CLOC: Compact Low-Overhead CFB ⋆

Name: CLOC v2

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

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

Date: August 29, 2015

⋆ This document was prepared based on [15,16] and [30].

1 Specification

CLOC (which stands for Compact Low-Overhead CFB, and is pronounced as "clock") is a blockcipher mode of operation for authenticated encryption with associated data (AEAD), which is also called an authenticated cipher. The design of CLOC aims at being provably secure and optimizing the implementation overhead beyond the blockcipher, the precomputation complexity, and the memory requirement. CLOC handles short input data efficiently, and is suitable for use with embedded processors.

CLOC was presented in [15], and the main difference of our CAESAR submission from [15] is that the minimum data unit is defined to be a byte (8 bits) string, and we instantiate CLOC based on AES blockcipher for 16-byte block length and TWINE blockcipher [30] for 8-byte block length.

1.1 Notation

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 $(X[1], \ldots, X[x]) \xleftarrow{\ell} X$, which is defined as follows. If $X = \varepsilon$, then $x = 1$ and $X[1] \xleftarrow{\ell} X$, 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 CLOC as byte-strings. Also we assume the big-endian format for all variables.

1.2 Algorithm and Parameters

CLOC 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. CLOC does not have the secret message number, i.e. it is always assumed to be of length zero. We require $1 \leq \ell_N \leq n - 9$ and $1 \leq \tau \leq n$, and assume that $\ell_N/8$ and $\tau/8$ are integers[⋆], and $n \in \{64, 128\}$. We write $\mathrm{CLOC}[E, \ell_N, \tau]$ for CLOC 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. $\mathrm{CLOC}[E, \ell_N, \tau] = (\text{CLOC-}\mathcal{E}, \text{CLOC-}\mathcal{D})$ consists of the encryption algorithm CLOC-$\mathcal{E}$ and the decryption algorithm CLOC-$\mathcal{D}$.

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

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

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

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

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

[⋆] In CLOC v1, the requirement was $1 \leq \ell_N \leq n - 1$, and this was updated to handle `param` in CLOC v2.

Algorithm CLOC-$\mathcal{E}_K(N, A, M)$	**Algorithm** CLOC-$\mathcal{D}_K(N, A, C, T)$
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 CLOC

Algorithm $\mathsf{HASH}_K(N, A)$	**Algorithm** $\mathsf{PRF}_K(V, C)$
1. $(A[1], \ldots, A[a]) \xleftarrow{n} A$	1. **if** $\lvert C\rvert = 0$ **then**
2. $S_{\mathsf{H}}[1] \leftarrow E_K(\mathsf{fix0}(\mathsf{ozp}(A[1])))$	2. $\quad T \leftarrow \mathsf{msb}_\tau(E_K(\mathsf{g}_1(V)))$
3. **if** $\mathsf{msb}_1(\mathsf{ozp}(A[1])) = 1$ **then**	3. $\quad$ **return** T
4. $\quad S_{\mathsf{H}}[1] \leftarrow \mathsf{h}(S_{\mathsf{H}}[1])$	4. $(C[1], \ldots, C[m]) \xleftarrow{n} C$
5. **if** $a \geq 2$ **then**	5. $S_{\mathsf{P}}[0] \leftarrow E_K(\mathsf{g}_2(V))$
6. $\quad$ **for** $i \leftarrow 2$ **to** $a - 1$ **do**	6. **for** $i \leftarrow 1$ **to** $m - 1$ **do**
7. $\quad\quad S_{\mathsf{H}}[i] \leftarrow E_K(S_{\mathsf{H}}[i-1] \oplus A[i])$	7. $\quad S_{\mathsf{P}}[i] \leftarrow E_K(S_{\mathsf{P}}[i-1] \oplus C[i])$
8. $\quad S_{\mathsf{H}}[a] \leftarrow E_K(S_{\mathsf{H}}[a-1] \oplus \mathsf{ozp}(A[a]))$	8. **if** $\lvert C[m]\rvert = n$ **then**
9. **if** $\lvert A[a]\rvert = n$ **then**	9. $\quad S_{\mathsf{P}}[m] \leftarrow E_K(\mathsf{f}_1(S_{\mathsf{P}}[m-1] \oplus C[m]))$
10. $\quad V \leftarrow \mathsf{f}_1(S_{\mathsf{H}}[a] \oplus \mathsf{ozp}(\mathtt{param} \,\|\, N))$	10. **else** $\quad$ // $1 \leq \lvert C[m]\rvert \leq n - 1$
11. **else** $\quad$ // $0 \leq \lvert A[a]\rvert \leq n - 1$	11. $\quad S_{\mathsf{P}}[m] \leftarrow E_K(\mathsf{f}_2(S_{\mathsf{P}}[m-1] \oplus \mathsf{ozp}(C[m])))$
12. $\quad V \leftarrow \mathsf{f}_2(S_{\mathsf{H}}[a] \oplus \mathsf{ozp}(\mathtt{param} \,\|\, N))$	12. $T \leftarrow \mathsf{msb}_\tau(S_{\mathsf{P}}[m])$
13. **return** V	13. **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 CLOC

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. In the figures, i is the identity function, and $\mathsf{i}(X) = X$ for all $X \in \{0,1\}^n$. In HASH, the nonce N is padded with $\mathtt{param} \in \mathbb{B}$ which is an 8-bit constant that depends on the parameters, E, ℓ_N, and τ. See Sect. 1.3 and Sect. 1.4 for the concrete values of $\mathtt{param}$. In the subroutines, we use the one-zero padding function $\mathsf{ozp} : \mathbb{B}^* \to \mathbb{B}^*$, the bit-fixing functions $\mathsf{fix0}, \mathsf{fix1} : \mathbb{B}^* \to \mathbb{B}^*$, and five tweak functions f_1, f_2, g_1, g_2, and h, which are functions over $\{0,1\}^n$.

The one-zero padding function ozp is used to adjust the length of an input string so that the total length becomes a positive multiple of n bits. For $X \in \mathbb{B}^*$, $\mathsf{ozp}(X)$ is defined as $\mathsf{ozp}(X) = X$ if $\lvert X\rvert = \ell n$ for some $\ell \geq 1$, and $\mathsf{ozp}(X) = X \,\|\, 10^{n-1-(\lvert X\rvert \bmod n)}$ otherwise. We note that $\mathsf{ozp}(\varepsilon) = 10^{n-1}$, and we also note that, in general, the function is not invertible.

The bit-fixing functions $\mathsf{fix0}$ and $\mathsf{fix1}$ are used to fix the most significant bit of an input string to zero and one, respectively. For $X \in \mathbb{B}^*$, $\mathsf{fix0}(X)$ is defined as $\mathsf{fix0}(X) = X \wedge 01^{\lvert X\rvert - 1}$, and $\mathsf{fix1}(X)$ is defined as $\mathsf{fix1}(X) = X \vee 10^{\lvert X\rvert - 1}$, where $\wedge$ and $\vee$ are the bit-wise AND operation, and the bit-wise OR operation, respectively.

