SCMAC and LOL2.0: An AEAD Design Framework and A New Version of LOL Stream Cipher Design Framework

Dengguo Feng, Lin Jiao, Yonglin Hao, Qunxiong Zheng, Wenling Wu, Wenfeng Qi, Lei Zhang, Liting Zhang, Siwei Sun, and Tian Tian

Abstract

In this paper, we introduce SCMAC, a general framework that transforms large-memory stream ciphers into AEAD schemes. It represents an intermediate design paradigm between Encrypt-then-MAC and dedicated single-pass AEAD, partially integrating encryption and authentication mechanisms while mitigating the risk of state leakage associated with immediate absorption and squeezing. Consequently, this approach harmonizes high performance with enhanced security. Additionally, we propose LOL2.0, an enhanced version of the blockwise stream cipher design framework LOL. This new framework improves security through modifications to the FSM update and output functions, and increases flexibility in constructing LFSR components. Based on SCMAC and LOL2.0, we present two AEAD ciphers, LOL2.0-Mini and LOL2.0-Double, which support both stream cipher and AEAD modes. These ciphers are tailored to Beyond 5G/6G environments, offering 256-bit key length and resistance to known cryptanalysis methods, including differential, linear, and integral attacks. They also provide 128-bit security against forgery attacks in the nonce-respecting setting. Due to their compatibility with AES-NI and SIMD instructions, LOL2.0-Mini and LOL2.0-Double achieve software performance of 90 Gbps and 144 Gbps in stream cipher mode, respectively. In AEAD mode, they perform at 59 Gbps and 110 Gbps, significantly faster than their predecessor's Encrypt-then-MAC versions.

Index Terms

Stream Cipher, AEAD, 6G Mobile System, Fast Software Implementation

I. Introduction

WITH the widespread commercial deployment of 5G telecommunication networks, both the research and industrial communities are now directing their focus toward the beyond-5G era. The 6G white paper [1] has outlined several key requirements for 6G, including data transmission speeds exceeding 100 Gbps and further adoption of software-defined virtual network, which was initially introduced in 5G. In these networks, all services and applications are implemented on general CPUs rather than dedicated hardware devices. These advancements necessitate significantly higher software performance for symmetric-key primitives. In addition to efficiency, security margins must be strengthened to counteract the rapid progress of quantum computation techniques, especially the Grover's algorithm. This requirement is emphasized by the 3GPP standardization organization, which requires the upcoming symmetric-key designs to support 256-bit keys starting from the proposal of 5G encryption algorithms [2]. Consequently, the challenge of balancing enhanced security with performance for symmetric-key primitives becomes increasingly critical.

To ensure both confidentiality and integrity for data, Authenticated Encryption with Associated Data (AEAD) schemes are essential. Traditional AEAD designs, such as encrypt-then-MAC, rely on separate mechanisms for confidentiality and integrity. Early standardized AEAD schemes emerged in the 2000s, with AES-GCM (NIST SP 800-38D, 2007) and ChaCha20-Poly1305 (RFC 7539, 2015) becoming widely adopted. The CAESAR Competition (2014–2019) promoted dedicated single-pass designs that integrate encryption and authentication into a single primitive, such as AEGIS and ASCON. While traditional schemes often provide strong formal security guarantees at the cost of moderate throughput, single-pass schemes offer superior high-performance characteristics but may introduce attack feasibility due to the immediate absorption of plaintext and the subsequent squeeze of ciphertext.

The development of the chip industry has made prominent contributions to high software efficiencies. Modern CPUs are usually equipped with advanced instruction sets following the idea of Single Instruction Multiple Data (SIMD). With a single assembly instruction, it is possible to perform multiple basic operations in parallel or execute one round of the standard AES block cipher. Such advanced instructions are not only supported by Intel top-class CPUs running on large clusters but also implemented in SoC architectures of mobile devices [3]. Many recent cryptographic designs

Dengguo Feng, Lin Jiao, Yonglin Hao are with State Key Laboratory of Cryptology, Beijing, China
Qunxiong Zheng, Wenfeng Qi, Tian Tian are with PLA Strategic Support Force Information Engineering University, Zhengzhou, 450001, China
Wenling Wu, Lei Zhang are with Institute of Software, Chinese Academy of Sciences, Beijing, China
Liting Zhang is with Westone Cryptologic Research Center, Beijing, China
Siwei Sun is with School of Cryptology, University of Chinese Academy of Sciences, Beijing, China

have utilized SIMD instructions and AES-NI as a foundational component to achieve high performance. For instance, SNOW-V/Vi [4], [5], LOL-Mini/Double [6], AEGIS [7], and Rocca [8] are specifically optimized to exploit the parallelism and efficiency offered by these instructions. Among these primitives, SNOW-V/Vi and LOL-Mini/Double are stream ciphers while AEGIS and Rocca are AEAD schemes. As a new proposal of the SNOW design team, SNOW-V/Vi inherits the structure of SNOW, which consists of an LFSR connected to an FSM. LOL-Mini/Double are designed based on a general stream cipher design framework LOL adopting a structure of nonlinear source with large memory FSM over finite fields. SNOW-V and LOL-Mini/Double defined their AEAD mode with the standard MAC, such as GMAC, following the encryption-then-MAC paradigm. AEGIS is the first primitives to use the parallel AES round functions and is one of the finalist candidates of the CAESAR competition [9]. As the first design dedicated to 6G systems, Rocca series [8], [10] can be regarded as an extension of AEGIS: with wiser organization of parallel AES round functions, Rocca enjoys the current highest software efficiency.

Motivations. Our primary motivations are based on the following two observations.

Observation 1: Stream ciphers have long been critical in protecting communication systems [11], [12], [13]. Notably, stream ciphers with large memory require fewer iterations per block on average, offering higher encryption efficiency compared to block ciphers counterparts (e.g., SNOW-V demonstrates superior software efficiency compared to AES-256). While stream ciphers can be transformed into AEAD schemes using universal hash under the encrypt-then-MAC paradigm, this approach significantly degrades their implementation performance. The AEAD schemes generated by SNOW-V and LOL-Mini/Double based on GCM achieve an implementation performance of approximately 30-40 Gbps, which is insufficient to meet the 6G requirements and significantly lower than the performance achieved by AEGIS and Rocca (approximately 130-150 Gbps). However, the security of the single-pass AEAD designs AEGIS and Rocca has been challenged by several security issues. For example, Hoeoyamada et al. showed a key-recovery attack on Rocca in the nonce-respecting setting in ToSC 2022(3)/FSE 2023 [14]. Specially, if the tag size is smaller than the key size, a state-recovery attack can be mounted by access to decryption oracle since each message block is absorbed by a weak permutation. Therefore, the first challenge we aim to address is identifying a balanced AEAD scheme between the encryption-then-MAC and single-pass paradigms, ensuring both high performance and robust security against current attacks. Especially, there should be an AEAD transformation suitable for stream ciphers with fewer efficiency losses.

