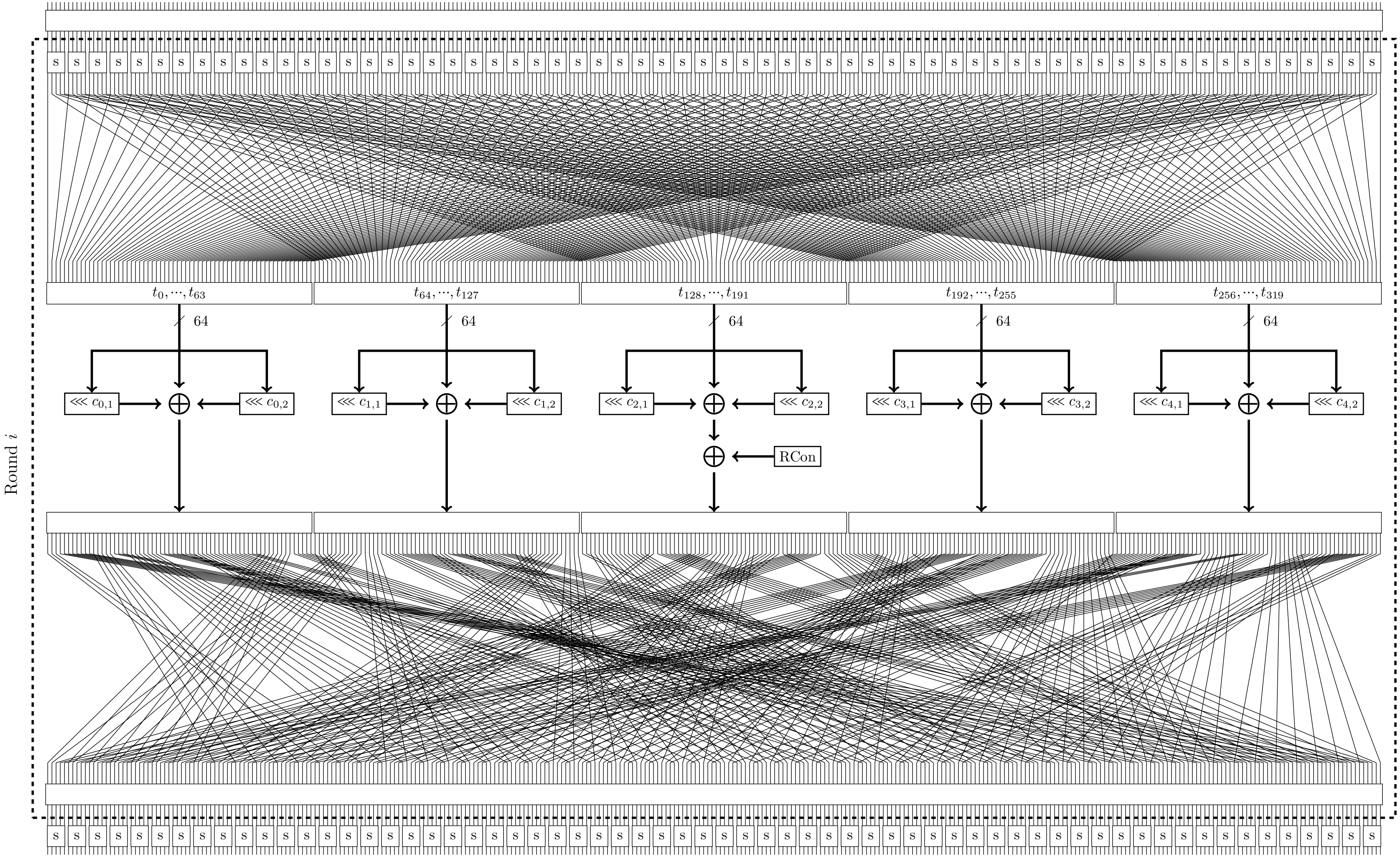

Figure 2.3: Round function of Π^ρ, here $c_{0,1} = 11, c_{0,2} = 22, c_{1,1} = 13, c_{1,2} = 26, c_{2,1} = 31, c_{2,2} = 62, c_{3,1} = 56, c_{3,2} = 60, c_{4,1} = 6, c_{4,2} = 12$.

Chapter 3

Security Analysis

In this chapter, we discuss the security features of SYCON. We use the provably secure sponge mode to guarantee the security of SYCON authenticated encryption and hash algorithms. To ensure the security of the SYCON algorithms, we investigate the security of the SYCON permutation against cryptanalytic attacks such as differential and linear cryptanalysis, impossible differential cryptanalysis, and zero-sum distinguisher.

3.1 Differential and Linear Cryptanalysis

We investigate the security of the SYCON permutation against differential and linear cryptanalysis. Differential [10] and linear cryptanalysis [18] are the two most powerful techniques to analyze symmetric-key primitives. A practical approach to measure the resistance against differential and linear cryptanalysis is to count the minimum number of differential and linear active S-boxes. These optimization problems can be modelled as a mixed integer linear programming problem (MILP), and some MILP solver like Gurobi [1] can be used to solve them as was shown in [19] for word oriented cipher like AES [15]. However, the MILP modelling slightly differs when it comes to analyse bit oriented ciphers like PRESENT [11]. We follow the MILP modelling for bit oriented ciphers as shown in [21], and count the number of active S-boxes for both differential and linear cryptanalysis for few rounds.

3.1.1 Differential Cryptanalysis

The maximum differential probability of SYCON's S-box is 2^{-2}. The differential branch number is 3, so diffusion itself starts from the S-box layer, which is further enhanced by passing through the SD layer and the `FIST` permutation layer. Table 3.1 provides a summary of the number of active S-boxes for the first few rounds. Thus, in 4 rounds, the

Table 3.1: Number of active S-boxes

Number of rounds	1	2	3	4
Active S-boxes	1	4	11	51^*

differential attack complexity[1] reaches at the order of 2^{102^*}.

[1] The ($*$) mark indicates the bound that Gurobi MILP solver gave us at the best.

3.1.2 Linear Cryptanalysis

The maximum linear probability of SYCON 's S-box is 2^{-2}. The linear branch number is 3 which helps to increase the number of active S-boxes for linear trails. Below we present the number of linearly active S-boxes for a few rounds.