The tweak function h is used in HASH if the most significant bit of $\mathsf{ozp}(A[1])$ is one. We use f_1 and f_2 in HASH and PRF, where f_1 is used if the last input block is full (i.e., if $\lvert A[a]\rvert = n$ or $\lvert C[m]\rvert = n$) and f_2 is

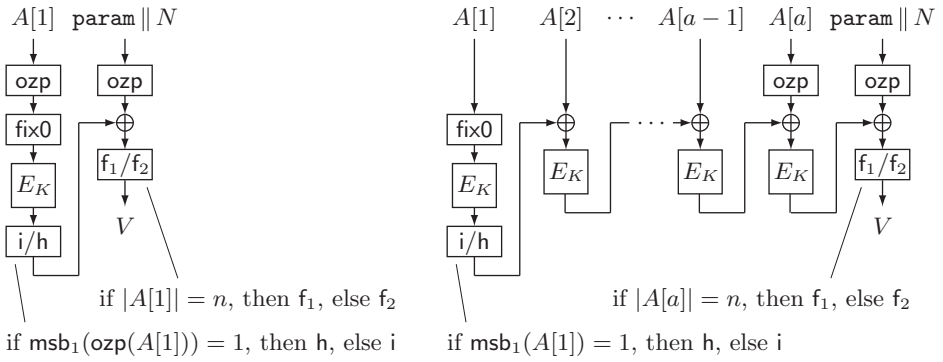

Fig. 3. $V \leftarrow \mathsf{HASH}_K(N, A)$ for $0 \le |A| \le n$ (left) and $|A| \ge n + 1$ (right)

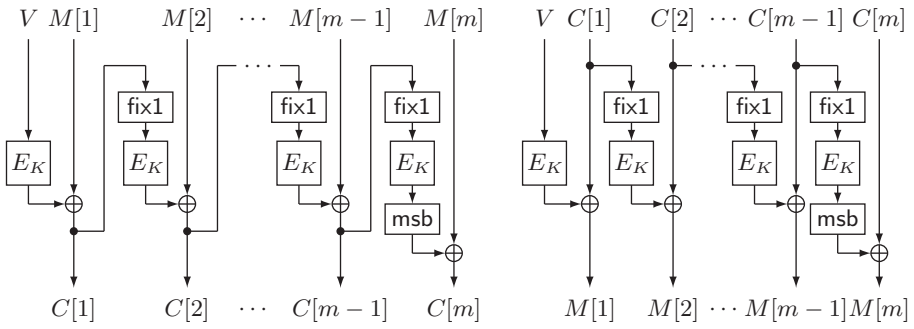

Fig. 4. $C \leftarrow \mathsf{ENC}_K(V, M)$ for $|M| \ge 1$ (left), and $\mathsf{DEC}_K(V, C)$ for $|C| \ge 1$ (right)

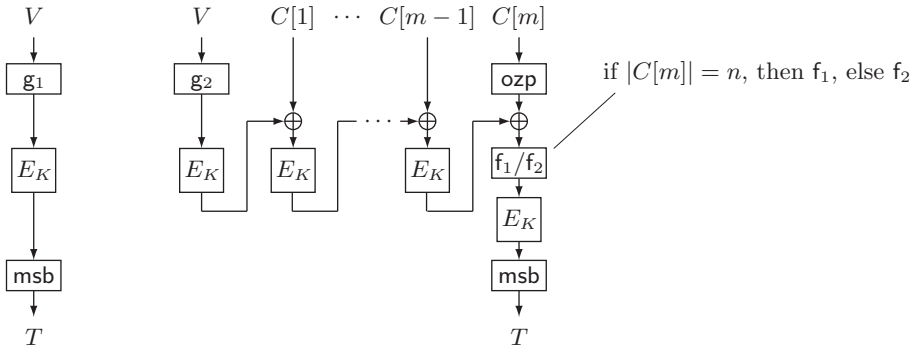

Fig. 5. $T \leftarrow \mathsf{PRF}_K(V, C)$ for $|C| = 0$ (left) and $|C| \ge 1$ (right)

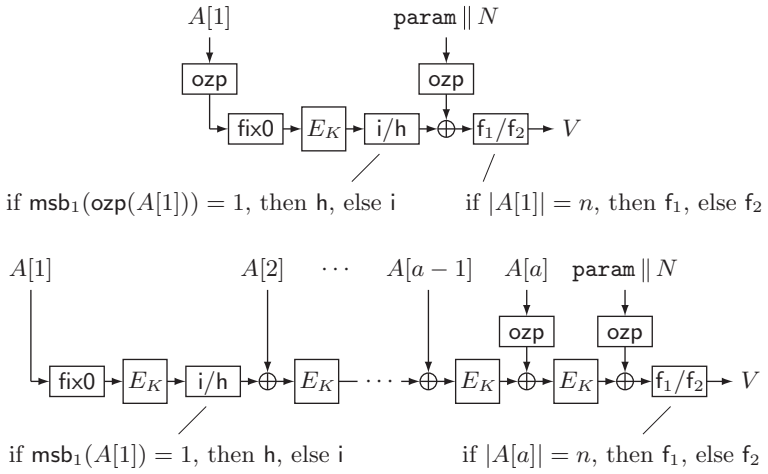

Fig. 6. $V \leftarrow \mathsf{HASH}_K(N, A)$ for $0 \le |A| \le n$ (top) and $|A| \ge n + 1$ (bottom)

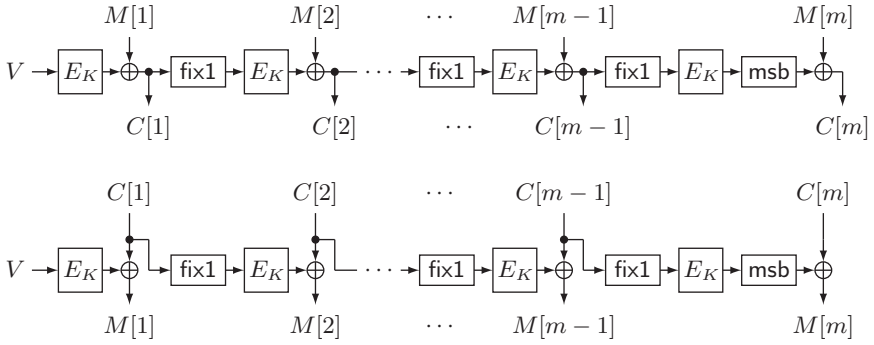

Fig. 7. $C \leftarrow \mathsf{ENC}_K(V, M)$ for $|M| \ge 1$ (top), and $\mathsf{DEC}_K(V, C)$ for $|C| \ge 1$ (bottom)

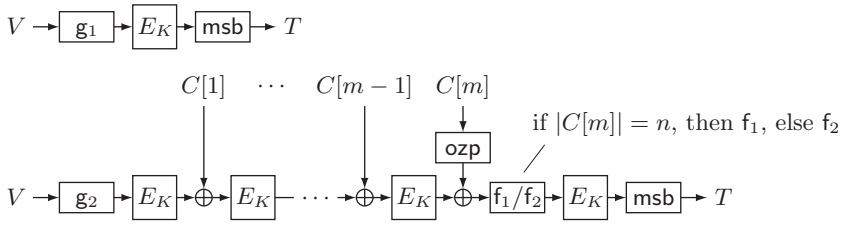

Fig. 8. $T \leftarrow \mathsf{PRF}_K(V, C)$ for $|C| = 0$ (top) and $|C| \ge 1$ (bottom)

Table 1. Definition of `param`. ℓ_N and τ are written in bytes, and `param` is in hex. The asterisk indicates the recommended parameter.

	E	ℓ_N	τ	`param`
*	AES-128	12	8	`0xc0`
	AES-128	12	12	`0xc1`
	AES-128	12	16	`0xc2`
	AES-128	12	4	`0xc3`
*	AES-128	8	8	`0xd0`
	AES-128	8	12	`0xd1`
	AES-128	8	16	`0xd2`
	AES-128	8	4	`0xd3`
	AES-128	14	8	`0xe0`
	AES-128	14	12	`0xe1`
	AES-128	14	16	`0xe2`
	AES-128	14	4	`0xe3`

	E	ℓ_N	τ	`param`
*	TWINE-80	6	4	`0xcc`
	TWINE-80	6	6	`0xcd`
	TWINE-80	6	8	`0xce`
	TWINE-80	4	4	`0xdc`
	TWINE-80	4	6	`0xdd`
	TWINE-80	4	8	`0xde`

used otherwise. We use g_1 and g_2 in PRF, where we use g_1 if the second argument of the input is the empty string (i.e., $|C| = 0$), and otherwise we use g_2. Now for $X \in \{0,1\}^n$, let $(X[1], X[2], X[3], X[4]) \xleftarrow{n/4} X$. Then f_1, f_2, g_1, g_2, and h are defined as follows.

$$\begin{cases} \mathsf{f}_1(X) = (X[1,3], X[2,4], X[1,2,3], X[2,3,4]) \\ \mathsf{f}_2(X) = (X[2], X[3], X[4], X[1,2]) \\ \mathsf{g}_1(X) = (X[3], X[4], X[1,2], X[2,3]) \\ \mathsf{g}_2(X) = (X[2], X[3], X[4], X[1,2]) \\ \mathsf{h}(X) = (X[1,2], X[2,3], X[3,4], X[1,2,4]) \end{cases}$$

Here $X[a,b]$ stands for $X[a] \oplus X[b]$ and $X[a,b,c]$ stands for $X[a] \oplus X[b] \oplus X[c]$.