Observation 2: Although SNOW-V/Vi, AEGIS and Rocca claimed to support 256-bit keys, the current cryptanalysis results indicate that they may not reach the theoretic 256-bit security. Specifically, the simple output function of AEGIS resulted in its vulnerability against linear distinguishing attacks [7], [15]. For full SNOW-V/Vi, there are fast correlation attacks (FCA) recovering the internal states within the 2^{256} secure bound [16], [17]. For Rocca, the vulnerability exploited by the attack in [14] stems from its pure reliance on AES round functions, thereby enabling an attack that leverages the properties of AES S-box. Thus, it is necessary to integrate certain non-AES components into the design in order to improve its resilience and diversity as a stream cipher with continuous output and relatively weak update functions, distinct from block ciphers. Simultaneously, the resistance against FCAs, which pose a significant threat to the aforementioned ciphers, should be substantially enhanced. Therefore, the second challenge is to design an encryption algorithm that not only meets the performance requirements for 6G implementation and but also, more importantly, can ensure verifiably sufficient security.

Contribution. To address these challenges, we propose SCMAC, a new universal design framework for transforming stream ciphers into AEAD schemes. We observe that the thoroughly diffused large memory in stream ciphers enables efficient configuration of large permutations for message absorption in AEAD. Furthermore, the output functions in stream ciphers can generate ciphertexts along with authentication tags. To fully exploit the inherent advantages of large-memory stream ciphers, we propose a method that bridges the gap between encrypt-then-MAC and dedicated single-pass designs. SCMAC scheme operates in a partial two-step mode: the internal states for generating ciphertexts and tags are updated independently, except for a "copy-and-feedback" interaction, which shares the initialization phase of the stream cipher for thorough diffusion. Both ciphertexts and tags in SCMAC are derived from the keystream bits of the underlying stream cipher. Compared to single-pass AEAD schemes, where encryption and tag generation typically rely on shared internal states, potentially leading to tight correlations between ciphertext and tag bits and introducing overlooked security vulnerabilities, SCMAC avoids such issues while maintaining high efficiency. Additionally, unlike the encrypt-then-MAC approach, which requires implementing two distinct primitives with differing operational principles, SCMAC achieves a more unified and efficient implementation. Furthermore, the security analysis of SCMAC can directly leverage existing cryptanalysis work on the underlying stream cipher, significantly reducing the associated workload. Consequently, the SCMAC design framework strikes an effective balance between high performance and enhanced security.

We propose an improved stream cipher design framework named LOL2.0. Compared to its predecessor LOL, LOL2.0 strengthens the data interaction between NFSRs and FSMs, thereby providing greater resistance against FCAs. Similar to LOL, LOL2.0 generates stream cipher family by incorporating the component library into extension modes. Additionally, LOL2.0 introduces several new approaches for constructing large-width LFSRs within the component library, which targets to achieve maximum-period LFSRs with fewer instructions. This expansion renders the component selection of LOL2.0 more flexible compared to LOL. Corresponding to the basic single mode and the extended parallel-dual mode of the LOL2.0 framework, we introduce two concrete LOL2.0 stream ciphers, namely LOL2.0-Mini and LOL2.0-Double. Both designs support a 256-bit key length and achieve software encryption speeds of 102 Gbps for LOL2.0-Mini and 142 Gbps for LOL2.0-Double, aligning with the performance requirements of 6G systems. Furthermore, by the SCMAC design framework, we transform the LOL2.0-Mini and LOL2.0-Double stream ciphers into AEAD schemes. These schemes further provide 128-bit security against forgery attacks, with software efficiencies of 59 Gbps and 108 Gbps, significantly surpassing their universal hash-based counterparts. Notably, LOL2.0-Double in the AEAD mode fulfills the performance requirement of 6G systems.

Outline. Notations and preliminaries are introduced in Section II. The LOL2.0 framework is described in Section IV, while the SCMAC framework is outlined in Section III. Instances of the LOL2.0 cipher utilizing SCMAC, along with their security analysis, are detailed in Section V and Section VI. The paper concludes in Section VIII.

II. Notations and Preliminaries

In this section, we declare the notations used hereafter.

- $\boldsymbol{S}$: internal state of the stream cipher (*ScState*).
- $\boldsymbol{E}$: additional state for MAC (*MacState*).
- K: secret key.
- IV: public initialization vector of the stream cipher (also served as the nonce of the AEAD scheme).
- $|X|$: length of X in bits.
- $\|$: concatenation operation.
- b: length of one keystream block in bits.
- τ: length of the authentication tag in bits.
- $\overline{X}$: padded X as $\overline{X} = X\|0^{\ell}$, where ℓ is the minimal non-negative integer making $|\overline{X}|$ a multiple of b.

General subroutines of stream ciphers are denoted as follows. The encryption/decryption process is summarized as scEndec, defined in Algorithm 1.

- $\boldsymbol{S} \leftarrow \texttt{Load}(K, IV)$: initialize the *ScState* $\boldsymbol{S}$ by loading the secret key and IV to appropriate positions.
- $\boldsymbol{S} \leftarrow \texttt{scInit}(\boldsymbol{S})$: run initialization rounds to prepare $\boldsymbol{S}$ for keystream generation.
- $(Z, \boldsymbol{S}) \leftarrow \texttt{scOut}(\boldsymbol{S})$: generate a b-bit keystream block Z and update $\boldsymbol{S}$.

The following operations are used in the new AEAD design framework SCMAC for authentications:

- $\boldsymbol{E} \leftarrow \texttt{COPY}(\boldsymbol{S})$: copy selected bits from $\boldsymbol{S}$ to create a new state $\boldsymbol{E}$.
- $\boldsymbol{E} \leftarrow \texttt{updE}(\boldsymbol{E}, D)$: update *MacState* $\boldsymbol{E}$ by absorbing a b-bit block D.
- $\boldsymbol{S} \leftarrow \texttt{Feedback}(\boldsymbol{S}, \boldsymbol{E})$: update partial $\boldsymbol{S}$ using all bits from $\boldsymbol{E}$ (via XOR or replacement).
- $T \leftarrow \texttt{trunc}(Z, \ell)$: truncate Z to ℓ bits ($|Z| \geq \ell$).

Algorithm 1: Encryption/decryption process of stream ciphers. X denotes plaintext or ciphertext.

```
procedure scEndec($K, IV, X$)
    Load $(K, IV)$: $\boldsymbol{S} \leftarrow \texttt{Load}(K, IV)$
    Run initialization rounds: $\boldsymbol{S} \leftarrow \texttt{scInit}(\boldsymbol{S})$
    if $|X| > 0$ then
        Set $m \leftarrow \lceil |X|/b \rceil$ and $n \leftarrow |X| \mod b$
        Pad $X$ to $\overline{X} = \overline{X}_0 \| \dots \| \overline{X}_{m-1}$
        for $i = 0, \dots m-1$ do
            $(Z_i, \boldsymbol{S}) \leftarrow \texttt{scOut}(\boldsymbol{S})$
            $Y_i \leftarrow \overline{X}_i \oplus Z_i$
        if $n > 0$ then
            $Y_{m-1} \leftarrow \texttt{trunc}(Y_{m-1}, n)$
        return $Y = Y_0 \| \dots \| Y_{m-1}$
    else
        return $\perp$
```

To describe LOL2.0 framework, we present the following notations:

- (H, L): state of LFSR.
- N: state of NFSR.
- S: state of FSM.
- Z: keystream block.

