SILC: SImple Lightweight CFB *

Name: SILC v1

Designers/Submitters: Tetsu Iwata, Nagoya University, Japan
Kazuhiko Minematsu, NEC Corporation, Japan
Jian Guo, Nanyang Technological University, Singapore
Sumio Morioka, NEC Europe Ltd., United Kingdom
Eita Kobayashi, NEC Corporation, Japan

Contact Address: Tetsu Iwata, iwata@cse.nagoya-u.ac.jp

Date: March 15, 2014

* This document was prepared based on [6,7,8] and [2,5].

1 Specification

SILC (which stands for SImple Lightweight CFB, and is pronounced as "silk") is a blockcipher mode of operation for authenticated encryption with associated data (AEAD), which is also called an authenticated cipher. SILC is built upon CLOC [6,7,8], and the design of SILC aims at optimizing the hardware implementation cost of CLOC. SILC also maintains the provable security based on the pseudorandomness of the underlying blockcipher. SILC is suitable for use within constrained hardware devices.

1.1 Notation

We use the same notation as in [8], but we repeat the notation for completeness.

Let $\{0,1\}^*$ be the set of all finite bit strings, including the empty string ε. For an integer $\ell \geq 0$, let $\{0,1\}^\ell$ be the set of all bit strings of ℓ bits. We let $\mathbb{B} = \{0,1\}^8$ be the set of bytes (8-bit strings), and $\mathbb{B}^*$ be the set of all finite byte strings. For $X, Y \in \{0,1\}^*$, we write $X \,\|\, Y$, (X, Y), or XY to denote their concatenation. For $\ell \geq 0$, we write $0^\ell \in \{0,1\}^\ell$ to denote the bit string that consists of ℓ zeros, and $1^\ell \in \{0,1\}^\ell$ to denote the bit string that consists of ℓ ones. For $X \in \{0,1\}^*$, $|X|$ is its length in bits, and for $\ell \geq 1$, $|X|_\ell = \lceil |X|/\ell \rceil$ is the length in ℓ-bit blocks. For $X \in \{0,1\}^*$ and $\ell \geq 0$ such that $|X| \geq \ell$, $\mathsf{msb}_\ell(X)$ is the most significant (the leftmost) ℓ bits of X. For instance we have $\mathsf{msb}_1(1100) = 1$ and $\mathsf{msb}_3(1100) = 110$. For $X \in \{0,1\}^*$ and $\ell \geq 1$, we write its partition into ℓ-bit blocks as $(X[1], \ldots, X[x]) \stackrel{\ell}{\leftarrow} X$, which is defined as follows. If $X = \varepsilon$, then $x = 1$ and $X[1] \stackrel{\ell}{\leftarrow} X$, where $X[1] = \varepsilon$. Otherwise $X[1], \ldots, X[x] \in \{0,1\}^*$ are unique bit strings such that $X[1] \,\|\, \cdots \,\|\, X[x] = X$, $|X[1]| = \cdots = |X[x-1]| = \ell$, and $1 \leq |X[x]| \leq \ell$.

In what follows, we fix a block length n and a blockcipher $E : \mathcal{K}_E \times \{0,1\}^n \to \{0,1\}^n$, where $\mathcal{K}_E$ is a non-empty set of keys. Let $\mathrm{Perm}(n)$ be the set of all permutations over $\{0,1\}^n$. We write $E_K \in \mathrm{Perm}(n)$ for the permutation specified by $K \in \mathcal{K}_E$, and $C = E_K(M)$ for the ciphertext of plaintext $M \in \{0,1\}^n$ under key $K \in \mathcal{K}_E$. Following the CAESAR call for submissions, we restrict all input and output variables of SILC as byte-strings. Also we assume the big-endian format for all variables.

1.2 Algorithm and Parameters

We follow the description of CLOC [8].

SILC takes three parameters, a blockcipher $E : \mathcal{K}_E \times \{0,1\}^n \to \{0,1\}^n$, a nonce length ℓ_N, and a tag length τ, where ℓ_N and τ are in bits. Here, a nonce corresponds to a public message number specified by the CAESAR call for submissions, and we may interchangeably use both names. SILC does not have the secret message number, i.e. it is always assumed to be of length zero. We require $1 \leq \ell_N \leq n-1$ and $1 \leq \tau \leq n$, and assume that $\ell_N/8$ and $\tau/8$ are integers, and $n \in \{64, 128\}$. We write $\mathrm{SILC}[E, \ell_N, \tau]$ for SILC that is parameterized by E, ℓ_N, and τ, and we often omit the parameters if they are irrelevant or they are clear from the context. $\mathrm{SILC}[E, \ell_N, \tau] = (\text{SILC-}\mathcal{E}, \text{SILC-}\mathcal{D})$ consists of the encryption algorithm SILC-$\mathcal{E}$ and the decryption algorithm SILC-$\mathcal{D}$.

SILC-$\mathcal{E}$ and SILC-$\mathcal{D}$ have the following syntax.

$$\begin{cases} \text{SILC-}\mathcal{E} : \mathcal{K}_{\mathrm{SILC}} \times \mathcal{N}_{\mathrm{SILC}} \times \mathcal{A}_{\mathrm{SILC}} \times \mathcal{M}_{\mathrm{SILC}} \to \mathcal{CT}_{\mathrm{SILC}} \\ \text{SILC-}\mathcal{D} : \mathcal{K}_{\mathrm{SILC}} \times \mathcal{N}_{\mathrm{SILC}} \times \mathcal{A}_{\mathrm{SILC}} \times \mathcal{CT}_{\mathrm{SILC}} \to \mathcal{M}_{\mathrm{SILC}} \cup \{\perp\} \end{cases}$$

$\mathcal{K}_{\mathrm{SILC}} = \mathcal{K}_E$ is the key space, which is identical to the key space of the underlying blockcipher, $\mathcal{N}_{\mathrm{SILC}} = \mathbb{B}^{\ell_N/8}$ is the nonce space, $\mathcal{A}_{\mathrm{SILC}} = \mathbb{B}^*$ is the associated data space, $\mathcal{M}_{\mathrm{SILC}} = \mathbb{B}^*$ is the plaintext space, $\mathcal{CT}_{\mathrm{SILC}} = \mathcal{C}_{\mathrm{SILC}} \times \mathcal{T}_{\mathrm{SILC}}$ is the ciphertext space, where $\mathcal{C}_{\mathrm{SILC}} = \mathbb{B}^*$ and $\mathcal{T}_{\mathrm{SILC}} = \mathbb{B}^{\tau/8}$ is the tag space, and $\perp \notin \mathcal{M}_{\mathrm{SILC}}$ is the distinguished reject symbol. We write $(C, T) \leftarrow \text{SILC-}\mathcal{E}_K(N, A, M)$ and $M \leftarrow \text{SILC-}\mathcal{D}_K(N, A, C, T)$ or $\perp \leftarrow \text{SILC-}\mathcal{D}_K(N, A, C, T)$. We make a restriction that the maximum lengths of A, M, and C are all $2^{n/2} - 1$ bytes.