Alternatively the tweak functions can be specified by a matrix. Let

$$\mathbf{M} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{1}$$

be a 4×4 binary matrix, and let $\mathbf{M}^i$ for $i \geq 0$ be exponentiations of $\mathbf{M}$, where $\mathbf{M}^0$ denotes the identity matrix. Then we have $\mathsf{f}_1(X) = X \cdot \mathbf{M}^8$, $\mathsf{f}_2(X) = X \cdot \mathbf{M}$, $\mathsf{g}_1(X) = X \cdot \mathbf{M}^2$, $\mathsf{g}_2(X) = X \cdot \mathbf{M}$, and $\mathsf{h}(X) = X \cdot \mathbf{M}^4$, where $X = (X[1], X[2], X[3], X[4])$ is interpreted as a vector.

1.3 Parameter Spaces

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

- Blockcipher E: AES-128 (AES with 128-bit key), or TWINE-80 (TWINE with 80-bit key).
- Nonce length ℓ_N: For AES-128, $\ell_N \in \{64$ bits (8 byte), 96 bits (12 bytes), 112 bits (14 bytes)$\}$, and for TWINE-80, $\ell_N \in \{32$ bits (4 byte), 48 bits (6 bytes)$\}$.
- Tag length τ: For AES-128, $\tau \in \{32$ bits (4 bytes), 64 bits (8 bytes), 96 bits (12 bytes), 128 bits (16 bytes)$\}$, and for TWINE-80, $\tau \in \{32$ bits (4 bytes), 48 bits (6 bytes), 64 bits (8 bytes)$\}$.

TWINE is a 64-bit blockcipher proposed by Suzaki, Minematsu, Morioka, and Kobayashi at SAC 2012 [30]. The specification of TWINE is described in Appendix A.

The choice of the parameter determines the value of $\mathtt{param} \in \mathbb{B}$ which is concatenated to the nonce N in HASH. The definition of `param` is given in Table 1.

Table 2. Security goal for confidentiality (privacy)

Parameter set	`aes128n12t8clocv2`	`aes128n8t8clocv2`	`twine80n6t4clocv2`
Data	64	64	32
Time	128	128	80

Table 3. Security goal for integrity (authenticity)

Parameter set	`aes128n12t8clocv2`	`aes128n8t8clocv2`	`twine80n6t4clocv2`
Data	64	64	32
Verify	64	64	32
Time	128	128	80

1.4 Recommended Parameter Sets

We specify the recommended parameter sets as follows.

- Parameter set 1, `aes128n12t8clocv2`: E = AES-128, $\ell_N = 96$ (12-byte nonce), $\tau = 64$ (8-byte tag)
- Parameter set 2, `aes128n8t8clocv2`: E = AES-128, $\ell_N = 64$ (8-byte nonce), $\tau = 64$ (8-byte tag)
- Parameter set 3, `twine80n6t4clocv2`: E = TWINE-80, $\ell_N = 48$ (6-byte nonce), $\tau = 32$ (4-byte tag)

These are marked with the asterisk in Table 1.

2 Security Goals

The security goal of CLOC 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. That is, to keep both confidentiality and integrity, the nonce of CLOC must be unique for all encryptions, and even if this condition is violated for some reason, say by a software error, CLOC retains the authenticity of sent messages, except for replays (which can be protected by some outer mechanism). Note that CLOC has no secret message number. CLOC 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). That is, for the block length of n bits, the security is guaranteed provided that the attacker obtains $\sigma \ll 2^{n/2}$ blocks of data. A detailed explanation on the attack models and the provable security bounds are given in Sect. 3.

Attack Workload. We provide security bounds of CLOC in Sect. 3 based on the pseudorandomness of the underlying blockcipher. We obtain Table 2 and Table 3 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 2, Data denotes σ_{priv} of our privacy theorem (Theorem 1), and this roughly suggests the number of data blocks that the adversary obtains. In Table 3, Data denotes σ_{auth} and Verify denotes q' of our authenticity theorem (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. We note that small constant factors are neglected in these tables. For instance the privacy bound in Theorem 1 is $5\sigma_{\mathrm{priv}}^2/2^n$, and it becomes void if $\sigma_{\mathrm{priv}} \approx (2^n/5)^{1/2}$, which is slightly less than $2^{n/2}$.

We have already mentioned that the nonce cannot be repeated to maintain the privacy. As an additional security goal, we claim that the privacy of CLOC holds as long as the uniqueness of (A, N), a pair of associated data and a nonce, is maintained. That is, even if the nonce is reused, if the uniqueness is maintained as the pair, then the privacy bound still holds. 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 blockcipher, TWINE, is not for general purpose applications. The birthday bound for the block length of 64 bits is usually unacceptable for conventional data transmission. As demonstrated by McGrew [22], it leaks information when the total data blocks reach about 32 Gbytes, if the key is not renewed. However, the parameter set with 64-bit blockcipher does not focus on commodity channels, e.g., the Internet, but it focuses on networks where the data rate is significantly low, with short packet data, and sparse data transmission from edge devices. Many protocols for wireless sensor devices have a low-data rate to suppress the power consumption. For example, IEEE 802.15.4 has 20/40/250 Kbps [4], which is used as physical and data-link layers of popular sensor protocols, Zigbee and 6LoWPAN. Another example is Z-Wave, which has 9.6 or 40 Kbps [5]. If edge devices are powered by a small battery, sending 32 Gbytes for one battery is unlikely to be possible in the first place, and rekeying should occur with battery replacement. This naturally implies that the total data blocks sent from one device for its life time is small, hence it may keep the acceptable security even with a 64-bit blockcipher. For example, when the edge device sends data of 512 bytes for every thirty minutes for 10 years (which is exceptionally long for battery-powered sensor devices), the total data amount sent from the device for its life time is about 90 Mbytes. With a standard birthday bound with the block length of 64 bits, the security bound is still below 2^{-17}, which can be acceptable for such constrained devices. Note that this setting assumes only one key, and if the device can renew the key, say, for each year, the bound can be reduced to 2^{-23}.

Of course if the target network has a high-data rate with stable power source, we recommend to use the parameter sets with a 128-bit blockcipher.

3 Security Analysis

In this section, we define the security notions of a blockcipher and CLOC, and present our security theorems. The following descriptions are taken from [15], and they hold for all recommended parameter sets.

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 [19]. We say that P is a random permutation if $P \xleftarrow{\$} \mathrm{Perm}(n)$, and define

$$\mathbf{Adv}^{\mathrm{prp}}_{E}(\mathcal{A}) \stackrel{\mathrm{def}}{=} \Pr\left[\mathcal{A}^{E_K(\cdot)} \Rightarrow 1\right] - \Pr\left[\mathcal{A}^{P(\cdot)} \Rightarrow 1\right],$$

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

Privacy Notion. We define the privacy notion for CLOC$[E, \ell_N, \tau]$ = (CLOC-$\mathcal{E}$, CLOC-$\mathcal{D}$). 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 CLOC encryption oracle, or a random-bits oracle. The encryption oracle takes $(N, A, M) \in \mathcal{N}_{\mathrm{CLOC}} \times \mathcal{A}_{\mathrm{CLOC}} \times \mathcal{M}_{\mathrm{CLOC}}$ as input and returns $(C, T) \leftarrow$ CLOC-$\mathcal{E}_K(N, A, M)$. The random-bits oracle, \$-oracle, takes $(N, A, M) \in \mathcal{N}_{\mathrm{CLOC}} \times \mathcal{A}_{\mathrm{CLOC}} \times \mathcal{M}_{\mathrm{CLOC}}$ as input and returns a random string $(C, T) \xleftarrow{\$} \{0,1\}^{|M|+\tau}$. We define the privacy advantage as

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

where the first probability is taken over $K \xleftarrow{\$} \mathcal{K}_{\mathrm{CLOC}}$ 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.