- f: LFSR state updating function.
- $\mathcal{R}$: SPN function used in NFSR/FSM.
- h: keystream generation function.
- F: feedback of LFSR.
- G: output of FSM.

Basic operations are defined as follows:

- $w \ll n$ ($w \gg n$): shift word w left (right) by n bits, padding lower (higher) n bits with 0.
- $w \lll n$ ($w \ggg n$): rotate word w left (right) by n bits.
- $X|_m^{\ll n}$ ($X|_m^{\gg n}$): divide state block X into m-bit words and apply $\ll n$ ($\gg n$) to each. For example, $X|_m^{\ll n} = (x_{d-1} \ll n, \ldots, x_0 \ll n)$ where $d = |X|/m$ and $x_0, \ldots, x_{d-1}$ are m-bit words forming $X = (x_{d-1}, \ldots, x_0)$.
- $X|_m^{\lll n}$ ($X|_m^{\ggg n}$): divide state block X into m-bit words and apply $\lll n$ ($\ggg n$) to each.
- $\delta(x, y, \mathtt{b})$: selection function where x, y are m-bit words and $\mathtt{b}$ is a single bit (0 or 1). Defined as

$$\delta(x, y, \mathtt{b}) = \begin{cases} x, & \mathtt{b} = 0 \\ y, & \mathtt{b} = 1 \end{cases} \tag{1}$$

- $\mathtt{blend}_m(X, Y, \mathtt{mask})$: blend two ω-bit state blocks X, Y, which are divided into m-bit words as $X = (x_{d-1}, \ldots, x_0)$ and $Y = (y_{d-1}, \ldots, y_0)$ for $d = \frac{\omega}{m}$. The d-bit mask $\mathtt{mask} = (\mathtt{b}_{d-1}, \ldots, \mathtt{b}_0)$ determines the output by

$$\mathtt{blend}_m(X, Y, \mathtt{mask}) = (\delta(x_{d-1}, y_{d-1}, \mathtt{b}_{d-1}), \ldots \delta(x_0, y_0, \mathtt{b}_0)) \tag{2}$$

III. The `SCMAC` Framework

AEAD schemes are essential for ensuring both confidentiality and integrity of messages. There are two main current approaches to designing stream cipher-based AEAD schemes:

- Encrypt-then-MAC. This approach involves encrypting the message first and then applying an independent MAC

$$C \leftarrow \mathrm{ENC}_{K_E}(M),\ T \leftarrow \mathrm{MAC}_{K_M}(C), \tag{3}$$

  exemplified by `SNOW-V-GCM`.
- Single-Pass Scheme. This approach processes the message only once to provide both confidentiality and integrity simultaneously, such as in `Rocca`. These designs typically use simpler update functions that do not have strong security claims on their own but follow the interleave absorption and squeezing paradigm, inspired by the sponge hashing mode used in SHA-3 (see Figure 1).

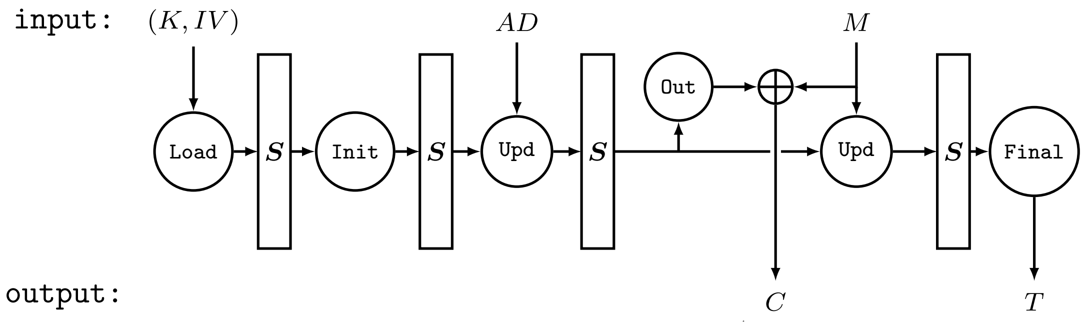

Fig. 1: Schematic diagram of interleave absorption and squeezing paradigm in typical single-pass AEAD schemes

Since the former can lead to efficiency losses and the latter may introduce potential vulnerabilities, we propose `SCMAC`, an alternative AEAD design framework, which bridges the gap between encrypt-then-MAC and dedicated single-pass designs. `SCMAC` is a design framework that transforms a stream cipher with large memory into an AEAD scheme. For a stream cipher, its initialization phase can be regarded as a thorough mixing of a large internal state, while the subsequent keystream generation phase can be adapted to generate keystream or produce authentication tags of desired lengths. `SCMAC` uses the stream cipher for encryption and tag generation after absorbing the message. The key idea lies in the absorption mechanism: instead of directly operating on the stream cipher's internal state, plaintext and associated data are first absorbed into an additional state called *MacState*. This *MacState* is initialized using part of the stream cipher's internal state post-initialization and updated independently but simultaneously from the encryption until the finalization phase, when it is fed back to the stream cipher's state. Then via reinitialization, the stream cipher generate a keystream of required length as authentication tags.

Due to its function as a stream cipher being independent, IVs must never be reused for the same key (nonce misuse scenario), and decryption should not release the unverified plaintext if the tag verification fails (decryption misuse scenario).

A. Specifications of SCMAC

A formal description of SCMAC is provided in Algorithm 2 and its specification in Algorithm 3, with an illustration in Figure 2. The input includes the secret key K, public IV/Nonce IV, message M, and associated data AD. The output consists of the ciphertext C and a tag T for authentication. The process of SCMAC are as follows:

1) Initialization. : Run the standard initialization process of the stream cipher, then copy some *ScState* bits to initialize *MacState*, as described in `Initialize`(K, IV).

2) Processing the associated data. : If $|AD| = 0$, skip this process. Otherwise, pad AD to $\overline{AD} = \overline{AD}_0 \| \ldots \| \overline{AD}_{d-1}$, where $d = \frac{|\overline{AD}|}{b}$, and update *MacState* by absorbing $\overline{AD}$, as described in `ProcessAD`$(\boldsymbol{E}, AD)$.

3) Encryption. : If M is empty, skip this process. Otherwise, pad M to $\overline{M} = \overline{M}_0 \| \ldots \| \overline{M}_{m-1}$, where $m = \frac{|\overline{M}|}{b}$. Update *MacState* by absorbing $\overline{M}$, meanwhile, generate the ciphertext by XORing $\overline{M}$ with keystream blocks and update *ScState*, as described in `Encrypt`$(\boldsymbol{S}, \boldsymbol{E}, M)$. If the last block of M is incomplete (length ℓ bits, where $0 < \ell < b$), truncate the last block of C to ℓ bits.