Table 3.2: Number of active Sboxes

Number of rounds	1	2	3	4
Active Sboxes	1	4	9	39^{**}

The complexity for the linear attack[2] in 4 rounds is $2^{78^{**}}$.

3.2 Impossible Differential Cryptanalysis

To find the existence of impossible differentials in the SYCON permutation, we need to check whether an input-output difference pair, denoted by (Δ_i, Δ_o), that is impossible or not. We apply the same MILP based automated technique as done in [20, 14]. We only search for one-weight input and output differences. As a result, for 4 rounds, we get the following impossible input and output difference pair. We are not aware of any attack that can be launched by exploiting this property of the permutation.

Table 3.3: Impossible differnce pair for 4 rounds

Input Difference (Δ_i)
0x0000000000000000000000000001000000000000 0000000000000000000000000000000000000000
Output Difference (Δ_o)
0x0000000000000000000000000000000000000000 0000000000000000000000000000002000000000

3.3 Zero-sum Distinguisher

We check the validity of the zero-sum distinguisher [5] on the SYCON permutation Π, which was first shown to distinguish the Keccak permutation [9]. For any function $\phi : \mathbb{F}_2^n \to \mathbb{F}_2^n$, the set $\{X_1, \ldots, X_\ell\} \subset \mathbb{F}_2$ is said to be zero-sum if $\sum_{i=1}^{\ell} X_i = 0$ and $\sum_{i=1}^{\ell} F(X_i) = 0$. Clearly, any function having this property can be distinguished from a random permutation. Suppose F is an iterated permutation that $F = \phi^t$. Let $\deg(F)$ denote the degree of F. Consider an intermediate round r, and compute the degree of $(\phi^{-1})^r$ and ϕ^{t-r}. Suppose there is an $s < n$ such that

$$\deg(\phi^{-1})^r < s \text{ and } \deg(\phi^{t-r}) < s$$

Then fix $(n-s)$ bits of the output of r (ϕ^r), and vary all possible s bits. Then the sum of images of these 2^s elements under ϕ^{-1} will be zero as $\deg(\phi^{-1})^r < s$. Further, the sum of images of these 2^s elements under ϕ^{n-r} will also be zero as $\deg(\phi^{t-r}) < s$. Therefore, we have a zero-sum distinguisher for F. Lower the degree of ϕ and ϕ^{-1} and lower the number of rounds of F, better the chance of having zero-sums.

[2]The $(**)$ mark indicates the bound that Gurobi MILP solver gave us at the best.

The degrees of SYCON S-box and its inverse are 2 and 3, respectively. Thus, the degrees of Π and Π^{-1} are 2 and 3 also. Therefore, the degree of $\Pi^t \leq \min(2^t, n-1)$, and the degree of $(\Pi^{-1})^t \leq \min(3^t, n-1)$. However, as noted in [13], the Walsh spectrum of S-box plays an important role in bounding the overall degree of the permutation. For instance, if there are n_s S-boxes present in the round function and each Walsh spectrum value is divided by 2^{w_s}, then we have

$$\deg(\phi^r) \leq n - n_s \cdot w_s + \deg(\phi^{r-1}).$$

Applying this to the SYCON permutation, we get that

$$\deg(\Pi^{-1})^7 \leq 320 - 64 \cdot 3 + \deg(\Pi^{-1})^6 = 192,$$

which is much less than $\min(319, 3^7)$. In fact, this trick was applied in [12] to show the existence of zero-sums in the Keccak permutation much efficiently. If we apply the same technique, we will not be able to show the existence of a zero-sum in the SYCON permutation. However, using the observation of [12] which adds one intermediate round for free, existence of zero-sum can be shown as follows. Fix the $320 - 260 = 60$ input bits to the 6th round. If we vary 260, then all $260/5 = 52$ S-boxes receive all possible values 5-bits and so are their outputs. Then the images of these all possible 260-bits sum to zero as the degree of $(\Pi^{-1})^5$ is $3^5 = 243$. Secondly, the output of the S-box layer of the 6th round will go through linear layer (no degree enhancement) followed by the rounds 7 to 14, that is 8 rounds in total. The degree of $\Pi^8 = 2^8 = 256$. That means images of these outputs of 6th round S-box layer will also sum to zero. Hence the zero-sum distinguisher is shown to exist for the full SYCON permutation ($\rho = 14$). However, the complexity is 2^{260}, which is way too high that the security claim that SYCON has, though it proves that SYCON is not an ideal permutation.

Chapter 4

Design Rationale for SYCON

In this chapter we discuss the rationale behind SYCON . We focus on choosing the components in such a way that they lead to low implementation cost in both hardware and software.

4.1 Choosing the Mode

For the choice of modes, we want to leverage existing provably secure modes to instantiate the SYCON permutation to obtain an AE and a hash function. In the literature, there are several lightweight variants of the sponge construction with goals of making the modes efficient and secure, for example, making key absorption efficient, domain separation for preventing attacks, and reducing the round of the permutation for efficiency [6]. The mode of SYCON_AEAD is a MonkeyDuplex mode where the key absorption is inspired from the sLiSCP mode [3], which is lightweight, and the domain separation is inspired from NORX [4]. For the hash algorithm, we use the sponge mode of operation [9]. These modes are widely used and have been proven secure.

One advantage of MonkeyDuplex mode in the lightweight AE applications, is that this mode does not require any key-schedule and it is efficient due to reduced internal rounds of the permutation. Otherwise this would have been an additional burden on the implementation. We take the advantage of flexible number of rounds of the permutation that processes associated data or the message part, as these parts of the mode do not need ideal permutation as compared to the initiation part. An interesting feature of this mode is that the same permutation works in the decryption, and overall overhead for decryption over encryption is minimal.

Security guarantee. The security of the AEAD modes of SYCON directly follows that of the MonkeyDuplex mode. An improved security bound of sponge-based constructions for authenticated encryption is proved in [17] in terms of the key size, the permutation state size, and the capacity size. More precisely, the security bound is $\min\{2^{\kappa}, 2^{c}, 2^{\frac{320}{2}}\}$ where κ and c are chosen in such a way that we can achieve 128-bit security for both instances (see Tables 2.5 and 2.7). For the keyed sponge, the relation between the usage exponent and the capacity is given by $c \geq a + \kappa + 1$ [8]. Relying on these two results, the security claims of SYCON algorithms are justified.