Privacy Theorem. Let $\mathcal{A}$ be an adversary that makes q queries, and suppose that the queries are $(N_1, A_1, M_1), \ldots, (N_q, A_q, M_q)$. Then we define the total associated data length as $a_1 + \cdots + a_q$, and the total plaintext length as $m_1 + \cdots + m_q$, where $(A_i[1], \ldots, A_i[a_i]) \xleftarrow{n} A_i$ and $(M_i[1], \ldots, M_i[m_i]) \xleftarrow{n} M_i$. We have the following information theoretic result.

Theorem 1. *Let* $\mathrm{Perm}(n)$*,* ℓ_N*, and* τ *be the parameters of* CLOC*. Let* $\mathcal{A}$ *be an adversary that makes at most* q *queries, where the total associated data length is at most* σ_A*, and the total plaintext length is at most* σ_M*. Then we have* $\mathbf{Adv}^{\mathrm{priv}}_{\mathrm{CLOC}[\mathrm{Perm}(n),\ell_N,\tau]}(\mathcal{A}) \le 5\sigma^2_{\mathrm{priv}}/2^n$*, where* $\sigma_{\mathrm{priv}} = q + \sigma_A + 2\sigma_M$*.*

A complete proof is presented in [15, Appendix A]. If we use a blockcipher E, which is secure in the sense of the PRP notion, instead of $\mathrm{Perm}(n)$, then the corresponding complexity theoretic result can be shown by a standard argument. See e.g. [8].

We note that, in general, the privacy of CLOC is broken if the nonce is reused. However, as long as $(N_i, A_i) \neq (N_j, A_j)$ holds for all $1 \le i < j \le q$, the bound in Theorem 1 holds. This is because, in the proof in [15, Appendix A], the transition from CLOC to CLOC5 works without the nonce-respecting assumption, and HASH5 and PRF5 used in CLOC5 generate random and independent output values if $(N_i, A_i) \neq (N_j, A_j)$ holds for all $1 \le i < j \le q$.

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 CLOC encryption oracle and the CLOC decryption oracle. The encryption oracle is defined as above. The decryption oracle takes $(N, A, C, T) \in \mathcal{N}_{\mathrm{CLOC}} \times \mathcal{A}_{\mathrm{CLOC}} \times \mathcal{C}_{\mathrm{CLOC}} \times \mathcal{T}_{\mathrm{CLOC}}$ as input and returns $M \leftarrow \text{CLOC-}\mathcal{D}_K(N, A, C, T)$ or $\perp \leftarrow \text{CLOC-}\mathcal{D}_K(N, A, C, T)$. The authenticity advantage is defined as

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

where the probability is taken over $K \stackrel{\$}{\leftarrow} \mathcal{K}_{\mathrm{CLOC}}$ 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.

Authenticity Theorem. Let $\mathcal{A}$ be an adversary that makes q encryption queries and q' decryption queries. Let $(N_1, A_1, M_1), \ldots, (N_q, A_q, M_q)$ be the encryption queries, and $(N'_1, A'_1, C'_1, T'_1), \ldots, (N'_{q'}, A'_{q'}, C'_{q'}, T'_{q'})$ be the decryption queries. Then we define the total associated data length in encryption queries as $a_1 + \cdots + a_q$, the total plaintext length as $m_1 + \cdots + m_q$, the total associated data length in decryption queries as $a'_1 + \cdots + a'_{q'}$, and the total ciphertext length as $m'_1 + \cdots + m'_{q'}$, where $(A_i[1], \ldots, A_i[a_i]) \stackrel{n}{\leftarrow} A_i$, $(M_i[1], \ldots, M_i[m_i]) \stackrel{n}{\leftarrow} M_i$, $(A'_i[1], \ldots, A'_i[a'_i]) \stackrel{n}{\leftarrow} A'_i$, and $(C'_i[1], \ldots, C'_i[m'_i]) \stackrel{n}{\leftarrow} C'_i$. We have the following information theoretic result.

Theorem 2. *Let* $\mathrm{Perm}(n)$*,* ℓ_N*, and* τ *be the parameters of* CLOC*. Let* $\mathcal{A}$ *be an adversary that makes at most* q *encryption queries and at most* q' *decryption queries, where the total associated data length in encryption queries is at most* σ_A*, the total plaintext length is at most* σ_M*, the total associated data length in decryption queries is at most* $\sigma_{A'}$*, and the total ciphertext length is at most* $\sigma_{C'}$*. Then we have* $\mathbf{Adv}^{\mathrm{auth}}_{\mathrm{CLOC}[\mathrm{Perm}(n),\ell_N,\tau]}(\mathcal{A}) \le 5\sigma^2_{\mathrm{auth}}/2^n + q'/2^\tau$*, where* $\sigma_{\mathrm{auth}} = q + \sigma_A + 2\sigma_M + q' + \sigma_{A'} + \sigma_{C'}$*.*

A complete proof is presented in [15, Appendix A]. As in the privacy case, if we use a blockcipher E secure in the sense of the PRP notion, then we obtain the corresponding complexity theoretic result by a standard argument in, e.g., [8].

4 Features

CLOC has the following features.

1. It uses only the encryption of the blockcipher both for encryption and decryption, and does not use bit-wise operations, such as a constant multiplication over $\mathrm{GF}(2^n)$.

2. It makes $\lceil |N|/n \rceil + \lceil |A|/n \rceil + 2\lceil |M|/n \rceil$ 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. We note that in CLOC, $1 \leq |N| \leq n-1$ holds (hence we always have $\lceil |N|/n \rceil = 1$), and when $|A| = 0$, it needs $\lceil |N|/n \rceil + 1 + 2\lceil |M|/n \rceil$ blockcipher calls.
3. It works with two state blocks (i.e. $2n$ bits).
4. Both encryption and decryption can be processed in an online manner.
5. Static associated data can be processed efficiently if the corresponding intermediate state value is stored.
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 second feature implies that a number of blockcipher calls required for processing short input data, say 16 or 32 bytes, is small. In particular, CLOC works without any precomputation of the blockcipher, say, computation of $E_K(0^n)$. The precomputation of CLOC is essentially the blockcipher key schedule, hence it can efficiently handle short input data even without precomputation. This feature is particularly desirable for low-power sensor networks, where messages are typically quite short and the devices have limited computational power. CLOC is designed to be used in embedded processors, but this feature may also be useful for powerful processors, for example when the key is frequently changed or when a large number of keys need to be processed. For example, when the input data consists of 1-block nonce, 1-block associated data, and 1-block plaintext, CLOC needs 4 blockcipher calls, while we need 5 or 6 calls in CCM [12], 7 calls (where 3 out of 7 can be precomputed) in EAX [9], and 5 calls (where 1 out of 5 can be precomputed) in EAX-prime [7], where the last one is insecure [24].

The first and the third features imply that CLOC works with small memory and its efficiency for processors with small words (say 8 or 16 bits). With these features CLOC is particularly suitable for embedded processors with severe ROM/RAM constraints. The last feature implies that CLOC 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, unlike many previous nonce-based AEADs.

Advantages over AES-GCM. Compared with AES-GCM [23], CLOC works efficiently for short input data on embedded processors, and the implementation of CLOC with AES can be smaller, as we do not use the full GF multiplier. In particular, AES-GCM is generally inefficient on embedded processors, since the GF multiplier is not fast (e.g. see [14]), while CLOC with AES can be efficiently implemented. For CLOC with TWINE, we expect even smaller implementation, at the cost of reduced security, which will be useful for ultimately tiny processors.

With respect to the security, the provable security bound of CLOC for authentication is better, since the bound of GCM has a term $q'(\ell_A + 1)/2^\tau$, which grows linearly with the block length ℓ_A of the associated data [17], while the corresponding term in CLOC is $q'/2^\tau$. This may have impact when τ is small. Furthermore, in GCM, the existence of weak keys was pointed out [26], while weak keys are not known in CLOC. Also, CLOC 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 TWINE for its suitability for embedded processors (comparable speed to AES, smaller code size), and good performance even for high-end platforms with SIMD operations [30]. Notably, TWINE allows efficient processing of two blocks in parallel for a wide range of platforms, which is desirable for CLOC, since in CLOC, the encryption process and tag generation can be done in parallel.