SILC-$\mathcal{E}$ and SILC-$\mathcal{D}$ are defined in Fig. 1. In these algorithms, we use four subroutines, HASH, PRF, ENC, and DEC. They have the following syntax.

$$\begin{cases} \mathsf{HASH} : \mathcal{K}_{\mathrm{SILC}} \times \mathcal{N}_{\mathrm{SILC}} \times \mathcal{A}_{\mathrm{SILC}} \to \{0,1\}^n \\ \mathsf{PRF} : \mathcal{K}_{\mathrm{SILC}} \times \{0,1\}^n \times \mathcal{C}_{\mathrm{SILC}} \to \mathcal{T}_{\mathrm{SILC}} \\ \mathsf{ENC} : \mathcal{K}_{\mathrm{SILC}} \times \{0,1\}^n \times \mathcal{M}_{\mathrm{SILC}} \to \mathcal{C}_{\mathrm{SILC}} \\ \mathsf{DEC} : \mathcal{K}_{\mathrm{SILC}} \times \{0,1\}^n \times \mathcal{C}_{\mathrm{SILC}} \to \mathcal{M}_{\mathrm{SILC}} \end{cases}$$

Algorithm SILC-$\mathcal{E}_K(N, A, M)$	**Algorithm** SILC-$\mathcal{D}_K(N, A, C, T)$
1. $V \leftarrow \mathsf{HASH}_K(N, A)$	1. $V \leftarrow \mathsf{HASH}_K(N, A)$
2. $C \leftarrow \mathsf{ENC}_K(V, M)$	2. $T^* \leftarrow \mathsf{PRF}_K(V, C)$
3. $T \leftarrow \mathsf{PRF}_K(V, C)$	3. **if** $T \neq T^*$ **then return** $\perp$
4. **return** (C, T)	4. $M \leftarrow \mathsf{DEC}_K(V, C)$
	5. **return** M

Fig. 1. Pseudocode of the encryption and the decryption algorithms of SILC

Algorithm $\mathsf{HASH}_K(N, A)$	**Algorithm** $\mathsf{PRF}_K(V, C)$
1. $S_\mathsf{H}[0] \leftarrow E_K(\mathsf{zpp}(N))$	1. $S_\mathsf{P}[0] \leftarrow E_K(\mathsf{g}(V))$
2. **if** $\lvert A \rvert = 0$ **then**	2. **if** $\lvert C \rvert = 0$ **then**
3. $\quad V \leftarrow \mathsf{g}(S_\mathsf{H}[0] \oplus \mathsf{Len}(A))$ // $\mathsf{Len}(A) = 0^n$	3. $\quad U \leftarrow \mathsf{g}(S_\mathsf{P}[0] \oplus \mathsf{Len}(C))$ // $\mathsf{Len}(C) = 0^n$
4. $\quad$ **return** V	4. $\quad T \leftarrow \mathsf{msb}_\tau(E_K(U))$
5. $(A[1], \ldots, A[a]) \xleftarrow{n} A$	5. $\quad$ **return** T
6. **for** $i \leftarrow 1$ **to** $a - 1$ **do**	6. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
7. $\quad S_\mathsf{H}[i] \leftarrow E_K(S_\mathsf{H}[i-1] \oplus A[i])$	7. **for** $i \leftarrow 1$ **to** $m - 1$ **do**
8. $S_\mathsf{H}[a] \leftarrow E_K(S_\mathsf{H}[a-1] \oplus \mathsf{zap}(A[a]))$	8. $\quad S_\mathsf{P}[i] \leftarrow E_K(S_\mathsf{P}[i-1] \oplus C[i])$
9. $V \leftarrow \mathsf{g}(S_\mathsf{H}[a] \oplus \mathsf{Len}(A))$	9. $S_\mathsf{P}[m] \leftarrow E_K(S_\mathsf{P}[m-1] \oplus \mathsf{zap}(C[m]))$
10. **return** V	10. $U \leftarrow \mathsf{g}(S_\mathsf{P}[m] \oplus \mathsf{Len}(C))$
	11. $T \leftarrow \mathsf{msb}_\tau(E_K(U))$
	12. **return** T

Algorithm $\mathsf{ENC}_K(V, M)$	**Algorithm** $\mathsf{DEC}_K(V, C)$
1. **if** $\lvert M \rvert = 0$ **then**	1. **if** $\lvert C \rvert = 0$ **then**
2. $\quad C \leftarrow \varepsilon$	2. $\quad M \leftarrow \varepsilon$
3. $\quad$ **return** C	3. $\quad$ **return** M
4. $(M[1], \ldots, M[m]) \xleftarrow{n} M$	4. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
5. $S_\mathsf{E}[1] \leftarrow E_K(V)$	5. $S_\mathsf{D}[1] \leftarrow E_K(V)$
6. **for** $i \leftarrow 1$ **to** $m - 1$ **do**	6. **for** $i \leftarrow 1$ **to** $m - 1$ **do**
7. $\quad C[i] \leftarrow S_\mathsf{E}[i] \oplus M[i]$	7. $\quad M[i] \leftarrow S_\mathsf{D}[i] \oplus C[i]$
8. $\quad S_\mathsf{E}[i+1] \leftarrow E_K(\mathsf{fix1}(C[i]))$	8. $\quad S_\mathsf{D}[i+1] \leftarrow E_K(\mathsf{fix1}(C[i]))$
9. $C[m] \leftarrow \mathsf{msb}_{\lvert M[m] \rvert}(S_\mathsf{E}[m]) \oplus M[m]$	9. $M[m] \leftarrow \mathsf{msb}_{\lvert C[m] \rvert}(S_\mathsf{D}[m]) \oplus C[m]$
10. $C \leftarrow (C[1], \ldots, C[m])$	10. $M \leftarrow (M[1], \ldots, M[m])$
11. **return** C	11. **return** M

Fig. 2. Subroutines used in the encryption and decryption algorithms of SILC

These subroutines are defined in Fig. 2, and illustrated in Fig. 3, Fig. 4, and Fig. 5. We also present equivalent figures in Fig. 6, Fig. 7, and Fig. 8. We note that ENC and DEC are the same as those in CLOC [6]. In the figures, i is the identity function, and $\mathsf{i}(X) = X$ for all $X \in \{0, 1\}^n$. In the subroutines, we use the zero prepending function $\mathsf{zpp} : \mathbb{B}^* \to \mathbb{B}^*$, the zero appending function $\mathsf{zap} : \mathbb{B}^* \to \mathbb{B}^*$, the bit-fixing function $\mathsf{fix1} : \mathbb{B}^* \to \mathbb{B}^*$, the tweak function $\mathsf{g} : \{0, 1\}^n \to \{0, 1\}^n$, and the length encoding function $\mathsf{Len} : \mathbb{B}^* \to \{0, 1\}^n$.