4.2 Choosing the S-boxes

We chose the 5×5 S-box with good cryptographic properties as well as efficient hardware and (bit-sliced) software implementations. The software efficiency of the S-box is defined as the minimum number of instructions need to implement the S-box. A lot of effort has been given to choose the S-box. We set the cut-off to choose S-boxes that need below 30 GE (under 65 nm technology), and with differential uniformity 8, nonlinearity 8, differential branch number 3 and linear branch number 3. While constructing such S-boxes, we set another criterion that the S-box should have less number of terms in the ANF. By setting differential and linear branch number to 3, we ensure an increasing number of active S-boxes per round for differential and linear trails. As a result, we obtain the S-box as mentioned in Table 2.1.

4.3 Choosing the Linear Diffusion Layer

Once the S-box is decided, we chose the SubBlockDiffusion layer which has an efficient hardware implementation. We stick to the linear branch number 4 for the SubBlockDiffusion layer. The implementation cost for the linear transformation $x \oplus (x \lll r_1) \oplus (x \lll r_2)$ depends on the rotation constant pair (r_1, r_2). First, we have generated all such diffusion layers with linear branch number 4 and classified those based on the number of XOR gates needed to implement them. As per our design we need five linear diffusion functions that act on 64-bits. We choose differential branch number 4 as a trade-off between the implementation cost and the number of active S-boxes. We choose linear diffusion function of the form $x \mapsto x \oplus (x <<< r_1) \oplus (x <<< r_2)$. Obviously this type of functions are highly software friendly. On the other hand they ensure that the Hamming weight of each row/column is 3. Apparently these type of functions need 128 XORs, however, we search for all possible rotation constant pairs (r_1, r_2), where we can take the advantage of subexpression elimination, so that the final XOR requirement come below 128. We chose five such pairs that are listed in Section 2.1.3.

4.4 Choosing the Round Constant

The main reason for introducing the round constant is to destroy the symmetry in the round output. For an efficient hardware implementation, every time we generate distinct 5 tuples that is produced by a 5-stage LFSR with primitive polynomial $x^5 + x^3 + 1$ that costs 2 XORs and 5 flip-flops. We decide to load the round constant whose 128 MSBs and 128 LSBs are all zeros, so that it serves our purpose at the same time puts less burden on the implementation.

4.5 Choosing the Bit Permutations

The primary reason to choose $\mathbf{PL}_2$ is to lift the branch number after the linear diffusion layer. We look for the bit permutation (`FIST` permutation) which result in a higher number of branch numbers at the end of certain number of rounds. Further, we search for such bit permutation in a special class. Note that in case of hardware, the bit permutation comes for free. However, for software (given the state size is more than 64-bits), implementing it needs few more instructions depending upon the implementation platform. The beauty of the `FIST` construction of bit permutation is that it can be handled in separate five 64-bit blocks, which further boils down to process 16 and 8-bit words. Therefore, `FIST` construction is suitable for as low as 8 bit microcontrollers.

4.6 Choice of Initial Vectors

The initial vectors uniquely identity different AEAD and hash instances of Sycon. The initial vectors iv1 used in Sycon_AEAD_128_r96 and iv2 in Sycon_HASH_256 are obtained by taking the output of the Riemann Zeta function [16] evaluated at 2 and 3, respectively, and then considering the 19 decimal places in the hex representation.

Chapter 5

Efficiency Evaluation of Sycon

In this chapter, we report the hardware implementation results in FPGAs and ASICs. We also assess the performance of the Sycon permutation, authenticated encryption and the hash algorithms on high-speed CPUs as well as microcontrollers.

5.1 Hardware Implementation Results

5.1.1 FPGA and ASIC Synthesis Results

The Hardware implementation was carried out using Verilog HDL and for FPGA the design was synthesized on the Xilinx Vivado while the ASIC synthesis was done on Synopsis Design Compiler using UMC 65nm technology. Two variants of Sycon have been implemented with the main difference being the implementation of the Sbox layer.

For the first variant, an iterative strategy has been followed to implement the underlying Sycon permutation. Entire state is processed using 64 parallel Sbox-es. The round function has been realized combinatorially. A 320-bit register has been used to store the state after every iteration of the round function. For the authenticated encryption mode, after one application of the permutation, the message is absorbed into the state and appropriate domain separators are applied before the message is fed back. The ciphertext blocks and the tag are output at appropriate times based on the algorithm. The hashing mode is similar except the fact that the message is now xored only in the absorption phase while the hash output happens in the squeezing phase as per the sponge construction. The datapaths of both the modes are furnished in Fig. 5.1 and Fig. 5.2.

For the second variant, a *single* Sbox is used to process the entire state serially in 64 clock cycles. This is achieved using a 320-bit circular shift register where the first 5-bits are processed through the Sbox and fed into the last 5-bits. Detailed FPGA results for three different target devices are given in Table 5.1 while the ASIC area results are furnished in Table 5.2. The difference in area between the two variants for both the modes can be appreciated in terms of the FPGA LUT count and ASIC GE.

5.2 Efficiency Evaluation in Software

5.2.1 Bit-Sliced efficiency on 64-bit CPUs

We have implemented the Sycon permutation, AEAD and hash algorithms in the bit-sliced fashion using the SIMD Intel Intrinsics including SSE2 and AVX2. The SSE2 supports operations on 128-bit XMM registers and AVX2 supports operations on 256-bits