For `aes128n12t8clocv2`, 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 `aes128n8t8clocv2`, we select $\ell_N = 64$ considering the data overhead, and this is suitable for applications where the nonce consists of a counter. For `twine80n6t4clocv2`, we select $\ell_N = 48$ by taking the half of 96 in `aes128n12t8clocv2`. For all cases, the tag length was chosen by taking the balance between the security and the data overhead.

Limitations. We also list several limitations of CLOC. For long input data, CLOC 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 CLOC, HASH, ENC, DEC, and HASH, are all sequential. However, the blockcipher calls in ENC and PRF can be done in parallel. We also note that the parallelization is always possible for multiple messages [11,10]. Due to the existence of five tweak functions, the hardware implementation of CLOC does not show the smallest size compared to existing schemes, since the implementation of tweak functions requires many selectors.

5 Design Rationale

The designers have not hidden any weaknesses in this cipher. The design rationale of CLOC is detailed in [15], and we repeat the rationale below.

Our goal is to provide an AEAD particularly efficient for processing short input data, while minimizing the memory consumption and precomputation outside the blockcipher. We mainly focus on constrained sensor networks, where each data packet is short. For example, Zigbee [6] limits the maximum packet length to 127 bytes, Bluetooth low energy limits to 47 bytes [1], and many previous proposals on sensor network security protocols, e.g., TinySec [18], defined similar limits, around 30 to 128 bytes. Another example is EPC tag, which is a replacement of bar-code using RFID and has typically 96 bits [2]. We here describe the design rationale of CLOC for achieving our goal.

At abstract level CLOC is a straightforward 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. However, when we want to achieve low-overhead computation and small memory consumption, we found that any other combination of a basic encryption mode and a MAC mode did not work. For instance, we could not use CTR or OFB, as they require one state block in processing a plaintext to hold a counter value or a blockcipher output. We then realized that combining CFB and CBC MAC was not an easy task. Since we avoid using two keys or using blockcipher pre-calls, such as $L = E_K(0^n)$ used in EAX, we could not computationally separate CFB and CBC MAC via input masking, such as Galois-field doubling (2^iL for the i-th block, where $2L$ denotes the multiplication of 2 and L in GF(2^n)) [9,28]. This implies that CFB leaks input and output pairs of the blockcipher calls, which can be freely used to guess or fake the internal chaining value of CBC MAC, leading to a break of the scheme. Lucks [20] proposed an AEAD scheme based on CFB, called CCFB. However, the problem is not relevant to CCFB due to the difference in the global structure. To overcome this obstacle in composition, we introduced the bit-fixing functions. Their role is to absolutely separate the input blocks of CFB and *the first input block* of CBC MAC. This imposes the most significant one bit of the input of CBC MAC being fixed to 0, implying one-bit input loss. The set of five tweak functions is used to compensate for this information loss. It also works to compensate the information loss caused by padding functions applied to the last input block to CBC MAC. A similar technique can be found in literature [25,31], however, the previous works only considered MACs and the tweak functions required bit operations.

In the following we explain the specific requirements for the tweak functions.

Definition of f_1, f_2, g_1, g_2, *and* h. These functions are defined to meet the following properties. First, they have the additive property. That is, for any $\mathsf{z} \in \{\mathsf{f}_1, \mathsf{f}_2, \mathsf{g}_1, \mathsf{g}_2, \mathsf{h}\}$, we have $\mathsf{z}(X \oplus X') = \mathsf{z}(X) \oplus \mathsf{z}(X')$ for all $X, X' \in \{0,1\}^n$. Next, these functions are invertible over $\{0,1\}^n$. For any $\mathsf{z} \in \{\mathsf{f}_1, \mathsf{f}_2, \mathsf{g}_1, \mathsf{g}_2, \mathsf{h}\}$, we have $\mathsf{z} \in \mathrm{Perm}(n)$. Finally, they satisfy the differential probability constraints specified in Fig. 9. Let z be a function in Fig. 9. Then we require that, for any $Y \in \{0,1\}^n$, $\Pr[\mathsf{z}(K) = Y] = 1/2^n$, where the probability is taken over $K \xleftarrow{\$} \{0,1\}^n$. When z is of the form $\mathsf{z} = \mathsf{z}' \oplus \mathsf{z}''$, then $\mathsf{z}(K)$ stands for $\mathsf{z}'(K) \oplus \mathsf{z}''(K)$. When z is of the form $\mathsf{z} = \mathsf{z}'\mathsf{z}''$, then $\mathsf{z}(K)$ stands for $\mathsf{z}'(\mathsf{z}''(K))$. Recall that we define i as $\mathsf{i}(K) = K$.

Choosing Tweak Functions. Finding simple and word-wise tweak functions fulfilling all properties is not a trivial task. We start with matrix $\mathbf{M}$ of (1), which is invertible and has order 15 (i.e. $\mathbf{M}^{15} = \mathbf{M}^0$), and test all combinations of the form $(\mathsf{f}_1, \mathsf{f}_2, \mathsf{g}_1, \mathsf{g}_2, \mathsf{h}) = (i_1, \ldots, i_5) \in \{1, \ldots, 14\}^5$, where $i_1 = 2$ means $\mathsf{f}_1(X) = X \cdot \mathbf{M}^2$, using a computer. There are 864 candidates out of 537,824 fulfilling the differential probability constraints of Fig. 9. The complexity increases as the index of $\mathbf{M}$ grows, when we implement the tweak function by iterating $\mathbf{M}$, which seems suitable for hardware. For software we would directly implement $\mathbf{M}^i$ using a word-wise permutation and xor, and in this case we observe slight irregular, but similar phenomena (e.g. $\mathbf{M}^1$ needs one xor while $\mathbf{M}^3$ needs three xor's). Fig. 10 shows $\mathbf{M}^i$ and the Feistel-like implementations using a word-wise permutation and xor. It shows that, except for $\mathbf{M}^5$ and $\mathbf{M}^{10}$, we