4) Finalization. : The lengths of the associated data and message are encoded[1] as a little-endian word $\Theta = |AD| \| |M|$. This word is then padded and absorbed by the *MacState*. Next, the bits of *MacState* are fed into the *ScState* to run the initialization rounds of the stream cipher for thorough diffusion. Finally, the first τ keystream bits are concatenated to form the tag $T = T_0 \| \ldots \| T_{\mu-1}$, where $\mu = \lceil \frac{\tau}{b} \rceil$. If the last block of T is incomplete (length of $\upsilon = \tau \mod b$ bits, where $0 < \upsilon < b$), it is truncated to the first υ bits. The detailed process is described in `Finalize`$(\boldsymbol{S}, \boldsymbol{E}, |AD|, |M|)$.

Algorithm 2: The Encryption/Decryption of SCMAC

```
procedure AeadEncrypt($K, IV, M, AD$)
    $\boldsymbol{S}, \boldsymbol{E} \leftarrow$ Initialize($K, IV$)
    if $|AD| > 0$ then
        $\boldsymbol{E} \leftarrow$ ProcessAD($\boldsymbol{E}, AD$)
    if $|M| > 0$ then
        $(\boldsymbol{S}, \boldsymbol{E}, C) \leftarrow$ Encrypt($\boldsymbol{S}, \boldsymbol{E}, M$)
    $T \leftarrow$ Finalize($\boldsymbol{S}, \boldsymbol{E}, |AD|, |M|$)
    return $(C, T)$
procedure AeadDecrypt($K, IV, C, T, AD$)
    $\boldsymbol{S}, \boldsymbol{E} \leftarrow$ Initialize($K, IV$)
    if $|AD| > 0$ then
        $\boldsymbol{E} \leftarrow$ ProcessAD($\boldsymbol{E}, AD$)
    if $|M| > 0$ then
        $(\boldsymbol{S}, \boldsymbol{E}, M) \leftarrow$ Decrypt($\boldsymbol{S}, \boldsymbol{E}, C$)
    if $T =$ Finalize($\boldsymbol{S}, \boldsymbol{E}, |AD|, |M|$) then
        return $M$
    else
        return $\perp$
```

B. Limits of the Tag Length τ

According to Algorithm 1 and Algorithm 2, the encryption security of SCMAC is guaranteed by the stream cipher itself. For authentication, forgeries can occur in two cases:

- Collision in the *MacState* $\boldsymbol{E}$ during the message absorption phase of `AeadEncrypt`.
- Collision in the tag T during the tag generation phase of `Finalize`.

Besides the Birthday Attack, the first case challenges the collision resistance of $\boldsymbol{E}$ and `updE`, while the second depends on the differential attack resistance of the stream cipher's initialization phase. We denote the complexity of finding an $\boldsymbol{E}$ collision as $\mathcal{C}_{\boldsymbol{E}}$ $(\leq |\boldsymbol{E}|/2)$ and that for finding a $\boldsymbol{S}$ collision as $\mathcal{C}_{\boldsymbol{S}}$. The maximum tag length $\tau_{\max}$ is defined as:

$$\tau_{\max} = \min\{\log \mathcal{C}_{\boldsymbol{E}}, \log \mathcal{C}_{\boldsymbol{S}}\} \tag{4}$$

[1]The encoding scheme is determined based on the upper bounds of the lengths of M and AD, along with the value of b.

Algorithm 3: The specification of `SCMAC`

```
procedure Initialization($K, IV$)
    Load $(K, IV)$: $\boldsymbol{S} \leftarrow \texttt{Load}(K, IV)$
    Run initialization rounds: $\boldsymbol{S} \leftarrow \texttt{scInit}(\boldsymbol{S})$
    Declare $MacState$: $\boldsymbol{E} \leftarrow \texttt{COPY}(\boldsymbol{S})$
    return $\boldsymbol{S}, \boldsymbol{E}$
procedure ProcessAD($\boldsymbol{E}, AD$)
    Set $d \leftarrow \lceil |AD|/b \rceil$
    Pad $AD$ to $\overline{AD} = \overline{AD}_0 \| \dots \| \overline{AD}_{d-1}$
    for $i = 0, \dots d-1$ do
        $\boldsymbol{E} \leftarrow \texttt{updE}(\boldsymbol{E}, \overline{AD}_i)$
    return $\boldsymbol{E}$
procedure Encrypt($\boldsymbol{S}, \boldsymbol{E}, M$)
    Set $m \leftarrow \lceil |M|/b \rceil$ and $\ell \leftarrow |M| \mod b$
    Pad $M$ to $\overline{M} = \overline{M}_0 \| \dots \| \overline{M}_{m-1}$
    for $i = 0, \dots m-1$ do
        $(Z_i, \boldsymbol{S}) \leftarrow \texttt{scOut}(\boldsymbol{S})$
        $C_i \leftarrow \overline{M}_i \oplus Z_i$
        $\boldsymbol{E} \leftarrow \texttt{updE}(\boldsymbol{E}, \overline{M}_i)$
    if $\ell > 0$ then
        $C_{m-1} \leftarrow \texttt{trunc}(C_{m-1}, \ell)$
    $C = C_0 \| \dots \| C_{m-1}$
    return $(\boldsymbol{S}, \boldsymbol{E}, C)$
procedure Decrypt($\boldsymbol{S}, \boldsymbol{E}, C$)
    Define the values $m \leftarrow \lceil |C|/b \rceil$ and $\ell \leftarrow |C| \mod b$
    Pad $C$ to $\overline{C} = \overline{C}_0 \| \dots \| \overline{C}_{m-1}$
    for $i = 0, \dots m-1$ do
        $M_i \leftarrow \overline{C}_i \oplus \texttt{scOut}(\boldsymbol{S})$
        $\boldsymbol{E} \leftarrow \texttt{updE}(\boldsymbol{E}, \overline{M}_i)$
    if $\ell > 0$ then
        $M_{m-1} \leftarrow \texttt{trunc}(M_{m-1}, \ell)$
    $M = M_0 \| \dots \| M_{m-1}$
    return $(\boldsymbol{S}, \boldsymbol{E}, M)$
procedure Finalize($\boldsymbol{S}, \boldsymbol{E}, |AD|, |M|$)
    Encode $|AD|, |M|$ as a little-endian word $\Theta \leftarrow |AD| \| |M|$ and pad $\Theta$ as $\overline{\Theta}$
    $\boldsymbol{E} \leftarrow \texttt{updE}(\boldsymbol{E}, \overline{\Theta})$
    $\boldsymbol{S} \leftarrow \texttt{Feedback}(\boldsymbol{S}, \boldsymbol{E})$
    $\boldsymbol{S} \leftarrow \texttt{scInit}(\boldsymbol{S})$
    With the tag length $\tau$, set $\mu \leftarrow \lceil \tau/b \rceil$ and $\upsilon \leftarrow \tau \mod b$
    for $i = 0, \dots \mu-1$ do
        $(T_i, \boldsymbol{S}) \leftarrow \texttt{scOut}(\boldsymbol{S})$
    if $\upsilon > 0$ then
        $T_{\mu-1} \leftarrow \texttt{trunc}(T_{\mu-1}, \upsilon)$
    $T = T_0 \| \dots \| T_{\mu-1}$
    return $T$
```