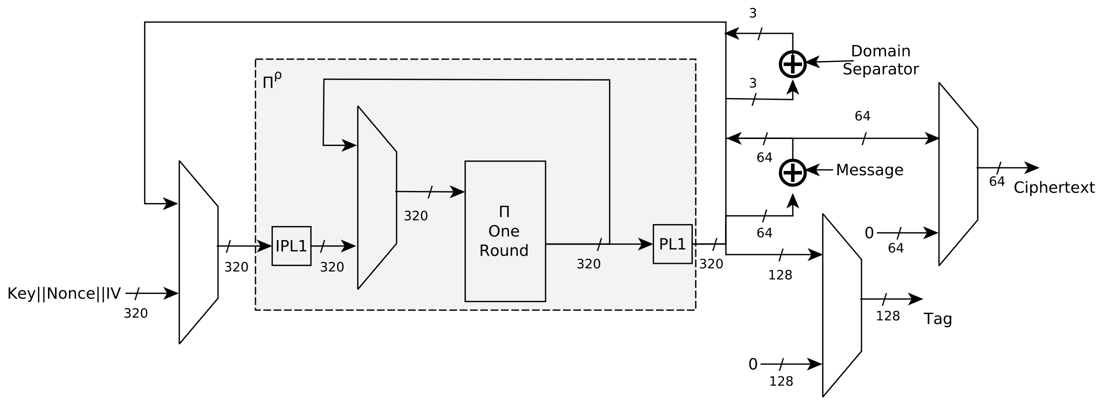

Figure 5.1: SYCON datapath for authenticated encryption mode

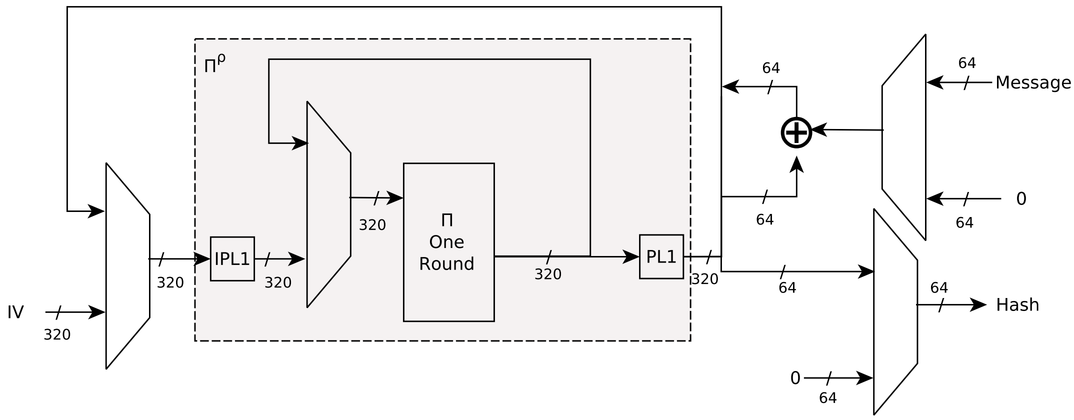

Figure 5.2: SYCON datapath for hashing mode

YMM registers. We use two different CPUs, namely Skylake and Haswell to obtain efficiency results. The codes were compiled using `gcc 5.4.0` (Skylake) and `llvm 10.0.0` (Haswell) with `-g -Wall -O2 -fomit-frame-pointer -funroll-all-loops` flags. Algorithm 7 presents an equivalent (bit-sliced) representation of the SYCON permutation. Table 5.3 presents the speed for the SYCON permutation, authenticated encryption and hash algorithms. The speed is measured in terms of the number of clock cycles per byte. The best speed achieved by the SYCON permutation is 3.23 cpb in the AVX2 implementation. When computing the speed for SYCON_AEAD, we chose a plaintext message of length 2048 bits and an associated data of length 128 bits where the speed computation includes executions of initialization, AD processing, encryption and tag generation algorithms.

5.2.2 Efficiency on microcontroller

To assess the software performance of SYCON on microcontrollers, we have implemented the SYCON authenticated encryption and hash algorithms on the 8-bit Atmel Atmega32 and a 32-bit MIPS32 from MIPS Technologies. The 8-bit Atmel Atmega32 microcontroller has 2 Kbytes of flash, 32 KBytes of RAM and 32 8-bit general purpose registers. MIPS32 has 32 32-bit general purpose registers. We implement the SYCON instance in assembly, and the AVR Simulator IDE was used to write the code. In our implementations, we implement the S-box in the bitsliced fashion, instead of look up table, to achieve highest efficiency while reducing memory. We use a plaintext of 572 bits in our experiment to obtain cycles

Table 5.1: FPGA and ASIC implementation results

		Parallel Sbox			Serial Sbox		
SYCON Variant	FPGA Platform	Slice Registers	Slice LUTs	Frequency (MHz)	Slice Registers	Slice LUTs	Frequency (MHz)
AEAD_r64	Spartan-7 (xc7s50ftgb196-1)	328	693	246.9	335	651	188.7
	Kintex-7 (xc7k160tfbv676-2)	328	693	416.7	335	651	333.3
	Artix-7 (xc7a200tfbv484-3)	328	693	358.4	335	651	277.8
AEAD_r96	Spartan-7 (xc7s50ftgb196-1)	328	727	246.9	335	682	188.7
	Kintex-7 (xc7k160tfbv676-2)	328	727	425.5	335	682	333.3
	Artix-7 (xc7a200tfbv484-3)	328	727	358.4	335	682	277.8
HASH_256	Spartan-7 (xc7s50ftgb196-1)	326	646	250.0	333	471	232.6
	Kintex-7 (xc7k160tfbv676-2)	326	646	476.2	333	471	384.6
	Artix-7 (xc7a200tfbv484-3)	326	646	363.6	333	471	327.9

Table 5.2: ASIC area results with UMC 65nm technology

	Chip Area	
Sycon Variant	μm^2	kGE
AE_r64 (Parallel Sbox)	8148.79	6.37
AE_r64 (Serial Sbox)	6618.56	5.17
Hash (Parallel Sbox)	8007.68	6.26

for AEAD and hash instances. For instance, the SYCON permutation evaluation requires 17,791 cycles, and the throughput of the permutation is 444.78 cycles/byte. Table 5.4 presents the cycle counts, code sizes in bytes, and cycles per byte.

Table 5.3: Performance of the Sycon permutation, AE and hash algorithms on 64-bit CPUs. The performance is measured in terms of clock cycles per byte (cpb).

Functionality	`Intel Core i5 CPU@2.6 GHz` (`Haswell`)		`Intel i7-6700 CPU@3.40GHz` (`Skylake`)	
	AVX2	SSE2	AVX2	SSE2
Π^{14} permutation	3.88	7.65	3.23	5.54
Sycon_AEAD_128_r64	12.75	22.44	14.20	20.55
Sycon_HASH_256	17.61	34.75	23.57	43.27

Table 5.4: Performance of different Sycon instances on Atmega32 (8-bit) and MIPS32 (32-bit) microcontrollers

	Sycon		Sycon_AEAD_128_r64		Sycon_AEAD_128_r96		Sycon_HASH_256	
Platform	8-bit	32-bit	8-bit	32-bit	8-bit	32-bit	8-bit	32-bit
Cycles	33,037	17,791	347,525	186,047	333,245	178,289	430,275	233,585
Code size[Bytes]	1,092	1,904	1,379	2,116	1,384	2,128	1,213	2,024
Cycles/byte	825.92	444.78	4,826.74	2,583.99	4,682.40	2,476.24	5,976.04	3,244.24

Acknowledgments

The designers would like to thank Amit Acharyya and Swati Bharadwaj for providing their ASIC synthesis platform and synthesizing the design on our behalf. The designers also thank Aikata for providing microcontroller results.