Both the zero prepending and appending functions are used to adjust the length of an input string so that the total length becomes a non-negative multiple of n bits (the output is the empty string if and only if the input is the empty string). For $X \in \mathbb{B}^*$, $\mathsf{zpp}(X)$ is defined as

$$\mathsf{zpp}(X) = \begin{cases} X & \text{if } \lvert X \rvert = \ell n \text{ for some } \ell \geq 0, \\ 0^{n - (\lvert X \rvert \bmod n)} \,\|\, X & \text{otherwise,} \end{cases}$$

and $\mathsf{zap}(X)$ is defined as

$$\mathsf{zap}(X) = \begin{cases} X & \text{if } \lvert X \rvert = \ell n \text{ for some } \ell \geq 0, \\ X \,\|\, 0^{n - (\lvert X \rvert \bmod n)} & \text{otherwise.} \end{cases}$$

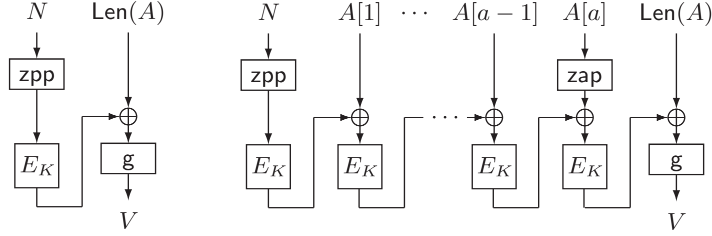

Fig. 3. $V \leftarrow \mathsf{HASH}_K(N, A)$ for $|A| = 0$ (left) and $|A| \geq 1$ (right)

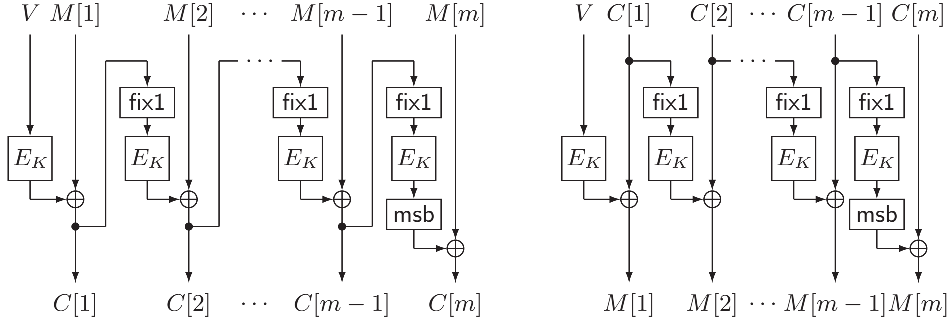

Fig. 4. $C \leftarrow \mathsf{ENC}_K(V, M)$ for $|M| \geq 1$ (left), and $\mathsf{DEC}_K(V, C)$ for $|C| \geq 1$ (right)

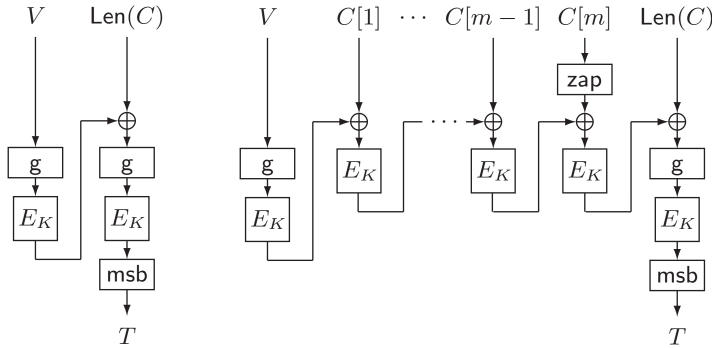

Fig. 5. $T \leftarrow \mathsf{PRF}_K(V, C)$ for $|C| = 0$ (left) and $|C| \geq 1$ (right)

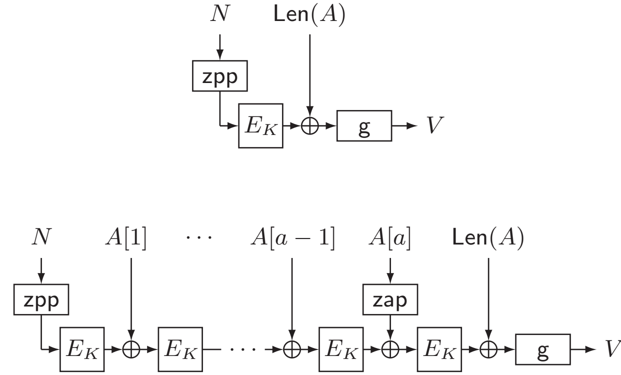

Fig. 6. $V \leftarrow \mathsf{HASH}_K(N, A)$ for $|A| = 0$ (top) and $|A| \geq 1$ (bottom)

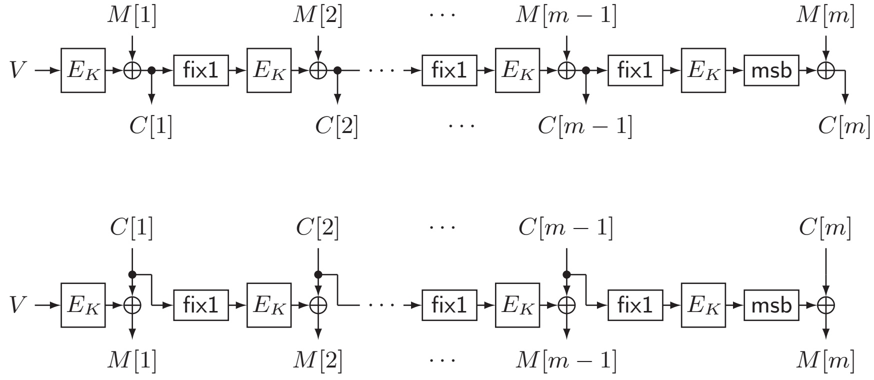

Fig. 7. $C \leftarrow \mathsf{ENC}_K(V, M)$ for $|M| \geq 1$ (top), and $\mathsf{DEC}_K(V, C)$ for $|C| \geq 1$ (bottom)