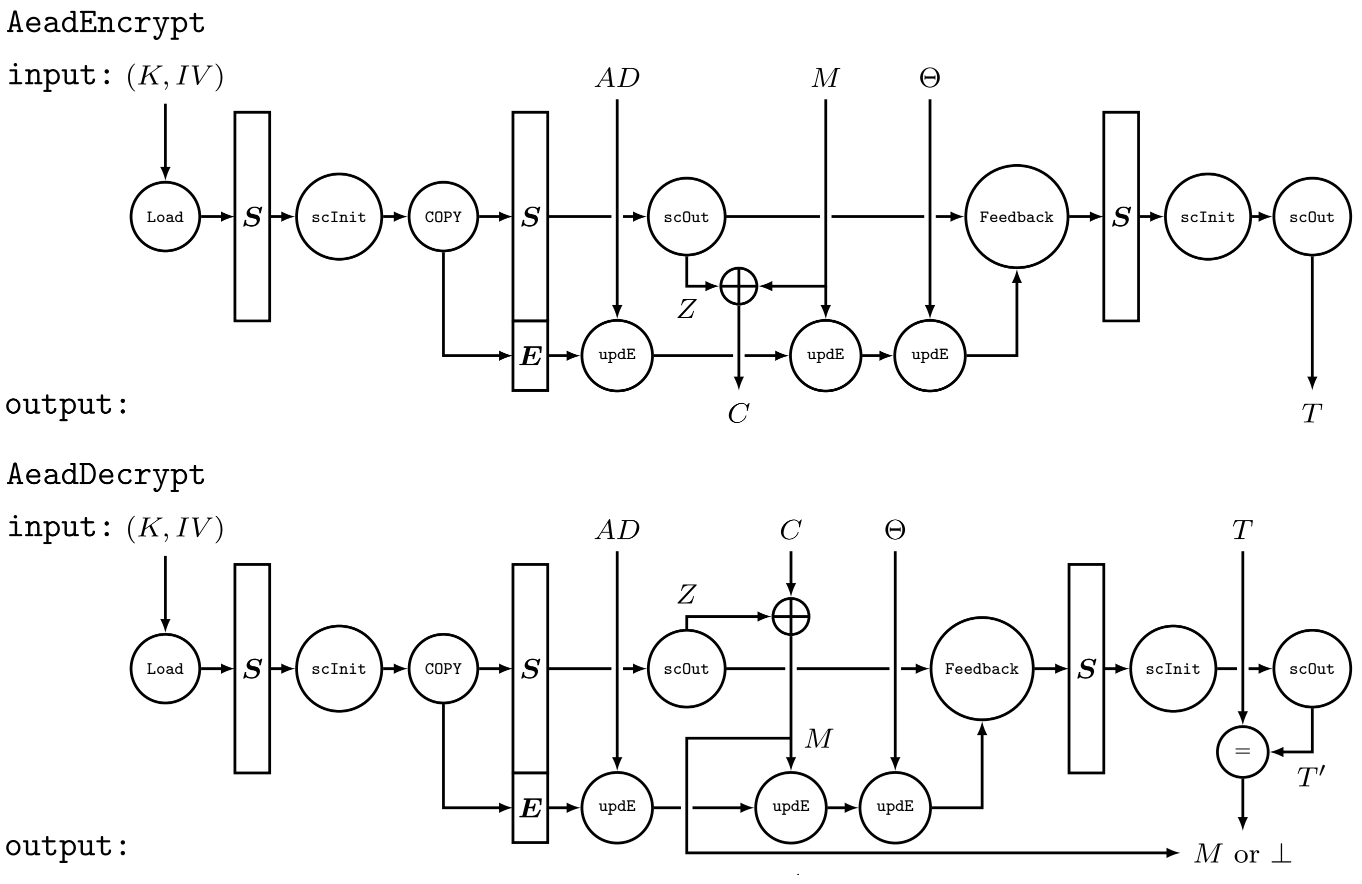

Fig. 2: AeadEncrypt and AeadDecrypt processes of SCMAC, where $\Theta = |AD| \| |M|$

under the constraint that the maximum number of different messages produced for a fixed key is less than $2^{\tau/2}$.

C. *Advantages of SCMAC Framework*

SCMAC design framework lies between Encrypt-then-MAC and Single-Pass Scheme. Encryption and authentication share the diffusion process, with encryption and information absorption performed in parallel and independently. This design offers two main benefits:

- Compared to Encrypt-then-MAC Scheme:
  SCMAC significantly improves efficiency and reduces the overhead of implementing two completely unrelated operations of encryption and MAC processes. For example, within the SNOW-V-GCM scheme, SNOW-V employs a "LFSR + FSM" cryptographic structure, with its underlying operations of modulo addition, XOR, bit shifting, and the AES round function. In contrast, GMAC's operations primarily consist of polynomial multiplication over the finite field $\mathbb{F}_{2^{128}}$, with its underlying operations being contingent upon the CPU's support for 64-bit carry-less multiplication instructions. The operational requirements for implementing these two algorithms differ significantly, rendering their implementation and optimization strategies incompatible.
  On the other hand, although the implementation performance of SNOW-V-GCM scheme is reduced from 58.25 Gbps of SNOW-V to 38.91 Gbps, the keystream width of 128 bits still makes the authentication framework of GMAC very easy to adopt to SNOW-V. Conversely, if schemes like LOL-Double use a keystream width of 256 bits, due to the inconsistency between the keystream width and the length of GMAC blocks, and the nonsupport of 128 or more bit no-carrier multiplication on current CPUs, each 256-bit keystream block needs to be split into two 128-bit blocks to participate in the AEAD operation, resulting in a significant reduction in the implementation performance (eg. LOL-Double's performance is over 140 Gbps, but LOL-Double-GCM's performance is about 40 Gbps). It is unable to demonstrate the high throughput advantage of LOL-Double like cipher.
  SCMAC can address these issues by employing the same update mechanism as scOUT via updE, while adopting a suitably sized MacState that matches the ScState of the stream cipher.
- Compared to Single-Pass Scheme: SCMAC enhances security by not adhering to the interleave absorption and squeezing paradigms. Instead, it independently absorbs the message into the *MacState* during encryption without immediate

output, thereby preventing consecutive information leakage. To demonstrate this, we refer to the general state-recovery attack [14] as a representative example.