$\mathsf{i} \oplus \mathsf{f}_1$	$\mathsf{i} \oplus \mathsf{f}_2\mathsf{h}$	$\mathsf{f}_1 \oplus \mathsf{f}_2\mathsf{h}$	$\mathsf{h} \oplus \mathsf{g}_2\mathsf{f}_1$	$\mathsf{g}_2\mathsf{f}_1 \oplus \mathsf{g}_1\mathsf{f}_2\mathsf{h}$
$\mathsf{i} \oplus \mathsf{g}_1\mathsf{f}_1$	$\mathsf{i} \oplus \mathsf{h}$	$\mathsf{f}_2 \oplus \mathsf{g}_1\mathsf{f}_1$	$\mathsf{h} \oplus \mathsf{f}_2$	$\mathsf{g}_2\mathsf{f}_1 \oplus \mathsf{f}_2\mathsf{h}$
$\mathsf{i} \oplus \mathsf{g}_1\mathsf{f}_1\mathsf{h}$	$\mathsf{i} \oplus \mathsf{g}_1$	$\mathsf{f}_2 \oplus \mathsf{g}_1\mathsf{f}_1\mathsf{h}$	$\mathsf{h} \oplus \mathsf{g}_1\mathsf{f}_2$	$\mathsf{g}_1\mathsf{f}_2 \oplus \mathsf{g}_2\mathsf{f}_1$
$\mathsf{i} \oplus \mathsf{g}_2\mathsf{f}_1$	$\mathsf{i} \oplus \mathsf{g}_2$	$\mathsf{f}_2 \oplus \mathsf{g}_2\mathsf{f}_1$	$\mathsf{h} \oplus \mathsf{g}_2\mathsf{f}_2$	$\mathsf{g}_1\mathsf{f}_2 \oplus \mathsf{g}_2\mathsf{f}_1\mathsf{h}$
$\mathsf{i} \oplus \mathsf{g}_2\mathsf{f}_1\mathsf{h}$	$\mathsf{f}_1 \oplus \mathsf{g}_1\mathsf{f}_1\mathsf{h}$	$\mathsf{f}_2 \oplus \mathsf{g}_2\mathsf{f}_1\mathsf{h}$	$\mathsf{g}_1\mathsf{f}_1 \oplus \mathsf{f}_1\mathsf{h}$	$\mathsf{g}_1\mathsf{f}_2 \oplus \mathsf{f}_1\mathsf{h}$
$\mathsf{i} \oplus \mathsf{f}_1\mathsf{h}$	$\mathsf{f}_1 \oplus \mathsf{g}_2\mathsf{f}_1\mathsf{h}$	$\mathsf{f}_2 \oplus \mathsf{f}_1\mathsf{h}$	$\mathsf{g}_1\mathsf{f}_1 \oplus \mathsf{g}_2\mathsf{f}_1\mathsf{h}$	$\mathsf{g}_1\mathsf{f}_2 \oplus \mathsf{g}_2\mathsf{f}_2\mathsf{h}$
$\mathsf{i} \oplus \mathsf{f}_2$	$\mathsf{f}_1 \oplus \mathsf{f}_2$	$\mathsf{f}_2 \oplus \mathsf{g}_1\mathsf{f}_2\mathsf{h}$	$\mathsf{g}_1\mathsf{f}_1 \oplus \mathsf{g}_2\mathsf{f}_2$	$\mathsf{g}_1\mathsf{f}_2 \oplus \mathsf{f}_2\mathsf{h}$
$\mathsf{i} \oplus \mathsf{g}_1\mathsf{f}_2$	$\mathsf{f}_1 \oplus \mathsf{g}_1\mathsf{f}_2$	$\mathsf{f}_2 \oplus \mathsf{g}_2\mathsf{f}_2\mathsf{h}$	$\mathsf{g}_1\mathsf{f}_1 \oplus \mathsf{g}_2\mathsf{f}_2\mathsf{h}$	$\mathsf{g}_2\mathsf{f}_2 \oplus \mathsf{g}_1\mathsf{f}_1\mathsf{h}$
$\mathsf{i} \oplus \mathsf{g}_1\mathsf{f}_2\mathsf{h}$	$\mathsf{f}_1 \oplus \mathsf{g}_1\mathsf{f}_2\mathsf{h}$	$\mathsf{g}_1 \oplus \mathsf{g}_2$	$\mathsf{g}_1\mathsf{f}_1 \oplus \mathsf{f}_2\mathsf{h}$	$\mathsf{g}_2\mathsf{f}_2 \oplus \mathsf{f}_1\mathsf{h}$
$\mathsf{i} \oplus \mathsf{g}_2\mathsf{f}_2$	$\mathsf{f}_1 \oplus \mathsf{g}_2\mathsf{f}_2$	$\mathsf{h} \oplus \mathsf{f}_1$	$\mathsf{g}_2\mathsf{f}_1 \oplus \mathsf{g}_1\mathsf{f}_1\mathsf{h}$	$\mathsf{g}_2\mathsf{f}_2 \oplus \mathsf{g}_1\mathsf{f}_2\mathsf{h}$
$\mathsf{i} \oplus \mathsf{g}_2\mathsf{f}_2\mathsf{h}$	$\mathsf{f}_1 \oplus \mathsf{g}_2\mathsf{f}_2\mathsf{h}$	$\mathsf{h} \oplus \mathsf{g}_1\mathsf{f}_1$	$\mathsf{g}_2\mathsf{f}_1 \oplus \mathsf{f}_1\mathsf{h}$	$\mathsf{g}_2\mathsf{f}_2 \oplus \mathsf{f}_2\mathsf{h}$

Fig. 9. Differential probability constraints of $\mathsf{f}_1, \mathsf{f}_2, \mathsf{g}_1, \mathsf{g}_2$, and h

have a simple implementation using at most four xor's. Based on these observations, we simply define the cost of computing $\mathbf{M}^i$ as i for $1 \leq i \leq 7$ and $15 - i$ for $8 \leq i \leq 14$. Then we define $f_{\mathrm{cost}}(i_1, \ldots, i_5)$ as

$$\left(i_1 \times \frac{1}{16} + i_2 \times \frac{15}{16}\right) \times 2 + i_4 + i_5 \times \frac{1}{2}.$$

This corresponds to the expected total cost for given $(i_1, \ldots, i_5)$, where associated data and a plaintext are assumed to be non-empty byte strings of random lengths, (as we expect the standard use of CLOC is AEAD, not MAC), and the most significant bit of the associated data is assumed to be random. Then there remains only two candidates giving the minimum value of f_{cost}, which are $(i_1, \ldots, i_5) = (8, 1, 2, 1, 4)$ and $(8, 1, 6, 1, 4)$. As smaller i_3 is better, we choose the former as the sole winner. We also tested other matrices, say the one replacing the forth column of $\mathbf{M}$ by the transposition of $(1, 0, 1, 0)$, but no better solution was found.

We note that $\mathbf{M}^8 = \mathbf{M}^2 \oplus \mathbf{M}^0$ and $\mathbf{M}^4 = \mathbf{M}^1 \oplus \mathbf{M}^0$ hold, implying that we have $\mathsf{f}_1(X) = \mathsf{g}_1(X) \oplus X$ and $\mathsf{h}(X) = \mathsf{f}_2(X) \oplus X = \mathsf{g}_2(X) \oplus X$, which may be useful in some implementations.

Selection of Blockciphers. For $n = 128$, we choose AES as the underlying blockcipher, because the security of AES has been extensively studied. For $n = 64$, we choose TWINE as the underlying blockcipher, because of its low-resource implementation for embedded processors shown in [30]. The use of 64-bit blockcipher is particularly useful if input length is quite short, say a few bytes, which is in fact possible for ultimately constrained, single-purpose sensors such as energy harvester devices like [3], gas or water metering, etc. In such cases, a 128-bit blockcipher can be inefficient, since it is likely that we have more redundant output bits from the blockcipher that has to be discarded.

6 Intellectual Property

We claim no intellectual property (IP) rights associated to CLOC other than its internal blockcipher, and are unaware of any relevant IPs to CLOC held by others. NEC Corporation (NEC) has pending patent applications related to its TWINE blockcipher proposal in CLOC: WO2011052585 and WO2011052587. In case that CLOC with TWINE blockcipher is included into the final portfolio, NEC is willing to provide to implementors, solely for the purpose of implementing CLOC, a royalty-free, non-exclusive license under the patents issuing on such patent applications, to the extent such patents are essential to implement CLOC as set forth in the final portfolio, provided said that implementor extends a reciprocal royalty-free license.