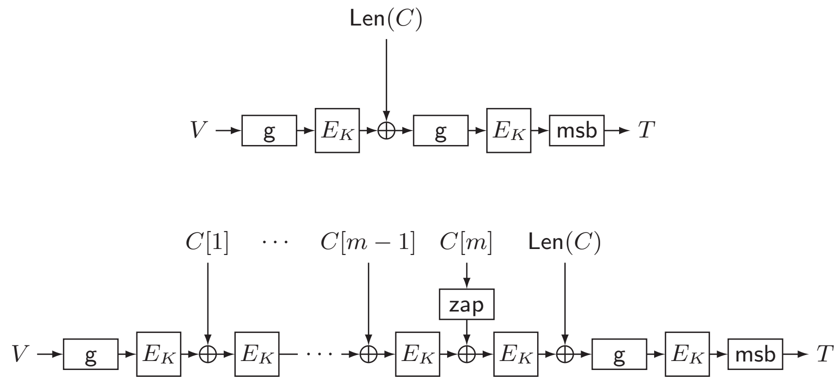

Fig. 8. $T \leftarrow \mathsf{PRF}_K(V, C)$ for $|C| = 0$ (top) and $|C| \geq 1$ (bottom)

In general, they are not invertible functions.

The bit-fixing function $\mathsf{fix1}$ is used to fix the most significant bit of an input string to one. For $X \in \mathbb{B}^*$, $\mathsf{fix1}(X)$ is defined as $\mathsf{fix1}(X) = X \vee 10^{|X|-1}$, where $\vee$ denotes the bit-wise OR operation.

The length encoding function $\mathsf{Len} : \mathbb{B}^* \to \{0,1\}^n$ is used to encode the input length (in bytes) in HASH and PRF. For $X \in \mathbb{B}^*$, it is defined as $\mathsf{Len}(X) = \mathsf{str}_n(|X|_8)$, where $\mathsf{str}_n(|X|_8)$ is the standard encoding of $|X|_8$ (the byte length of X) into an n-bit string. For example, when $X = \varepsilon$, we have $\mathsf{Len}(X) = 0^n$, and when $|X|_8 = 5$, we have $\mathsf{Len}(X) = 0^{n-4} \,\|\, 0101$. As the maximum lengths of A, M, and C are all $2^{n/2} - 1$ bytes, the most significant $n/2$ bits of $\mathsf{Len}(X)$ in HASH and PRF are fixed to $0^{n/2}$.

The tweak function $\mathsf{g} : \{0,1\}^n \to \{0,1\}^n$ is used in HASH and PRF. If $n = 128$, for $X \in \{0,1\}^n$, we let $(X[1], X[2], \ldots, X[16]) \stackrel{n/16}{\leftarrow} X$. Then $\mathsf{g}(X)$ is defined as

$$\mathsf{g}(X) = (X[2], X[3], \ldots, X[16], X[1,2]),$$

where $X[a,b]$ stands for $X[a] \oplus X[b]$. Similarly, if $n = 64$, we let $(X[1], X[2], \ldots, X[8]) \stackrel{n/8}{\leftarrow} X$ and define $\mathsf{g}(X)$ as

$$\mathsf{g}(X) = (X[2], X[3], \ldots, X[8], X[1,2]).$$

For both cases, g can be interpreted as one byte left shift with the rightmost output byte being the xor of the leftmost two input bytes.

1.3 Parameter Spaces

As the CAESAR submission we specify the parameter spaces of SILC as follows.

- Blockcipher E: AES-128 (AES with 128-bit key), or PRESENT-80 (PRESENT with 80-bit key), or LED-80 (LED with 80-bit key).
- Nonce length ℓ_N: For AES-128, $\ell_N \in \{8 \text{ bits (1 byte)}, 16 \text{ bits (2 bytes)}, \ldots, 120 \text{ bits (15 bytes)}\}$, and for PRESENT-80 and LED-80, $\ell_N \in \{8 \text{ bits (1 byte)}, 16 \text{ bits (2 bytes)}, \ldots, 56 \text{ bits (7 bytes)}\}$.
- Tag length τ: For AES-128, $\tau \in \{32 \text{ bits (4 bytes)}, 40 \text{ bits (5 bytes)}, \ldots, 128 \text{ bits (16 bytes)}\}$, and for PRESENT-80 and LED-80, $\tau \in \{32 \text{ bits (4 bytes)}, 40 \text{ bits (5 bytes)}, \ldots, 64 \text{ bits (8 bytes)}\}$.

PRESENT is a 64-bit blockcipher proposed by Bogdanov et al. at CHES 2007 [2], and LED is a 64-bit blockcipher proposed by Guo et al. at CHES 2011 [5]. The specification of PRESENT is described in Appendix A, and that of LED is described in Appendix B.

1.4 Recommended Parameter Sets

We specify the recommended parameter sets as follows.

- Parameter set 1, `aes128n12silcv1`: $E =$ AES-128, $\ell_N = 96$ (12-byte nonce), $\tau = 64$ (8-byte tag)
- Parameter set 2, `aes128n8silcv1`: $E =$ AES-128, $\ell_N = 64$ (8-byte nonce), $\tau = 64$ (8-byte tag)
- Parameter set 3, `present80n6silcv1`: $E =$ PRESENT-80, $\ell_N = 48$ (6-byte nonce), $\tau = 32$ (4-byte tag)
- Parameter set 4, `led80n6silcv1`: $E =$ LED-80, $\ell_N = 48$ (6-byte nonce), $\tau = 32$ (4-byte tag)

2 Security Goals

The security goal of SILC is to provide the provable security in terms of confidentiality (or privacy) of plaintexts under nonce-respecting adversaries, and integrity (or authenticity) of plaintext, associated data, and nonce (public message number) under nonce-reusing adversaries. These goals are the same as CLOC. We note that, as CLOC, SILC has no secret message number.

SILC has provable security guarantees both for confidentiality and integrity, up to the standard birthday bound of the block length of the underlying blockcipher, based on the assumption that the blockcipher is a pseudorandom permutation (PRP). The attack models are given in Sect. 3, which are the same as in CLOC [8]. The exact security bounds and the corresponding proofs are in preparation, and they will be made public once they are finalized.