If the tag size $|T|$ is smaller than the key size $|K|$, it becomes possible to exploit the decryption oracle to mount a general state recovery attack on the single-pass schemes that use weak permutations, such as Rocca [8], with a time complexity lower than $2^{|K|}$. The attack proceeds as follows:

1) The attacker makes an encryption query (IV, M) to obtain (C, T).
2) The attacker introduces a proper difference Δ to the ciphertext and makes a decryption query $(IV, C \oplus \triangle, T')$, attempting q_d possible values of T'. This process yields a "nonce-repeated" plaintext M' with a probability of $q_d/2^{|T|}$.
3) By exploiting the collected "nonce-repeated" pair, the attacker recovers the internal state according to the interleave message block absorption and ciphertext block squeezing paradigmas.

While the key-recovery attack can be prevented even if the state-recovery attack succeeds through some irreversible technology, this still poses a potential threat to the AEAD scheme, as attackers can extract more information from the full secret state. To effectively resist state recovery attacks, it is imperative that the tag length of the AEAD output in these designs matches the key length, thereby precluding the provision of variable-length tags.

Conversely, `SCMAC` is not susceptible to the aforementioned attack, even if an adversary obtains a "nonce-repeated" plaintext M'. This resilience stems from its design, which avoids simultaneous absorption and squeezing, thereby preventing internal state leakage and rendering it infeasible to recover the internal state or construct state collisions via manipulated messages. Therefore, `SCMAC` provides variable-length tags.

IV. The `LOL2.0` Framework

A. *Brief Description of* `LOL`

The `LOL` framework, proposed in [6], serves as a general approach for designing blockwise stream ciphers. The overall structure integrates a nonlinear driver, comprising an LFSR on the finite field concatenated with an NFSR, along with an FSM with large memories. Specifically, the LFSR consists of two ℓm-bit registers H, L, which operates by outputting L and updating according to Equation (5)

$$F \leftarrow f(H, L) = \lambda(H) \oplus \sigma(L),\ (H, L) \leftarrow (F, H), \tag{5}$$

which can be represented by Feistel-like structure as shown in Figure 3. In light of the effective utilization of the instruction set, $m \in \{8, 16, 32, 64\}$. Here, $\lambda(H) = C \times H$, where C denotes a vector of roots of ℓ m-degree irreducible polynomials in $\mathbb{F}_2[x]$, and $\times$ denotes cell-to-cell finite field multiplication in $\mathbb{F}_{2^m}$. σ denotes a cell-wise permutation, and XOR integrates different expansion fields via polynomial basis correspondence in $\mathbb{F}_{2^m}$. With appropriate λ and σ, the LFSR can be proved an m-sequence with the maximum period of $2^{2\ell m} - 1$. Such LFSRs are denoted as `LFSR1` throughout this paper.

The NFSR, concatenated with the LFSR's L register, consists of an ℓm-bit register N. It operates by outputting and updating the entire N as Equation (6)

$$N \leftarrow \mathcal{R}(N) \oplus L, \tag{6}$$

where $\mathcal{R}$ is an SPN function.

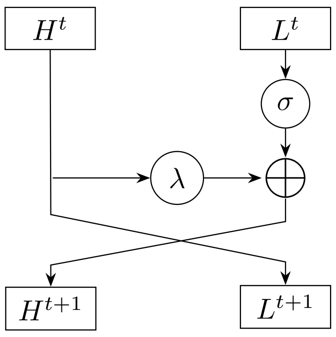

Fig. 3: The Feistel-like structure of LFSR updating function

The FSM consists of n ℓm-bit registers $(S_0, S_1, \ldots, S_{n-1})$, updated in parallel by SPN functions $\mathcal{R}$ as shown in Equation (7), inspired by [7].

$$G \leftarrow \mathcal{R}(S_{n-1}),\ S_i \leftarrow \mathcal{R}(S_{i-1}) \oplus S_i \text{ for } i = n-1, \ldots, 1,\ S_0 \leftarrow S_0 \oplus F \oplus G. \tag{7}$$

Here, S_0 and S_{n-1} serve as connectors for interfacing with the nonlinear driver or supporting potential structural extensions, while the other registers serve as plugins adjusted according to the requirement of security and efficiency. The keystream block Z is generated by Equation (8)

$$Z \leftarrow h(G, N) = G \oplus N. \tag{8}$$

The initialization phase first loads K and IV into the FSM while setting all nonlinear driver to a constant; then, runs an r-round updating iterations during which the keystream block Z^t $(t = 0, \ldots, r-1)$ is not output but fed back to the nonlinear driver as Equation (9); adopts FP-(1) mode [18], wherein the key is XORed with the internal state at the final round of initialization phase.

$$\begin{aligned} H &\leftarrow F \oplus Z, \\ N &\leftarrow \mathcal{R}(N) \oplus L \oplus Z. \end{aligned} \tag{9}$$

The structure above is referred to as the `LOL` *single mode*. To expand the keystream block size by a factor of $\gamma \geq 2$, we only need to combine γ `LOL` single mode primitives following the `LOL` *parallel-γ mode*: for the nonlinear driver, LFSRs are integrated into a unified whole and its updating function f is redefined with appropriate parameters (λ, σ) to preserve the maximum period, while NFSRs keep their updating method fed into the divisions of L separately; for the FSM, connectors are interconnected end to end and fed into the divisions of F, while plugins retain their updating method; the output functions remain unchanged and their outputs are concatenated to form the expanded keystream block. Especially when $\gamma = 2$, the expansion is called a *parallel-dual mode*. A concrete description will be provided through specific cipher examples. For more details, please refer to [6].

B. Upgrades in `LOL2.0`

The `LOL2.0` framework, in the parts of structure, update mechanism, initialization procedure and expansion rule, is exactly the same as in `LOL`, with only two differences as follows.

1) Making `LOL2.0` more flexible by adding new LFSR construction methods (defined as `LFSR2`): Our objective is to design an LFSR that is well-suited for efficient software implementation. In contrast to the LFSR design in `LOL`, the key idea of newly proposed methods is that such a construction does not require reliance on the finite field $\mathbb{F}_{2^m}$. Instead, we aim to exploit any linear update function with basic software operations to achieve rapid updates of large-width primitive Galois LFSRs. Specifically, this involves minimizing the use of shift, circular shift, and XOR operations while ensuring high performance.