Bibliography

[1] Gurobi: Gurobi optimizer reference manual. `http://www.gurobi.com`.

[2] NIST lightweight cryptography project. 2018. `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf`.

[3] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. sliscp: Simeck-based permutations forlightweight sponge cryptographicprimitives. In Carlisle Adams and Jan Camenisch, editors, *Selected Areas in Cryptography – SAC 2017*, pages 129–150, Cham, 2018. Springer International Publishing.

[4] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. Norx: parallel and scalable aead. In *European Symposium on Research in Computer Security*, pages 19–36. Springer, 2014.

[5] Jean-Philippe Aumasson and Willi Meier. Zero-sum distinguishers for reduced keccak-f and for the core functions of luffa and hamsi. Presented at the rump session of Cryptographic Hardware and Embedded Systems–CHES 2009, 2009.

[6] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Permutation-based encryption, authentication and authenticated encryption. Presented at DIAC 2012, July 2012. Stockholm, Sweden.

[7] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *International Workshop on Selected Areas in Cryptography*, pages 320–337. Springer, 2011.

[8] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the security of the keyed sponge construction. In *Symmetric Key Encryption Workshop*, volume 2011, 2011.

[9] Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. Keccak specifications, 2009.

[10] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *J. Cryptology*, 4(1):3–72, 1991.

[11] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *LNCS*, pages 450–466. Springer, 2007.

[12] Christina Boura and Anne Canteaut. Zero-sum distinguishers for iterated permutations and application to keccak-f and hamsi-256. In *Proceedings of the 17th International Conference on Selected Areas in Cryptography*, SAC'10, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Anne Canteaut and Marion Videau. Degree of composition of highly nonlinear functions and applications to higher order differential cryptanalysis. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 2002.

[14] Tingting Cui, Keting Jia, Kai Fu, Shiyao Chen, and Meiqin Wang. New automatic search tool for impossible differentials and zero-correlation linear approximations. *IACR Cryptology ePrint Archive*, 2016:689, 2016.

[15] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[16] H.M. Edwards. Riemann's zeta function. Dover Publications, 2001.

[17] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond 2c/2 security in sponge-based authenticated encryption modes. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 85–104, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[18] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1993.

[19] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuankun Wu, Moti Yung, and Dongdai Lin, editors, *Information Security and Cryptology - 7th International Conference, Inscrypt 2011, Beijing, China, November 30 - December 3, 2011. Revised Selected Papers*, volume 7537 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2011.

[20] Yu Sasaki and Yosuke Todo. New impossible differential search tool from design and cryptanalysis aspects - revealing structural properties of several ciphers. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 185–215, 2017.

[21] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: Application to simon, present, lblock, DES(L) and other bit-oriented block ciphers. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings,*

Part I, volume 8873 of *Lecture Notes in Computer Science*, pages 158–178. Springer, 2014.

Appendix A

Test Vectors and Sycon Permutation Details

In this section, we present the test vectors for all Sycon instances, and some details about the bit permutations and S-box.

A.1 Test Vectors for Sycon Permutations, AE and Hash

Here we present the test vectors for two authenticated encryption algorithms and the hash algorithms.
Sycon **Permutation.** The input and output after 14 rounds of the permutation are as follows.
Input:
0000000000000000000000000000000000000000AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Output:
31DED502D85527B07357D8E2BFA2AA39DED003A4D911131FBC9A5BA15618C464F23AD59EC3F5F72B

The step-by-step input-outputs of the permutation for 14 rounds are given below:

```
0000000000000000000000000000000000000000AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
5715FDAABFF55F00A8AA0AFAAFFDFF5FFFFFFFFFD03FFFFFFFFFFFFFFFFFFFFF5555555555555555
299478D4E7704576AC8F0952ABDDB639336B85EF2634A01A5A8974D984001EE4531EAEAEDC855246
4AB269C2B739F02BD52273F4B6138AFF5E8CB33018FA4FCD9F65866DF08C297F61DFEC1016AFC40B
24535F18E691605BB8D53E1D651C1DA4DBDB83ACF527A8E8DEBDC83DB1BE8DF371B0D84C06DEC747
47740C7E603DF3C110111211F57A0128AF1B30AD328734AD3EBAFEFD0418C1138E85EF0694539BD5
C4D5E5FD63F897A2481F485070E19DEF3550612135D228432A7404D07263DF01318D79367BE9DDAD
3818F0EB0C0C3C2A62BADF200D727F0146B58F352ABBFFBD57ABC8A1B8ED42F55B6C5C3E1AFE06A2
7E8F444890FDDE0DC7881CB8C3605682A6439188F2D54B07226991FE7E606AE70BD50EB3B255F54C
FF5D5CCB135C8E5B4BE107B7B2980366074C1AAF48D8B28FC7DCCEAABA6CD24333C50987A984898B
8417A7027EF861358DFAC8AE0657E23432EE9F374CF795E9F7BC8C29A5DBD6D5770038408539EB7E
0DB1A793BA7C670FBCD388A4668E3049E1D4E999313222EF422D6E5004A9A15D61A0400474D072F5
4EBE001A4A2799D269ED1BF88EE5650E02A10F92CE0441A63A97143AEE31C03DD34E9CE23D79C294
BF31DB1311CB4BDF52B66CDCA995591EE013B6D076253D77ACD8FA96C18D0F0E534C52AC396D9AD5
31DED502D85527B07357D8E2BFA2AA39DED003A4D911131FBC9A5BA15618C464F23AD59EC3F5F72B
```

Test Vectors for SYCON_AEAD_128_r64:

Key	000102030405060708090A0B0C0D0E0F
Nonce	000102030405060708090A0B0C0D0E0F
Associated data	05AE023DC3105DA62894A16A0E260956
Plaintext	“To authenticate, or not to authenticate”
Plaintext (byte)	546F2061757468656E7469636174652C206F72206E6F7420746F2061757468656E746963617465
Ciphertext	0535E98A36A013905C884ED69C752D05F81C3D57EA7DA62C5857B66824E8361A8C0B2FC9691B74
Tag	D744AB39F5233F2FD6357A9BE330D9A2

Test Vectors for SYCON_AEAD_128_r96:

Key	000102030405060708090A0B0C0D0E0F
Nonce	000102030405060708090A0B0C0D0E0F
Associated data	05AE023DC3105DA62894A16A0E260956
Plaintext	“To authenticate, or not to authenticate”
Plaintext (byte)	546F2061757468656E7469636174652C206F72206E6F7420746F2061757468656E746963617465
Ciphertext	164F4F0463781EF41C3A512264B74C3B06A53BD345B8EB8E3B8D8F0930AC920591B16C4A3B5DF9
Tag	06F990758FE75620A11210C7095EBECD

Test Vectors for SYCON_HASH_256:

Plaintext	“To authenticate, or not to authenticate”
Plaintext (byte)	546F2061757468656E7469636174652C206F72206E6F7420746F2061757468656E746963617465
Digest	95088252C915EF0B5013BEA358ACEB366096D2E179603ED49D1BF60A9BF52956

A.2 Details of PLayer and S-box of Π

In this section, we provide two bit permutation layers $\mathbf{PL}_1$ and $\mathbf{PL}_2$, the difference distribution table and the linear approximation tables of the S-box.

Table A.1: Bit permutation $\mathbf{PL}_1$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$PL_1(i)$	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$PL_1(i)$	80	85	90	95	100	105	110	115	120	125	130	135	140	145	150	155
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$PL_1(i)$	160	165	170	175	180	185	190	195	200	205	210	215	220	225	230	235
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$PL_1(i)$	240	245	250	255	260	265	270	275	280	285	290	295	300	305	310	315
i	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
$PL_1(i)$	1	6	11	16	21	26	31	36	41	46	51	56	61	66	71	76
i	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
$PL_1(i)$	81	86	91	96	101	106	111	116	121	126	131	136	141	146	151	156
i	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
$PL_1(i)$	161	166	171	176	181	186	191	196	201	206	211	216	221	226	231	236
i	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
$PL_1(i)$	241	246	251	256	261	266	271	276	281	286	291	296	301	306	311	316
i	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
$PL_1(i)$	2	7	12	17	22	27	32	37	42	47	52	57	62	67	72	77
i	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
$PL_1(i)$	82	87	92	97	102	107	112	117	122	127	132	137	142	147	152	157
i	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
$PL_1(i)$	162	167	172	177	182	187	192	197	202	207	212	217	222	227	232	237
i	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
$PL_1(i)$	242	247	252	257	262	267	272	277	282	287	292	297	302	307	312	317
i	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
$PL_1(i)$	3	8	13	18	23	28	33	38	43	48	53	58	63	68	73	78
i	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
$PL_1(i)$	83	88	93	98	103	108	113	118	123	128	133	138	143	148	153	158
i	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
$PL_1(i)$	163	168	173	178	183	188	193	198	203	208	213	218	223	228	233	238
i	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
$PL_1(i)$	243	248	253	258	263	268	273	278	283	288	293	298	303	308	313	318
i	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271
$PL_1(i)$	4	9	14	19	24	29	34	39	44	49	54	59	64	69	74	79
i	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287
$PL_1(i)$	84	89	94	99	104	109	114	119	124	129	134	139	144	149	154	159
i	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303
$PL_1(i)$	164	169	174	179	184	189	194	199	204	209	214	219	224	229	234	239
i	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319
$PL_1(i)$	244	249	254	259	264	269	274	279	284	289	294	299	304	309	314	319

Table A.2: Bit permutation $\mathbf{PL}_2$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$PL_2(i)$	51	68	178	215	285	52	69	179	216	286	53	70	180	217	287	54
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$PL_2(i)$	71	181	218	272	55	72	182	219	273	56	73	183	220	274	57	74
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$PL_2(i)$	184	221	275	58	75	185	222	276	11	116	154	231	309	12	117	155
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$PL_2(i)$	232	310	13	118	156	233	311	14	119	157	234	312	15	120	158	235
i	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
$PL_2(i)$	313	0	121	159	236	314	1	122	144	237	315	2	123	145	238	316
i	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
$PL_2(i)$	19	76	170	239	269	20	77	171	224	270	21	78	172	225	271	22
i	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
$PL_2(i)$	79	173	226	256	23	64	174	227	257	24	65	175	228	258	25	66
i	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
$PL_2(i)$	160	229	259	26	67	161	230	260	35	124	162	223	261	36	125	163
i	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
$PL_2(i)$	208	262	37	126	164	209	263	38	127	165	210	264	39	112	166	211
i	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
$PL_2(i)$	265	40	113	167	212	266	41	114	168	213	267	42	115	169	214	268
i	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
$PL_2(i)$	43	92	130	199	301	44	93	131	200	302	45	94	132	201	303	46
i	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
$PL_2(i)$	95	133	202	288	47	80	134	203	289	32	81	135	204	290	33	82
i	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
$PL_2(i)$	136	205	291	34	83	137	206	292	59	100	138	255	317	60	101	139
i	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
$PL_2(i)$	240	318	61	102	140	241	319	62	103	141	242	304	63	104	142	243
i	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
$PL_2(i)$	305	48	105	143	244	306	49	106	128	245	307	50	107	129	246	308
i	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
$PL_2(i)$	27	84	186	247	277	28	85	187	248	278	29	86	188	249	279	30
i	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271
$PL_2(i)$	87	189	250	280	31	88	190	251	281	16	89	191	252	282	17	90
i	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287
$PL_2(i)$	176	253	283	18	91	177	254	284	3	108	146	207	293	4	109	147
i	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303
$PL_2(i)$	192	294	5	110	148	193	295	6	111	149	194	296	7	96	150	195
i	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319
$PL_2(i)$	297	8	97	151	196	298	9	98	152	197	299	10	99	153	198	300

The algebraic normal form of the S-box is as follows.