Table 1. Security goal for confidentiality (privacy)

Parameter set	aes128n12silcv1	aes128n8silcv1	present80n6silcv1	led80n6silcv1
Data	64	64	32	32
Time	128	128	80	80

Table 2. Security goal for integrity (authenticity)

Parameter set	aes128n12silcv1	aes128n8silcv1	present80n6silcv1	led80n6silcv1
Data	64	64	32	32
Verify	64	64	32	32
Time	128	128	80	80

Attack Workload. We confirmed that SILC has provable security bounds up to the standard birthday bound, based on the pseudorandomness of the underlying blockcipher. Table 1 and Table 2 are obtained from these bounds. The variables in the tables denote the required workload of an adversary to break the cipher, in logarithm base 2. If one of the variables reaches the suggested number, then there is no security guarantee anymore, and the cipher can be broken. In Table 1, Data denotes σ_{priv} of our privacy theorem. The theorem corresponds to [8, Theorem 1], and this roughly suggests the number of data blocks that the adversary obtains. In Table 2, Data denotes σ_{auth} and Verify denotes q' of our authenticity theorem, which corresponds to [8, Theorem 2], where σ_{auth} roughly suggests the number of data blocks that the adversary obtains, and q' denotes the number of decryption queries. In both tables, Time denotes the time complexity, which we assume to be equal to the bit length of the key of the underlying blockcipher.

As in CLOC, the nonce cannot be repeated to maintain the privacy. However, the privacy of SILC is kept as long as the uniqueness of (A, N), a pair of associated data and a nonce, is maintained for all encryption queries. We note that the authenticity holds in this setting as well, since it is maintained even if the nonce is reused.

On the Use of 64*-Bit Blockcipher.* We emphasize that the use of 64-bit blockciphers, PRESENT or LED, is not for general purpose applications, since the birthday bound for the block length of 64 bits is usually unacceptable for conventional data transmission, as pointed out by McGrew [11]. However, there are various practical applications that benefit from the low implementation cost even with the limited security guarantee. See [8, Sect. 2] for such examples.

3 Security Analysis

In this section, we define the security notions of a blockcipher and SILC, which are the same as in [8, Sect. 3]

PRP Notion. We assume that the blockcipher $E : \mathcal{K}_E \times \{0,1\}^n \to \{0,1\}^n$ is a pseudorandom permutation, or a PRP [10]. We say that P is a random permutation if $P \overset{\$}{\leftarrow} \text{Perm}(n)$, and define

$$\mathbf{Adv}_E^{\text{prp}}(\mathcal{A}) \overset{\text{def}}{=} \Pr\left[\mathcal{A}^{E_K(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{P(\cdot)} \Rightarrow 1\right],$$

where the first probability is taken over $K \overset{\$}{\leftarrow} \mathcal{K}_E$ and the randomness of $\mathcal{A}$, and the last is over $P \overset{\$}{\leftarrow} \text{Perm}(n)$ and $\mathcal{A}$. We write SILC[Perm$(n), \ell_N, \tau$] for SILC that uses P as E_K, and the encryption and decryption algorithms are written as SILC-$\mathcal{E}_P$ and SILC-$\mathcal{D}_P$.

Privacy Notion. We define the privacy notion for SILC$[E, \ell_N, \tau] = (\text{SILC-}\mathcal{E}, \text{SILC-}\mathcal{D})$. This notion captures the indistinguishably of a nonce-respecting adversary in a chosen plaintext attack setting. We consider an adversary $\mathcal{A}$ that has access to the SILC encryption oracle, or a random-bits oracle. The encryption oracle takes $(N, A, M) \in \mathcal{N}_{\text{SILC}} \times \mathcal{A}_{\text{SILC}} \times \mathcal{M}_{\text{SILC}}$ as input and returns $(C, T) \leftarrow \text{SILC-}\mathcal{E}_K(N, A, M)$. The random-bits oracle, \$-oracle, takes $(N, A, M) \in \mathcal{N}_{\text{SILC}} \times \mathcal{A}_{\text{SILC}} \times \mathcal{M}_{\text{SILC}}$ as input and returns a random string $(C, T) \overset{\$}{\leftarrow} \{0,1\}^{|M|+\tau}$. We define the privacy advantage as

$$\mathbf{Adv}_{\text{SILC}[E,\ell_N,\tau]}^{\text{priv}}(\mathcal{A}) \overset{\text{def}}{=} \Pr\left[\mathcal{A}^{\text{SILC-}\mathcal{E}_K(\cdot,\cdot,\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{\$(\cdot,\cdot,\cdot)} \Rightarrow 1\right],$$

where the first probability is taken over $K \xleftarrow{\$} \mathcal{K}_{\mathrm{SILC}}$ and the randomness of $\mathcal{A}$, and the last is over the random-bits oracle and $\mathcal{A}$. We assume that $\mathcal{A}$ in the privacy game is nonce-respecting, that is, $\mathcal{A}$ does not make two queries with the same nonce.

Authenticity Notion. We next define the authenticity notion, which captures the unforgeability of an adversary in a chosen ciphertext attack setting. We consider a strong adversary that can repeat the same nonce multiple times. Let $\mathcal{A}$ be an adversary that has access to the SILC encryption oracle and the SILC decryption oracle. The encryption oracle is defined as above. The decryption oracle takes $(N, A, C, T) \in \mathcal{N}_{\mathrm{SILC}} \times \mathcal{A}_{\mathrm{SILC}} \times \mathcal{C}_{\mathrm{SILC}} \times \mathcal{T}_{\mathrm{SILC}}$ as input and returns $M \leftarrow \text{SILC-}\mathcal{D}_K(N, A, C, T)$ or $\perp \leftarrow \text{SILC-}\mathcal{D}_K(N, A, C, T)$. The authenticity advantage is defined as

$$\mathbf{Adv}^{\mathrm{auth}}_{\mathrm{SILC}[E,\ell_N,\tau]}(\mathcal{A}) \stackrel{\mathrm{def}}{=} \Pr\left[\mathcal{A}^{\text{SILC-}\mathcal{E}_K(\cdot,\cdot,\cdot),\text{SILC-}\mathcal{D}_K(\cdot,\cdot,\cdot,\cdot)} \text{ forges}\right],$$

where the probability is taken over $K \xleftarrow{\$} \mathcal{K}_{\mathrm{SILC}}$ and the randomness of $\mathcal{A}$, and the adversary forges if the decryption oracle returns a bit string (other than $\perp$) for a query (N, A, C, T), but (C, T) was not previously returned to $\mathcal{A}$ from the encryption oracle for a query (N, A, M). The adversary $\mathcal{A}$ in the authenticity game is not necessarily nonce-respecting, and $\mathcal{A}$ can make two or more queries with the same nonce. Specifically, $\mathcal{A}$ can repeat using the same nonce for encryption queries, a nonce used for encryption queries can be used for decryption queries and vice-versa, and the same nonce can be repeated for decryption queries. Without loss of generality, we assume that $\mathcal{A}$ does not make trivial queries, i.e., if the encryption oracle returns (C, T) for a query (N, A, M), then $\mathcal{A}$ does not make a query (N, A, C, T) to the decryption oracle, and $\mathcal{A}$ does not repeat a query.