Another three construction methods for the an $2\ell m$-bit primitive LFSR are provided as follows. The LFSR also consists of two ℓm-bit registers H, L, and employs the update function $F \leftarrow f(H, L)$, $(H, L) \leftarrow (F, H)$. Construction 1-3 preserve the update mechanism of the LFSR as $f(H, L) = \lambda(H) \oplus \sigma(L)$ as described in Equation (8), substituting the ℓ parallel multiplications over the finite field $\mathbb{F}_{2^m}$, i.e., $\lambda(H) = C \times H$, with circular shift or shift operations, while leaving all other aspects unchanged. When selecting the parameters, we should ensure the maximum period of $2^{2\ell m} - 1$ for the LFSR. The verification method is similar with that demonstrated in [6]: define the generation matrix $G \in \mathbb{F}_2^{2\ell m \times 2\ell m}$ of the LFSR ensuring that $(f(H, L), H)^T = G(H, L)^T$ according to the linear operations in f; deduce the characteristic polynomial of G as $\det(xI \oplus G) = \eta(x) \in \mathbb{F}_2[x]$ and verify its primitivity using third-party mathematical software libraries, such as NTL or MAPLE. Then, the LFSR is primitive as long as $\eta(x)$ is primitive. For clarity and conciseness, we provide some instances of 256-bit LFSRs with 128-bit output per cycle for each method. In particular, for Methods 1-3, the parameter σ in the update function adopts the values used in `LOL-Mini`, namely,

$$(x_7, \ldots, x_1, x_0) \xrightarrow{\sigma} (x_5, x_0, x_3, x_6, x_4, x_7, x_2, x_1). \tag{10}$$

Construction 1: $f(H, L) = \lambda(H) \oplus \sigma(L)$, where $\lambda(H)$ is composed of cyclic shifts and shifts of m-bit cells. The examples are as follows:

$$\lambda(H) = (h_7 \ll 7, h_6 \ll 5, h_5, h_4, h_3 \lll 1, h_2, h_1, h_0)$$
$$\lambda(H) = (h_7 \ll 5, h_6 \ll 7, h_5, h_4, h_3 \lll 1, h_2, h_1, h_0)$$
$$\lambda(H) = (h_7 \ll 9, h_6 \ll 3, h_5, h_4, h_3 \lll 1, h_2, h_1, h_0)$$

Next, we consider their implementation. When AVX2 instructions are employed, the circular shift on m-bit cells necessitates two shift operations and one XOR operation. Thus, it requires 4 shift operations, 1 XOR operation and 3 blend operations to implement $\lambda(H)$. A blend operation involves selecting packed m-bit cells from two registers based on a control mask, as given in Equation (2). However, when AVX512 instructions are employed, the circular shift on m-bit cells and the operation that shifts the bits of packed m-bit cells by varying numbers of bits simultaneously are both supported in 1 latency. In this way, it requires 1 shift operations, 1 circular shift operation and 1 blend operations to implement $\lambda(H)$.

Construction 2: $f(H,L)=\lambda(H)\oplus\sigma(L)$, where $\lambda(H)$ is composed of only shifts of m-bit cells. The examples are as follows:

$$\lambda(H)=(h_7\lll 3, h_6\lll 5, h_5, h_4, h_3\ggg 11, h_2, h_1, h_0)$$
$$\lambda(H)=(h_7\lll 7, h_6, h_5, h_4\ggg 5, h_3\ggg 9, h_2, h_1, h_0)$$

Next, we consider their implementation. When AVX2 instructions are employed, it requires 3 shift operations and 3 blend operations to implement $\lambda(H)$. However, when AVX512 instructions are employed, it requires 2 shift operations and 1 blend operations to implement $\lambda(H)$.

Construction 3: To address the challenge of non-synchronous shift operations in Method 2, we propose utilizing operations on cells of varying sizes of H. Specifically, we note that the left shift operation on a 32-bit cell can be equivalently expressed using operations on 16-bit cells as follows:

$$\begin{aligned}((h_7\|h_6)\lll i,(h_5\|h_4)\lll i,(h_3\|h_2)\lll i,(h_1\|h_0)\lll i) =&(h_7\lll i\oplus h_6\ggg(16-i), h_6\lll i, h_5\lll i\oplus h_4\ggg(16-i), h_4\lll i,\\ &h_3\lll i\oplus h_2\ggg(16-i), h_2\lll i, h_1\lll i\oplus h_0\ggg(16-i), h_0\lll i),\end{aligned}$$

which simultaneously includes left shift and right shift operations on 16-bit cells. Thus, using a 256-bit LFSR as an example, we partition H into 32-bit cells and 16-bit cells respectively, and execute parallel shift operations on 32-bit cells and 16-bit cells independently. Subsequently, we integrate the results of the two shift operations through a blend operation on the 16-bit cells, thereby realizing $\lambda(H)$. All aforementioned operations are supported by both AVX2 and AVX512, with a latency of 1 for each. For $f(H,L)=\lambda(H)\oplus\sigma(L)$, the examples are as follows:

$$\lambda(H)=(h_7\lll 5\oplus h_6\ggg 11, h_6\lll 5, h_5\lll 5\oplus h_4\ggg 11, h_4\ggg 6, h_3\ggg 6, h_2\lll 5, h_1\lll 5\oplus h_0\ggg 11, h_0\ggg 6)$$
$$\lambda(H)=(h_7\ggg 4, h_6\lll 7, h_5\ggg 4, h_4\lll 7, h_3\ggg 4, h_2\lll 7, h_1\lll 7\oplus h_0\ggg 9, h_0\lll 7)$$
$$\lambda(H)=(h_7\ggg 2, h_6\ggg 2, h_5\lll 9\oplus h_4\ggg 7, h_4\lll 9, h_3\ggg 2, h_2\ggg 2, h_1\ggg 2, h_0\lll 9)$$
$$\lambda(H)=(h_7\lll 10\oplus h_6\ggg 6, h_6\ggg 13, h_5\ggg 13, h_4\lll 10, h_3\lll 10\oplus h_2\ggg 6, h_2\lll 10, h_1\lll 10\oplus h_0\ggg 6, h_0\ggg 13)$$

As we discussed above, regardless of whether $\lambda(H)$ is implemented using AVX2 or AVX512, 2 shift operations and 1 blend operation are suffice.

Given that all the above LFSRs have passed the primality test, we only analyze them from the perspective of software implementation, including `LFSR1`.

`LFSR1` $f(H,L)=\lambda(H)\oplus\sigma(L)$, where $\lambda(H)=C\times H$. The parallel finite field multiplication can be implemented by 2 shift operations, 1 AND operation and 1 XOR operation using either AVX2 or AVX512.

Furthermore, a cell-wise permutation σ within a 128-bit register can be implemented by 1 shuffle operation using AVX2, whereas implementing σ within a 256-bit register using AVX2 requires 2 shuffle operations, 1 permute operation, and 1 XOR operation. For implementing σ within a 256-bit register using AVX512, a single permute operation suffices. Herein, the permute operation has a latency of 3, while all other operations exhibit latency 1. The number of operations required for these LFSR constructions are summarized in Table I.