$$
\begin{aligned}
y_0 &= x_0 \oplus x_1x_3 \oplus x_2x_3 \oplus x_3x_4 \oplus x_4 \\
y_1 &= x_0x_1 \oplus x_0x_3 \oplus x_0 \oplus x_1x_3 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \\
y_2 &= x_0x_2 \oplus x_1 \oplus x_2x_4 \oplus x_2 \oplus x_3 \\
y_3 &= x_0 \oplus x_2x_3 \oplus x_2 \oplus x_3x_4 \oplus x_3 \oplus 1 \\
y_4 &= x_0x_4 \oplus x_0 \oplus x_1 \oplus x_3
\end{aligned}
$$

Table A.3: DDT of S-box

δ \ Δ	0	1	2	t3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	4	0	4	0	4	0	4	0	0	0	0	0	0	0	0	0	4	0	4	0	4	0	4
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	8	8	8	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2
4	0	0	0	8	0	0	0	8	0	0	8	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	2	0	2	0	2	0	2	2	0	2	0	2	0	2	0	0	2	0	2	0	2	0	2	2	0	2	0	2	0	2	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	4	0	4	0	4	0	4	0	4	0	4	0	4	0
7	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4	0	0	0	0	4	4	4	4
9	0	0	0	0	2	2	2	2	2	2	2	2	0	0	0	0	2	2	2	2	0	0	0	0	0	0	0	0	2	2	2	2
10	0	8	8	0	0	0	0	0	8	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	4	4	0	0	4	4	0	0	0	0	0	0	4	4	0	0	0	0	0	0	0	0	0	0	4	4	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
13	0	0	2	2	2	2	0	0	2	2	0	0	0	0	2	2	0	0	2	2	2	2	0	0	2	2	0	0	0	0	2	2
14	0	4	4	0	0	4	4	0	4	0	0	4	4	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	2
16	0	0	0	4	0	0	0	4	0	0	4	0	0	0	4	0	0	0	0	4	0	0	0	4	0	0	4	0	0	0	4	0
17	0	4	0	4	0	0	0	0	4	0	4	0	0	0	0	0	0	4	0	4	0	0	0	0	4	0	4	0	0	0	0	0
18	0	0	0	4	0	0	0	4	0	4	0	0	0	4	0	0	0	4	0	0	0	4	0	0	0	0	0	4	0	0	0	4
19	0	0	0	0	4	0	4	0	0	0	0	0	4	0	4	0	0	0	0	0	4	0	4	0	0	0	0	0	4	0	4	0
20	0	0	0	0	0	0	0	0	0	8	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	8	0	0
21	0	0	0	0	8	0	8	0	0	0	0	0	0	0	0	0	8	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	4	4	0	0	4	4	0	0	0	0	0	0	0	0	0	4	0	0	4	4	0	0	4
23	0	0	0	0	4	4	4	4	0	0	0	0	0	0	0	0	4	4	4	4	0	0	0	0	0	0	0	0	0	0	0	0
24	0	2	2	0	0	2	2	0	2	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	0	2	2	0	0	2	2	0
25	0	0	0	0	2	2	2	2	0	0	0	0	2	2	2	2	0	0	0	0	2	2	2	2	0	0	0	0	2	2	2	2
26	0	2	2	0	0	2	2	0	2	0	0	2	2	0	0	2	0	2	2	0	0	2	2	0	2	0	0	2	2	0	0	2
27	0	0	4	4	0	0	0	0	0	0	4	4	0	0	0	0	0	0	4	4	0	0	0	0	0	0	4	4	0	0	0	0
28	0	4	4	0	0	0	0	0	0	0	0	0	4	0	0	4	0	0	0	0	4	0	0	4	0	4	4	0	0	0	0	0
29	0	0	0	0	2	2	2	2	0	0	0	0	2	2	2	2	2	2	2	2	0	0	0	0	2	2	2	2	0	0	0	0
30	0	4	4	0	0	0	0	0	0	0	0	0	4	0	0	4	0	0	0	0	0	4	4	0	4	0	0	4	0	0	0	0
31	0	0	0	0	4	4	0	0	0	0	0	0	4	4	0	0	4	4	0	0	0	0	0	0	4	4	0	0	0	0	0	0