Bounds. We confirmed that SILC is provably secure with respect to the above two security notions up to the standard birthday bound security having a small constant. That is, we have security bounds based on the assumption that the underlying blockcipher is a PRP, and hence we have the security guarantee up to about $2^{n/2}$ blocks of data. The proofs are similar to those of CLOC [7], and the exact bounds and the corresponding proofs will be made public once they are finalized.

4 Features

SILC has the following features.

1. It uses only the encryption of the blockcipher both for encryption and decryption.
2. It carefully avoids hardware-unfriendly operations as much as possible, e.g., conditional operation branching, which requires multiplexers in hardware, and dynamic change of data shift amount.
3. It makes $\lceil |N|/n \rceil + \lceil |A|/n \rceil + 2\lceil |M|/n \rceil + 2$ blockcipher calls for a nonce N, associated data A, and a plaintext M, when $|A| \geq 1$. No precomputation other than the blockcipher key scheduling is needed. As a result, no extra hardware register for storing the precomputed result is necessary. We note that in SILC, $1 \leq |N| \leq n - 1$ holds (hence we always have $\lceil |N|/n \rceil = 1$), and when $|A| = 0$, it needs $\lceil |N|/n \rceil + 1 + 2\lceil |M|/n \rceil + 2$ blockcipher calls.
4. The memory cost other than the blockcipher is low. It works with two state blocks (i.e. $2n$ bits) to store chaining blocks for encryption and authentication, plus a counter for storing the message length.
5. Both encryption and decryption can be processed in an online manner.
6. For security, the privacy and authenticity are proved based on the PRP assumption of the blockcipher, assuming standard nonce-respecting adversaries. Moreover, the authenticity is proved with even stronger, nonce-reusing adversaries.

The first, second, and fourth features imply SILC's suitability for small hardware. SILC essentially consists a blockcipher encryption function E_K and other functions, zpp, zap, fix1, Len, and g. These functions are chosen by taking the hardware efficiency into account. For instance the 10^* padding function is commonly used in many blockcipher modes, but due to the operation branch depending on the input length, it imposes non-negligible increase in circuit gates compared with zpp or zap. At the cost of one additional blockcipher call for Len, the padding is significantly simplified.

The last feature implies that SILC provides standard security as a nonce-based AEAD, and in addition a level of security (i.e. authenticity only) even when the nonce is reused.

Advantages over AES-GCM. Compared with AES-GCM [12], the hardware implementation of SILC with AES can be smaller, since we avoid using a full Galois-Field (GF) multiplier. In hardware, AES-GCM is generally fast, however, a fast GF multiplier requires a rather large number of gates, in addition to those needed for the AES encryption function. While SILC with AES can be efficiently implemented, it is also fast if AES is fast. For SILC with PRESENT or LED, we expect even smaller implementations with reduced power consumption, at the cost of reduced security which is reasonable for constrained hardware. The parameter set with PRESENT or LED would be beneficial to tiny devices, such as RFID or CPLD.

With respect to the security, SILC inherits the advantages of CLOC over GCM. That is, the provable security bound of SILC for authentication is better than that of GCM presented in [9]. In GCM, the existence of weak keys was pointed out [14], while weak keys are not known in SILC. Also, SILC provides some level of security even if the nonce is reused.

Justifications of Parameter Sets. For the 128-bit blockcipher, we select AES for its excellent performance and extensively studied security. For the 64-bit blockcipher, we select PRESENT and LED. Both ciphers can be implemented with small gate size, and in particular, PRESENT is selected for its high throughput, and LED is selected for its high security margin against various cryptanalysis.

For `aes128n12silcv1`, we select $\ell_N = 96$ from the current trend on the length of the nonce, and this is suitable, for instance, if a part of the nonce is randomly chosen and the other part consists of a counter. For `aes128n8silcv1`, we select $\ell_N = 64$ considering the data overhead, and this is suitable for applications where the nonce consists of a counter. For `present80n6silcv1` and `led80n6silcv1`, we select $\ell_N = 48$ by taking the half of 96 in `aes128n12silcv1`. For all cases, the tag length was chosen by taking the balance between the security and the data overhead.

Limitations. We list several limitations of SILC. SILC is designed to reduce the hardware gates of CLOC as much as possible, while maintaining the provable security based on the pseudorandomness of the underlying blockcipher, at the cost of constant increase in the number of blockcipher calls. Also, it does not handle static associated data efficiently, as we first process a nonce and then associated data. We chose this order as the small hardware is the main target of SILC, and hence it is unlikely that we keep the intermediate state block to improve the efficiency. SILC also inherits limitations of CLOC. For long input data, SILC is not efficient as it needs two blockcipher calls per one plaintext block. The nonce length is fixed, which may be problematic in some applications. The four functions used in SILC, HASH, ENC, DEC, and HASH, are all sequential, but the blockcipher calls in ENC and PRF can be done in parallel. We also note that the parallelization is always possible for multiple messages [4,3].

5 Design Rationale

The designers have not hidden any weaknesses in this cipher.

Our goal is to provide an AEAD particularly efficient for hardware, requiring a small number of gates other than the blockcipher implementation, that is, a small implementation overhead. For achieving hardware efficiency, we set our design strategy as follows.

- Construct data flow with minimized kinds/amount of functions, minimized flow branching and merging, which implies extra multiplexers and registers, and the use of same ordering of functions in different steps, which makes hardware sharing easy.
- Avoid functions not suitable to hardware, such as dynamic data shifting, which requires a barrel shifter, and integer operations etc.
- Avoid to use many pre-computed values, which consumes extra registers.

SILC is built upon CLOC, and inherits the overall structure. Basically, SILC is a combination of CFB and CBC MAC, where CBC MAC is called twice for processing associated data and a ciphertext, and CFB is called once to generate a ciphertext. In order to keep implementation overhead as small as possible, we choose CFB, since CBC needs the decryption of the blockcipher, and CTR or OFB requires additional state for counter or intermediate output block. Since a naive combination of CFB and CBC MAC does not work, and we do not want to use precomputed blockcipher outputs such as $L = E_K(0^n)$ used in EAX, as this requires additional memory state, we use fix1 and zpp functions to logically separate CFB and CBC MAC. Here, instead of zpp, any function that forces the first input bit to CBC MAC

to zero would work, however, we choose zpp for its simplicity in hardware. This loses the capability of efficient handling of static associated data, but we think this is the right treading-off between the size and simplicity, considering our target (e.g. it is unlikely for small hardware to have a memory block and a control logic for caching static associated data).