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

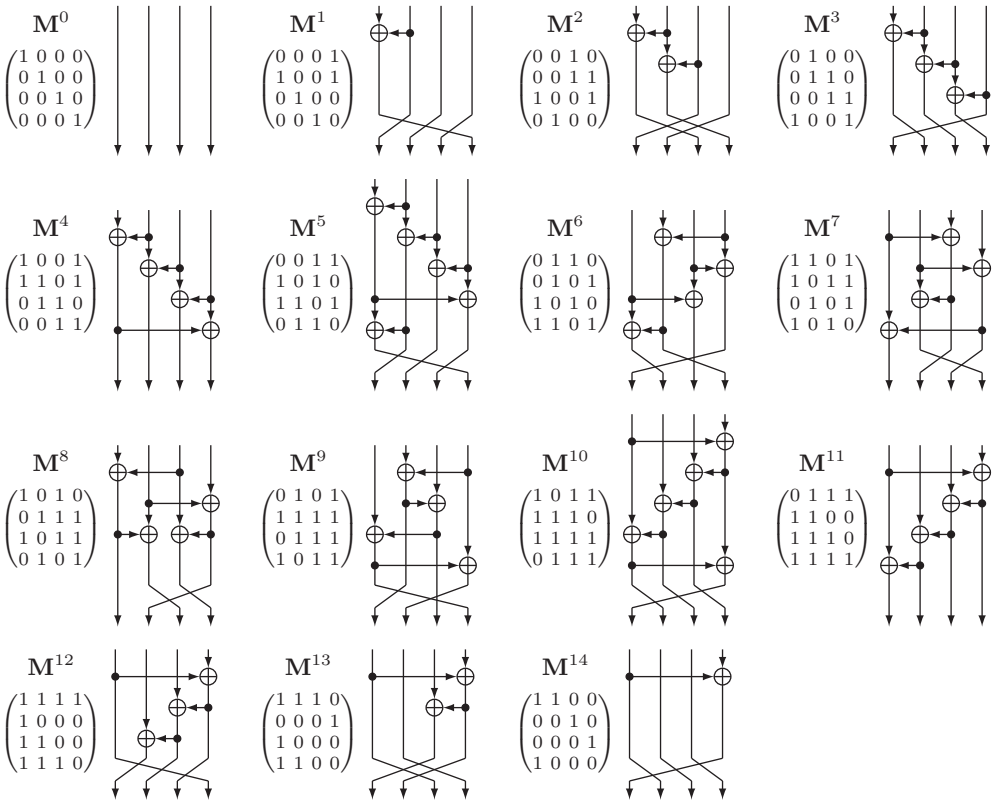

Fig. 10. Matrix exponentiations for the tweak functions

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, and was supported in part by JSPS KAKENHI, Grant-in-Aid for Scientific Research (B), Grant Number 26280045. The work by Jian Guo was partially supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Bluetooth low energy, `http://www.bluetooth.com/Pages/Low-Energy.aspx/`
2. Electronic Product Code (EPC) Tag Data Standard (TDS), `http://www.epcglobalinc.org/standards/tds/`
3. EnOcean, `http://www.enocean.com/`
4. IEEE 802.15.4, `http://en.wikipedia.org/wiki/IEEE_802.15.4/`
5. Z-Wave Alliance, `http://www.z-wavealliance.org/`
6. ZigBee Alliance, `http://www.zigbee.org/`
7. American National Standard Protocol Specification For Interfacing to Data Communication Networks. ANSI C12.22-2008. (2008)

8. Bellare, M., Kilian, J., Rogaway, P.: The Security of the Cipher Block Chaining Message Authentication Code. J. Comput. Syst. Sci. 61(3), 362–399 (2000)
9. 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)
10. Bogdanov, A., Lauridsen, M., Tischhauser, E.: AES-Based Authenticated Encryption Modes in Parallel High-Performance Software. Cryptology ePrint Archive, Report 2014 (2014)
11. Bogdanov, A., Mendel, F., Regazzoni, F., Rijmen, V., Tischhauser, E.: ALE: AES-Based Lightweight Authenticated Encryption. Pre-proceedings of Fast Software Encryption 2013 (2013)
12. Dworkin, M.: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. NIST Special Publication 800-38C (2004)
13. Eichlseder, M.: Remark on variable tag lengths and OMD. A comment on the CAESAR mailing list (2014), `https://groups.google.com/forum/#!forum/crypto-competitions`
14. Gouvêa, C.P.L., López, J.: High Speed Implementation of Authenticated Encryption for the MSP430X Microcontroller. In: Hevia, A., Neven, G. (eds.) LATINCRYPT. Lecture Notes in Computer Science, vol. 7533, pp. 288–304. Springer (2012)
15. Iwata, T., Minematsu, K., Guo, J., Morioka, S.: CLOC: Authenticated Encryption for Short Input. Pre-proceedings of FSE 2014 (2014)
16. 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/`
17. 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)
18. Karlof, C., Sastry, N., Wagner, D.: TinySec: a link layer security architecture for wireless sensor networks. In: Stankovic, J.A., Arora, A., Govindan, R. (eds.) SenSys. pp. 162–175. ACM (2004)
19. Luby, M., Rackoff, C.: How to Construct Pseudorandom Permutations from Pseudorandom Functions. SIAM J. Comput. 17(2), 373–386 (1988)
20. Lucks, S.: Two-Pass Authenticated Encryption Faster Than Generic Composition. In: Gilbert, H., Handschuh, H. (eds.) FSE. Lecture Notes in Computer Science, vol. 3557, pp. 284–298. Springer (2005)
21. Manger, J.H.: [Cfrg] Attacker changing tag length in OCB. A discussion thread on Cfrg (2013), `http://www.ietf.org/mail-archive/web/cfrg/current/msg03433.html`
22. McGrew, D.: Impossible Plaintext Cryptanalysis and Probable-Plaintext Collision Attacks of 64-bit Block Cipher Modes. Pre-proceedig of FSE 2013 (2013)
23. 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)
24. Minematsu, K., Lucks, S., Morita, H., Iwata, T.: Attacks and Security Proofs of EAX-Prime. Pre-proceedings of Fast Software Encryption 2013 (2013), full-version available at `http://eprint.iacr.org/2012/018`
25. 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)
26. Procter, G., Cid, C.: On Weak Keys and Forgery Attacks against Polynomial-Based MAC Schemes. Pre-proceedig of FSE 2013 (2013)
27. Rogaway, P., Wagner, D.: A Critique of CCM. Cryptology ePrint Archive, Report 2003/070 (2003), `http://eprint.iacr.org/`
28. 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)
29. Suzaki, T., Minematsu, K.: Improving the Generalized Feistel. In: Hong, S., Iwata, T. (eds.) FSE. Lecture Notes in Computer Science, vol. 6147, pp. 19–39. Springer (2010)
30. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE : A Lightweight Block Cipher for Multiple Platforms. In: Knudsen, L.R., Wu, H. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 7707, pp. 339–354. Springer (2012)
31. Zhang, L., Wu, W., Zhang, L., Wang, P.: CBCR: CBC MAC with rotating transformations. SCIENCE CHINA Information Sciences 54(11), 2247–2255 (2011)

A TWINE Blockcipher [30]

We describe TWINE blockcipher by reusing materials from [30].

Data Processing Part. TWINE is a 64-bit blockcipher with 80 or 128-bit keys. We write TWINE-80 or TWINE-128 to denote the key length. We here focus on TWINE-80, which is used in CLOC. The global structure of TWINE is a variant of Type-2 generalized Feistel structure (GFS) with 16 nibbles (i.e. 4-bit sub-blocks). A round function of TWINE consists of a nonlinear layer using 4-bit S-boxes and a diffusion

layer, which is a permutation on 16 nibbles. The diffusion layer of TWINE is not a cyclic shift and is chosen to provide a better diffusion property than the cyclic shift, according to the result of Suzaki and Minematsu [29]. This round function is iterated for 36 times for both key lengths, where the diffusion layer of the last round is omitted. For $i = 1, \ldots, 36$, the i-th round uses a 32-bit round key, RK^i, which is derived from the 80-bit secret key, K, using the key schedule.

The data processing part essentially consists of a 4-bit S-box, denoted by S, and a permutation π over the indexes of 4-bit nibbles. That is, we have $\pi : \{0, \ldots, 15\} \rightarrow \{0, \ldots, 15\}$, where the j-th sub-block is mapped to the $\pi[j]$-th sub-block. Fig. 11 shows the encryption procedure, TWINE.Enc, and the decryption procedure, TWINE.Dec, using the derived round keys. Fig. 11 also shows S-box S, and the permutation π and its inverse. In all figures of this section, a variable X may have a subscript (i) to express its length, i.e., X may be written as $X_{(|X|)}$, for clearness. The round function is also illustrated in Fig. 13.

Key Schedule Part. The key schedule produces $RK_{(32\times 36)}$ from the 80-bit secret key K. It is also a variant of GFS with nibbles using the same S-box as data processing part. The key schedule uses 6-bit round constants, $CON^i_{(6)} = CON^i_{H(3)} \| CON^i_{L(3)}$ for $i = 1$ to 35. Fig. 12 shows the pseudocode of the key schedule, and Fig. 14 illustrates the key schedule for one round. In Fig. 12 Rot$i(x)$ means i-bit left cyclic shift of x. We remark that CON^i corresponds to 2^i in GF(2^6) with primitive polynomial $z^6 + z + 1$. The values of CON^i are also listed at Fig. 12.

We provide a test vector in Table 4.