Table A.4: LAT of S-box

Input Mask \ Output Mask	0	1	2	t3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	-4	4	0	0	0	0	0	0	-4	4	0	0	0	0	0	8	-4	-4	0	0	0	0	0	-8	-4	-4	0
2	0	0	0	0	0	-4	4	0	0	0	0	-8	4	0	0	4	0	4	0	4	0	0	4	4	-4	0	4	0	0	0	4	-4
3	0	0	0	0	0	0	0	0	0	0	0	-8	-4	-4	4	-4	0	-4	0	-4	0	4	0	4	4	0	-4	0	0	4	0	-4
4	0	0	0	4	0	0	0	4	0	0	-4	0	0	0	-4	0	0	-4	-4	4	8	4	4	-4	0	4	0	0	0	4	0	0
5	0	0	0	-4	0	-4	-4	4	-8	0	-4	0	0	4	0	0	0	-4	4	4	0	0	0	4	0	-4	0	0	0	0	-4	0
6	0	0	0	4	0	4	-4	4	0	0	4	0	-4	0	4	4	0	0	4	0	0	4	0	0	-4	4	4	0	0	-4	-4	-4
7	0	8	0	4	0	0	0	-4	0	0	-4	0	-4	4	0	-4	0	0	-4	0	0	0	-4	0	-4	-4	4	0	0	0	0	-4
8	0	0	0	0	0	0	4	4	0	0	0	8	0	0	4	-4	0	0	0	0	0	0	4	4	0	0	0	-8	0	0	4	-4
9	0	0	0	0	0	4	0	4	0	0	0	-8	0	4	0	-4	0	0	0	0	0	-4	0	-4	0	0	0	-8	0	-4	0	4
10	0	0	0	0	8	-4	0	4	0	0	0	0	-4	0	-4	0	8	4	0	-4	0	0	0	0	4	0	4	0	0	0	0	0
11	0	0	0	0	0	0	-4	-4	0	0	0	0	-4	-4	0	0	8	-4	0	4	0	-4	4	0	-4	0	-4	0	0	-4	4	0
12	0	0	0	4	0	0	-4	0	0	0	4	0	0	0	0	-4	0	4	4	4	0	4	0	0	0	-4	0	0	-8	4	4	4
13	0	0	0	-4	0	4	0	0	-8	0	4	0	0	-4	-4	-4	0	4	-4	4	0	0	-4	0	0	4	0	0	0	0	0	-4
14	0	0	0	4	8	4	0	0	0	0	-4	0	4	0	0	0	0	0	4	0	0	-4	-4	4	-4	4	-4	0	0	4	0	0
15	0	-8	0	4	0	0	4	0	0	0	4	0	-4	4	-4	0	0	0	-4	0	0	0	0	4	-4	-4	-4	0	0	0	-4	0
16	0	0	0	4	0	-4	4	4	0	0	-4	0	0	-4	0	0	0	0	0	4	-8	4	-4	-4	0	0	-4	0	0	-4	0	0
17	0	8	0	-4	0	0	0	4	8	0	4	0	0	0	-4	0	0	0	0	4	0	0	0	4	0	0	-4	0	0	0	-4	0
18	0	0	0	-4	0	0	0	-4	0	0	-4	0	-4	-4	0	4	0	4	0	0	0	4	0	0	-4	0	0	-8	0	4	-4	4
19	0	0	0	-4	0	4	4	4	0	0	-4	0	-4	0	4	-4	0	4	0	0	0	0	4	0	-4	0	0	8	0	0	0	4
20	0	0	-8	0	0	4	4	0	0	-8	0	0	0	-4	-4	0	0	-4	4	0	0	0	0	0	0	-4	4	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	0	8	4	-4	0	0	-4	-4	0	8	-4	4	0
22	0	0	0	0	0	0	0	0	0	8	0	0	4	-4	-4	-4	0	0	4	-4	0	0	4	-4	-4	-4	0	0	0	0	-4	-4
23	0	0	8	0	0	4	4	0	0	0	0	0	-4	0	0	4	0	0	4	4	0	-4	0	-4	4	-4	0	0	0	4	0	-4
24	0	0	0	-4	0	4	0	0	0	0	-4	0	0	4	-4	4	0	0	0	-4	0	4	0	0	0	0	-4	0	-8	-4	4	-4
25	0	8	0	4	0	0	4	0	-8	0	4	0	0	0	0	4	0	0	0	-4	0	0	4	0	0	0	-4	0	0	0	0	4
26	0	0	0	4	0	0	-4	0	0	0	-4	0	-4	-4	-4	0	-8	4	0	0	0	-4	4	4	4	0	0	0	0	-4	0	0
27	0	0	0	4	-8	4	0	0	0	0	-4	0	4	0	0	0	8	4	0	0	0	0	0	4	4	0	0	0	0	0	-4	0
28	0	0	0	0	0	-4	0	-4	0	-8	0	0	0	4	0	-4	0	4	4	0	0	0	4	-4	0	4	-4	0	0	0	-4	-4
29	0	0	8	0	0	0	4	-4	0	0	0	0	0	0	-4	-4	0	-4	4	0	0	4	0	4	0	4	4	0	0	-4	0	4
30	0	0	8	0	0	0	-4	4	0	-8	0	0	4	-4	0	0	0	0	-4	-4	0	0	0	0	-4	-4	0	0	0	0	0	0
31	0	0	0	0	8	4	0	-4	0	0	0	0	4	0	4	0	0	0	-4	4	0	4	4	0	4	-4	0	0	0	-4	-4	0

A.3 Bit-sliced Representation of the Permutation

Algorithm 7 Sycon Π^{14}

procedure Sycon
  Input: $S = A_0 \| A_1 \| A_2 \| A_3 \| A_4$ and $\{\mathsf{rc}_i\}$
  Output: S
  for $i = 0...13$ **do**
`//Sbox Layer`
    $t_0 = A_2 \oplus A_4$
    $t_1 = t_0 \oplus A_1$
    $t_2 = A_1 \oplus A_3$
    $t_3 = A_0 \oplus A_4$
    $t_4 = t_3 \oplus (t_1 \wedge A_3)$
    $A_1 \leftarrow ((\neg A_1) \wedge A_3) \oplus t_1 \oplus ((\neg t_2) \wedge A_0)$
    $t_1 = ((\neg t_3) \wedge A_2) \oplus t_2$
    $A_3 \leftarrow ((\neg t_0) \wedge A_3) \oplus A_0 \oplus (\neg A_2)$
    $A_4 \leftarrow ((\neg A_4) \wedge A_0) \oplus t_2$
    $A_0 \leftarrow t_4;\ A_2 \leftarrow t_1;$
`//SubBlockDiffusion Layer`
    $A_0 \leftarrow A_0 \oplus (A_0 \lll 11) \oplus (A_0 \lll 22)$
    $A_1 \leftarrow A_1 \oplus (A_1 \lll 13) \oplus (A_1 \lll 26)$
    $A_2 \leftarrow A_2 \oplus (A_2 \lll 31) \oplus (A_2 \lll 62)$
    $A_3 \leftarrow A_3 \oplus (A_3 \lll 56) \oplus (A_3 \lll 60)$
    $A_4 \leftarrow A_4 \oplus (A_4 \lll 6) \oplus (A_4 \lll 12)$
`//AddRoundConstant Layer`
    $A_2 \leftarrow A_2 \oplus \mathsf{rc}_i$
`//PLayer` (P)
    $A_0 \leftarrow \texttt{ROT16}(A_0, 11)$
    $A_0 \leftarrow \texttt{ByteShuffle}(A_0, \pi_0)$
    $A_1 \leftarrow \texttt{ROT16}(A_1, 4)$
    $A_1 \leftarrow \texttt{ByteShuffle}(A_1, \pi_1)$
    $A_2 \leftarrow \texttt{ROT16}(A_2, 10)$
    $A_2 \leftarrow \texttt{ByteShuffle}(A_2, \pi_2)$
    $A_3 \leftarrow \texttt{ROT16}(A_3, 7)$
    $A_3 \leftarrow \texttt{ByteShuffle}(A_3, \pi_3)$
    $A_4 \leftarrow \texttt{ROT16}(A_4, 5)$
    $A_4 \leftarrow \texttt{ByteShuffle}(A_4, \pi_4)$
  end for
  Set $S \leftarrow (A_0, A_1, A_2, A_3, A_4)$
end procedure