For the tweak function, as in CLOC, we avoid using GF doubling (a multiplication by two over $GF(2^n)$), a common operation for many blockcipher modes [1,15]. Instead, we have adopted the g function to reduce the hardware logic size. When implemented as combinational circuits, the g function is much simpler than the GF doubling because it consists of a static amount of shifting, which consumes no hardware resources, and a minimum amount of xors. The role of the g function is to tweak an input value of the blockcipher, and a similar technique can be found in the context of MAC [13,16]. There is only one tweak function in SILC, which is different from CLOC [6] that has five tweak functions. This means the hardware implementation of SILC does not need many selectors. The tweak function is selected so that it satisfies the following conditions, which is needed for provable security. First, it is linear with respect to xor (i.e. $\mathsf{g}(X \oplus X') = \mathsf{g}(X) \oplus \mathsf{g}(X')$ holds for all $X, X' \in \{0,1\}^n$). Next, it is invertible over $\{0,1\}^n$. Finally, let $K \in \{0,1\}^n$ be uniform over $\{0,1\}^n$. Then, we require that the following functions are (close to) uniform over $\{0,1\}^n$.

$$\begin{cases} \mathsf{g}(K) \\ \mathsf{g}(K) \oplus K \\ \mathsf{g}(\mathsf{g}(K)) \\ \mathsf{g}(\mathsf{g}(K)) \oplus K \\ \mathsf{g}(\mathsf{g}(K)) \oplus \mathsf{g}(K) \end{cases}$$

It can be easily confirmed that our g function fulfills these conditions for both $n = 64$ and $n = 128$ by computing the corresponding matrix ranks over GF(2) as was done in [6].

At the end of HASH and PRF, we use a simple padding function with additional length encoding. Though this always requires one additional blockcipher call compared to popular 10^* padding used by many blockcipher modes, the former is much more efficient in terms of the gate size. We remark that our padding scheme here is similar to the one used in GCM.

Selection of Blockciphers. For 128-bit block size we choose AES as the underlying blockcipher, because the security of AES has been extensively studied. For 64-bit block size we choose PRESENT and LED as the underlying blockcipher. As explained in Sect. 4, both ciphers are chosen for their small hardware size, and we think PRESENT is useful when the application requires high throughput, and LED is useful when long-term security is required, where LED's high security margin will help.

6 Intellectual Property

We claim no intellectual property (IP) rights associated to SILC, and are unaware of any relevant IP held by others. We note that the statement does not cover the internal blockcipher. Nanyang Technological University has a patent related to LED blockcipher: WO2012154129 A1.

If any of this information changes, the submitter will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

7 Consent

The submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitters acknowledge that

the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

Acknowledgments. The work by Tetsu Iwata was carried out in part while visiting Nanyang Technological University, Singapore. The work by Jian Guo was supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Bellare, M., Rogaway, P., Wagner, D.: The EAX Mode of Operation. In: Roy, B.K., Meier, W. (eds.) FSE. Lecture Notes in Computer Science, vol. 3017, pp. 389–407. Springer (2004)
2. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES. Lecture Notes in Computer Science, vol. 4727, pp. 450–466. Springer (2007)
3. Bogdanov, A., Lauridsen, M., Tischhauser, E.: AES-Based Authenticated Encryption Modes in Parallel High-Performance Software. Cryptology ePrint Archive, Report 2014 (2014)
4. Bogdanov, A., Mendel, F., Regazzoni, F., Rijmen, V., Tischhauser, E.: ALE: AES-Based Lightweight Authenticated Encryption. Pre-proceedings of Fast Software Encryption 2013 (2013)
5. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED Block Cipher. In: Preneel, B., Takagi, T. (eds.) CHES. Lecture Notes in Computer Science, vol. 6917, pp. 326–341. Springer (2011), an updated version available `http://eprint.iacr.org/2012/600.pdf`
6. Iwata, T., Minematsu, K., Guo, J., Morioka, S.: CLOC: Authenticated Encryption for Short Input. Pre-proceedings of FSE 2014 (2014)
7. Iwata, T., Minematsu, K., Guo, J., Morioka, S.: CLOC: Authenticated Encryption for Short Input. Cryptology ePrint Archive, Report 2014 (2014), full version of FSE 2014 paper, `http://eprint.iacr.org/`
8. Iwata, T., Minematsu, K., Guo, J., Morioka, S.: CLOC: Compact Low-Overhead CFB. Submission to the CAESAR competition (2014), CLOC v1, March 15, 2014
9. Iwata, T., Ohashi, K., Minematsu, K.: Breaking and Repairing GCM Security Proofs. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO. Lecture Notes in Computer Science, vol. 7417, pp. 31–49. Springer (2012)
10. Luby, M., Rackoff, C.: How to Construct Pseudorandom Permutations from Pseudorandom Functions. SIAM J. Comput. 17(2), 373–386 (1988)
11. McGrew, D.: Impossible Plaintext Cryptanalysis and Probable-Plaintext Collision Attacks of 64-bit Block Cipher Modes. Pre-proceedig of FSE 2013 (2013)
12. McGrew, D.A., Viega, J.: The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In: Canteaut, A., Viswanathan, K. (eds.) INDOCRYPT. Lecture Notes in Computer Science, vol. 3348, pp. 343–355. Springer (2004)
13. Nandi, M.: Fast and Secure CBC-Type MAC Algorithms. In: Dunkelman, O. (ed.) FSE. Lecture Notes in Computer Science, vol. 5665, pp. 375–393. Springer (2009)
14. Procter, G., Cid, C.: On Weak Keys and Forgery Attacks against Polynomial-Based MAC Schemes. Pre-proceeding of FSE 2013 (2013)
15. Rogaway, P.: Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In: Lee, P.J. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 3329, pp. 16–31. Springer (2004)
16. Zhang, L., Wu, W., Zhang, L., Wang, P.: CBCR: CBC MAC with rotating transformations. SCIENCE CHINA Information Sciences 54(11), 2247–2255 (2011)

A PRESENT [2]

PRESENT is a blockcipher with 80-bit or 128-bit keys, and employs the SP-network. We describe the 80-bit key version, which we write PRESENT-80, using the materials in [2].

It consists of 31 rounds, and each of the 31 rounds consists of an xor operation of a round key K_i for $1 \leq i \leq 32$, where K_{32} is used for post-whitening, a linear bitwise permutation, and a non-linear substitution layer. The non-linear layer uses a single 4-bit S-box S which is applied 16 times in parallel in each round. The cipher is described in the following pseudocode.