Algorithm TWINE.Enc$(P_{(64)}, RK_{(32\times 36)}, C_{(64)})$

1. $X^1_{0(4)} \| X^1_{1(4)} \| \ldots \| X^1_{15(4)} \leftarrow P$
2. $RK^1_{(32)} \| \ldots \| RK^{36}_{(32)} \leftarrow RK_{(32\times 36)}$
3. **for** $i = 1$ **to** 35 **do**
4. $\quad RK^i_{0(4)} \| RK^i_{1(4)} \| \ldots \| RK^i_{7(4)} \leftarrow RK^i_{(32)}$
5. $\quad$ **for** $j = 0$ **to** 7 **do**
6. $\quad\quad X^i_{2j+1} \leftarrow S(X^i_{2j} \oplus RK^i_j) \oplus X^i_{2j+1}$
7. $\quad$ **for** $h = 0$ **to** 15 **do**
8. $\quad\quad X^{i+1}_{\pi[h]} \leftarrow X^i_h$
9. **for** $j = 0$ **to** 7 **do**
10. $\quad X^{36}_{2j+1} \leftarrow S(X^{36}_{2j} \oplus RK^{36}_j) \oplus X^{36}_{2j+1}$
11. $C \leftarrow X^{36}_0 \| X^{36}_1 \| \ldots \| X^{36}_{15}$

Algorithm TWINE.Dec$(C_{(64)}, RK_{(32\times 36)}, P_{(64)})$

1. $X^{36}_{0(4)} \| X^{36}_{1(4)} \| \ldots \| X^{36}_{15(4)} \leftarrow C$
2. $RK^1_{(32)} \| \ldots \| RK^{36}_{(32)} \leftarrow RK_{(32\times 36)}$
3. **for** $i = 36$ **to** 2 **do**
4. $\quad RK^i_{0(4)} \| RK^i_{1(4)} \| \ldots \| RK^i_{7(4)} \leftarrow RK^i_{(32)}$
5. $\quad$ **for** $j = 0$ **to** 7 **do**
6. $\quad\quad X^i_{2j+1} \leftarrow S(X^i_{2j} \oplus RK^i_j) \oplus X^i_{2j+1}$
7. $\quad$ **for** $h = 0$ **to** 15 **do**
8. $\quad\quad X^{i-1}_{\pi^{-1}[h]} \leftarrow X^i_h$
9. **for** $j = 0$ **to** 7 **do**
10. $\quad X^1_{2j+1} \leftarrow S(X^1_{2j} \oplus RK^1_j) \oplus X^1_{2j+1}$
11. $P \leftarrow X^1_0 \| X^1_1 \| \ldots \| X^1_{15}$

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

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[h]$	5	0	1	4	7	12	3	8	13	6	9	2	15	10	11	14
$\pi^{-1}[h]$	1	2	11	6	3	0	9	4	7	10	13	14	5	8	15	12

Fig. 11. Data processing part of TWINE (top) with S-box S (middle) and permutation π (bottom)

Table 4. A test vector of TWINE-80 in hexadecimal notation

key (80 bits)	00112233 44556677 8899
plaintext	01234567 89ABCDEF
ciphertext	7C1F0F80 B1DF9C28

Algorithm TWINE.KeySchedule-80($K_{(80)}, RK_{(32\times 36)}$)

1. $WK_{0(4)} \| WK_{1(4)} \| \ldots \| WK_{19(4)} \leftarrow K$
2. **for** $r = 1$ **to** 35 **do**
3. $\quad RK^r_{(32)} \leftarrow WK_1 \| WK_3 \| WK_4 \| WK_6 \| WK_{13} \| WK_{14} \| WK_{15} \| WK_{16}$
4. $\quad WK_1 \leftarrow WK_1 \oplus S(WK_0)$
5. $\quad WK_4 \leftarrow WK_4 \oplus S(WK_{16})$
6. $\quad WK_7 \leftarrow WK_7 \oplus 0 \| CON^r_H$
7. $\quad WK_{19} \leftarrow WK_{19} \oplus 0 \| CON^r_L$
8. $\quad WK_0 \| \ldots \| WK_3 \leftarrow \mathrm{Rot4}(WK_0 \| \ldots \| WK_3)$
9. $\quad WK_0 \| \ldots \| WK_{19} \leftarrow \mathrm{Rot16}(WK_0 \| \ldots \| WK_{19})$
10. $RK^{36}_{(32)} \leftarrow WK_1 \| WK_3 \| WK_4 \| WK_6 \| WK_{13} \| WK_{14} \| WK_{15} \| WK_{16}$
11. $RK \leftarrow RK^1 \| RK^2 \| \ldots \| RK^{36}$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CON^i	01	02	04	08	10	20	03	06	0C	18	30	23	05	0A	14	28	13	26
i	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	
CON^i	0F	1E	3C	3B	35	29	11	22	07	0E	1C	38	33	25	09	12	24	

Fig. 12. Key schedule of TWINE-80. S-box S is the same as Fig. 11.

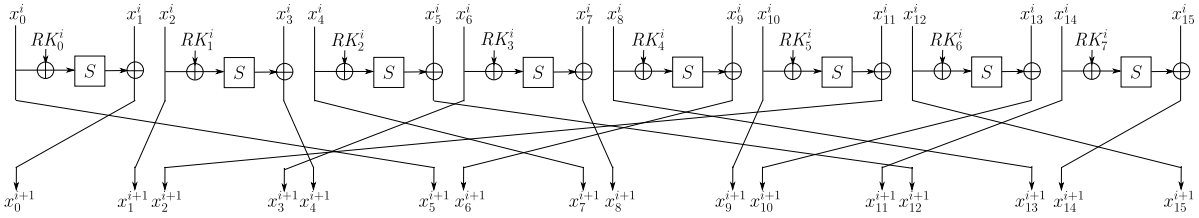

Fig. 13. Round function of TWINE

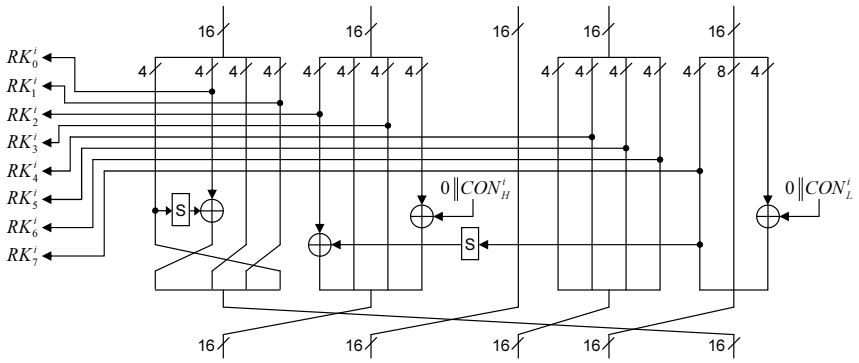

Fig. 14. 80-bit key schedule

B Changes

B.1 Changes from CLOC v1 to CLOC v2

The specification of CLOC v2 uses `param` so that the encryption and decryption algorithms depend on the choice of the parameters, which are E, ℓ_N, and τ. This type of dependency was previously highlighted, e.g., in [13,21,27]. There are three things to note:

- The introduction of `param` does not mean that CLOC v2 handles variable length nonces nor variable length tags. All the parameters, E, ℓ_N, and τ, have to be fixed during the lifetime of the secret key.
- The introduction of `param` does not affect the provable security result of CLOC, since we may consider `param` $\| N$ as a nonce, and then the provable security results in [15] still hold.

– We also note that `param` does not remove the dependency to other blockcipher modes of operation. For instance the concurrent use (with the same secret key) of CLOC and ECB mode results in the loss of security. Similarly, CLOC and SILC cannot be used concurrently.

The following part of the document was updated.

– The condition on the nonce length was updated to $1 \leq \ell_N \leq n - 9$ to handle `param`.
– Lines 10 and 12 in the definition of HASH in Fig. 2 were updated to concatenate `param` to N.
– Figures 3 and 6 were updated.
– Sect. 1.3 was updated. The parameter space was reduced so that different parameters can be encoded into `param`, and Table 1 was added.
– Sect. 6, Intellectual Property, was updated.
– We also made minor changes.