1. generateRoundKeys()

2. **for** $i \leftarrow 1$ **to** 31 **do**
3. addRoundKey(STATE, K_i)
4. sBoxLayer(STATE)
5. pLayer(STATE)
6. **end for**
7. addRoundKey(STATE, K_{32})

Throughout this section, we number bits from zero with bit zero on the right of a block or word. Each stage is specified below.

addRoundKey. Given round key $K_i = \kappa^i_{63} \dots \kappa^i_0$ for $1 \le i \le 32$ and current STATE $b_{63} \dots b_0$, addRoundKey consists of the operation for $0 \le j \le 63$,

$$b_j \rightarrow b_j \oplus \kappa^i_j.$$

sBoxlayer. The S-box used in PRESENT is a 4-bit to 4-bit S-box $S : \{0,1\}^4 \rightarrow \{0,1\}^4$. The following table shows the input and output of the S-box in hexadecimal notation.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

For sBoxLayer the current STATE $b_{63} \dots b_0$ is considered as sixteen 4-bit words $w_{15} \dots w_0$ where $w_i = b_{4*i+3} \,\|\, b_{4*i+2} \,\|\, b_{4*i+1} \,\|\, b_{4*i}$ for $0 \le i \le 15$ and the output nibble $S[w_i]$ provides the update state values in the obvious way.

pLayer. The bit permutation used in PRESENT is given by the following table. Bit i of STATE is moved to bit position $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

The key schedule. PRESENT can take keys of either 80 or 128 bits. In the 80-bit key version, the user-supplied key is stored in a key register K and represented as $k_{79}k_{78} \dots k_0$. At round i the 64-bit round key $K_i = \kappa_{63}\kappa_{62} \dots \kappa_0$ consists of the 64 leftmost bits of the current contents of register K. Thus at round i we have that:

$$K_i = \kappa_{63}\kappa_{62} \dots \kappa_0 = k_{79}k_{78} \dots k_{16}.$$

After extracting the round key K_i, the key register $K = k_{79}k_{78} \dots k_0$ is updated as follows.

1. $[\mathtt{k}_{79}\mathtt{k}_{78} \dots \mathtt{k}_1\mathtt{k}_0] = [\mathtt{k}_{18}\mathtt{k}_{17} \dots \mathtt{k}_{20}\mathtt{k}_{19}]$
2. $[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}] = S[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}]$
3. $[\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] = [\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] \oplus$ `round_counter`

Thus, the key register is rotated by 61 bit positions to the left, the left-most four bits are passed through the PRESENT S-box, and the `round_counter` value i is xor'ed with bits $\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}$ of K with the least significant bit of `round_counter` on the right.

B LED [5]

LED [5] is a 64-bit lightweight blockcipher family designed by Guo et al. in 2011, consists of mainly two variants of 64-bit and 128-bit key, denoted as LED-64 and LED-128, respectively. The 64-bit plaintext m is split into 16 4-bit nibbles $m_0\|m_1\|\ldots\|m_{15}$, and can be represented in a square array as:

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

LED is AES like, and every round function consists of 4 operations: SubByte, ShiftRow, MixColumn, and AddConstant.

SubByte applies the PRESENT S-box, as already described in Appendix A, to every nibble, i.e., $m_i = S(m_i)$ for $i = 0, \ldots, 15$.

ShiftRow shifts the i-th row to the left by i positions for $i = 0, \ldots, 3$, and the resulted matrix becomes

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix} \leftarrow \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_5 & m_6 & m_7 & m_4 \\ m_{10} & m_{11} & m_8 & m_9 \\ m_{15} & m_{12} & m_{13} & m_{14} \end{bmatrix}$$

MixColumn applies Galois-Field multiplication, with irreducible polynomial $f(x) = x^4 + x + 1$, of MDS matrix to each column. The MDS matrix is defined as

$$M = (A)^4 = \begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 4\,1\,2\,2 \end{pmatrix}^4 = \begin{pmatrix} 4\,1\,2\,2 \\ 8\,6\,5\,6 \\ \mathtt{B}\,\mathtt{E}\,\mathtt{A}\,9 \\ 2\,2\,\mathtt{F}\,\mathtt{B} \end{pmatrix}.$$

Then for $i = 0, 1, 2, 3$,

$$\begin{bmatrix} m_{i+0} \\ m_{i+4} \\ m_{i+8} \\ m_{i+12} \end{bmatrix} = M \times \begin{bmatrix} m_{i+0} \\ m_{i+4} \\ m_{i+8} \\ m_{i+12} \end{bmatrix}.$$

AddConstant adds a round-dependent value rc and key-size dependent value ks (ks is an 8-bit representation of the master key size) to the state. The constant format is as follows.

$$\begin{bmatrix} 0 \oplus (\mathrm{ks}_7\|\mathrm{ks}_6\|\mathrm{ks}_5\|\mathrm{ks}_4) & (\mathrm{rc}_5\|\mathrm{rc}_4\|\mathrm{rc}_3) & 0 & 0 \\ 1 \oplus (\mathrm{ks}_7\|\mathrm{ks}_6\|\mathrm{ks}_5\|\mathrm{ks}_4) & (\mathrm{rc}_2\|\mathrm{rc}_1\|\mathrm{rc}_0) & 0 & 0 \\ 2 \oplus (\mathrm{ks}_3\|\mathrm{ks}_2\|\mathrm{ks}_1\|\mathrm{ks}_0) & (\mathrm{rc}_5\|\mathrm{rc}_4\|\mathrm{rc}_3) & 0 & 0 \\ 3 \oplus (\mathrm{ks}_3\|\mathrm{ks}_2\|\mathrm{ks}_1\|\mathrm{ks}_0) & (\mathrm{rc}_2\|\mathrm{rc}_1\|\mathrm{rc}_0) & 0 & 0 \end{bmatrix}$$

The values of $(\mathrm{rc}_5, \mathrm{rc}_4, \mathrm{rc}_3, \mathrm{rc}_2, \mathrm{rc}_1, \mathrm{rc}_0)$ for rounds $r = 1, \ldots, 48$ are shown below:

Rounds	Constants
1–24	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E,1D,3A,35,2B,16,2C,18,30
25–48	21,02,05,0B,17,2E,1C,38,31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

Every 4 rounds are then grouped together to form a Step, and the key material is added in every step. In this proposal, we make use of LED-80, which follows LED-128. The 80-bit key is padded with '0's and then split into two 64-bit subkeys K_1 and K_2 (note K_1 and K_2 can be encoded in the same way as for plaintext), which are then added into the state alternatively in every one of the 12 steps, as shown in Fig. 9.

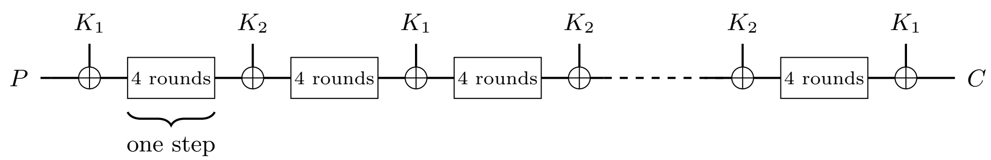

Fig. 9. Encryption of LED-80