TABLE I: The number of instruments required for these LFSR constructions

Instrument set		AVX2				AVX512			
Construction		`LFSR1`	Method1	Method2	Method3	`LFSR1`	Method1	Method2	Method3
f	λ	4	8	6	3	4	3	3	3
	σ	1 (128-bit register); 4 (256-bit register)				1			
	$\oplus$	1				1			

Besides, we also deviate from the update mode specified in Equation (8) by employing a permutation σ, and instead directly construct an ℓm-bit update function. This function utilizes circular shift, shift, and XOR operations, formulated as $f(H)=(L\lll r_1)\oplus H\oplus(H\gg r_2)$ with $0<r_1,r_2<\ell m$. The examples are as follows:

$$\begin{aligned}(r_1,r_2)\in\{&(5,17),(31,17),(37,107),(47,23),(49,75),(53,89),(59,97),(79,17),\\&(91,97),(105,7),(107,67),(109,19),(109,113),(69,11)\}\end{aligned}$$

However, owing to the absence of dedicated instructions for full-register circular shift operations, implementing this method proves challenging regardless of whether AVX2 or AVX512 is employed. Consequently, we opted to discard this type of construction scheme.

From Table I, it is evident that Method 3 exhibits superior universality for both AVX2 and AVX512, we opt for Method 3 as `LFSR2` in `LOL2.0` in this context. Furthermore, we formalize this type of construction method as:

LFSR2 The LFSR consists of two ℓm-bit registers H, L, and employs the 128-bit width update function $f(H,L) = \lambda(H) \oplus \sigma(L)$, where

$$\lambda(H) = \texttt{blend}_m \left(H|^{\lll n_1}_{\rho_1 m}, H|^{\ggg n_2}_{\rho_2 m}, \texttt{mask} \right). \tag{11}$$

Here, ρ_1, ρ_2, n_1, n_2 are integers satisfying $1 \leq \rho_1, \rho_2 < \ell$, $0 \leq n_1 < \rho_1 m$ and $0 \leq n_2 < \rho_2 m$.

Such LFSRs can be searched for by adjusting the parameters ρ_1, ρ_2, n_1, n_2, and `mask`, as well as the cell-wise permutation σ, which allows us to construct a series of primitive LFSRs whose large state can be efficiently updated by half in each cycle.

2) Making `LOL2.0` more secure by optimizing the update function of FSM and the output function of keystream: The FSM in `LOL2.0` also comprises n ℓm-bit registers $(S_0, S_1, \ldots, S_{n-1})$ similar to that in `LOL`. However, the update function of the FSM is modified in the generation of G, compared to Equation (7), as shown in Equation (12):

$$\begin{cases} G \leftarrow \mathcal{R}(S_{n-1}) \oplus N, \\ \textbf{for } i = n-1, \ldots, 1 \textbf{ do } S_i \leftarrow \mathcal{R}(S_{i-1}) \oplus S_i, \\ S_0 \leftarrow S_0 \oplus F \oplus G, \end{cases} \tag{12}$$

which adds interactions between the NFSR and the FSM so as to make better diffusion. In this manner, in contrast to Equation (7), all the round key additions in $\mathcal{R}$ function are fully exploited.

The output function in `LOL2.0` to generate keystream block Z^t has been upgraded as shown in Equation (13)

$$Z^t = h(G^t, N^t) = \mathcal{R}(G^t) \oplus N^t, \tag{13}$$

which incorporates an additional $\mathcal{R}$ operation compared to Equation (8) in `LOL`.

V. The `LOL2.0` Ciphers with `SCMAC`

In this section, we present 2 stream cipher-based AEAD instances following the `LOL2.0` framework and `SCMAC` framework. These 2 ciphers are named `LOL2.0-Mini` and `LOL2.0-Double` respectively: the former adopts the `LOL2.0` single mode while the latter follows the parallel-dual mode. We first present some universal preliminaries here. In cases where no ambiguity arises, we make no distinction between the stream cipher mode `LOL2.0-Mini` (`LOL2.0-Double`) and the AEAD mode `LOL2.0-Mini-SCMAC`(`LOL2.0-Double-SCMAC`), abbreviating both uniformly as `LOL2.0-Mini` (`LOL2.0-Double`).

A. Instantiations

The `LOL2.0` ciphers in this section employ 128-bit state blocks as storage units in order to ensure the suitability of SIMD implementation. A 128-bit state block x can be divided into sixteen 8-bit bytes $b_{15}, \ldots, b_0$, or into eight 16-bit words $w_7, \ldots, w_0$, or into four 32-bit words $u_3, \ldots, u_0$, where w_0, b_0, u_0 are the least significant units, i.e. $x = (b_{15}, \ldots, b_0) = (w_7, \ldots, w_0) = (u_3, \ldots, u_0)$.

All $\mathcal{R}$ functions are defined as a compound of AES SubByte (SB), ShiftRow (SR), and MixColumn (MC) operations, which is equivalent to the AES round function without AddRoundKey (ARK), i.e. $\mathcal{R}(x) = \text{MC} \circ \text{SR} \circ \text{SB}(x)$. The mapping between a 128-bit state block $x = (w_{15}, \ldots, w_0)$ and the 4×4-byte state array of the AES round function is as follows

$$x = \begin{pmatrix} w_0 & w_4 & w_8 & w_{12} \\ w_1 & w_5 & w_9 & w_{13} \\ w_2 & w_6 & w_{10} & w_{14} \\ w_3 & w_7 & w_{11} & w_{15} \end{pmatrix} \tag{14}$$

Based on the LFSR construction method outlined in Section IV-B, we present several choices for the `LFSR2` used in the `LOL2.0` ciphers in this section, as detailed in Table II. The λs are obtained under the same parameters of m, ℓ and σ as described in [6].

TABLE II: Some choices for the `LFSR2` used in the `LOL2.0` ciphers

No.	$m = 16, \ell = 8, \sigma$:Equation (22)	$m = 16, \ell = 16, \sigma$:Equation (27)
1	$\lambda_0 : \texttt{blend}_{16} \left(x\|^{\lll 5}_{32}, x\|^{\ggg 6}_{16}, \texttt{0x19} \right)$	$(\lambda_1, \lambda_0) : \texttt{blend}_{16} \left(x\|^{\lll 5}_{32}, x\|^{\ggg 6}_{16}, \texttt{0x5619} \right)$
2	$\lambda_0 : \texttt{blend}_{16} \left(x\|^{\lll 7}_{32}, x\|^{\ggg 4}_{16}, \texttt{0xa8} \right)$	$(\lambda_1, \lambda_0) : \texttt{blend}_{16} \left(x\|^{\lll 7}_{32}, x\|^{\ggg 4}_{16}, \texttt{0xdba8} \right)$
3	$\lambda_0 : \texttt{blend}_{16} \left(x\|^{\lll 9}_{32}, x\|^{\ggg 2}_{16}, \texttt{0xce} \right)$	$(\lambda_1, \lambda_0) : \texttt{blend}_{16} \left(x\|^{\lll 9}_{32}, x\|^{\ggg 2}_{16}, \texttt{0x2cce} \right)$
4	$\lambda_0 : \texttt{blend}_{16} \left(x\|^{\lll 10}_{32}, x\|^{\ggg 13}_{16}, \texttt{0x61} \right)$	$(\lambda_1, \lambda_0) : \texttt{blend}_{16} \left(x\|^{\lll 10}_{32}, x\|^{\ggg 13}_{16}, \texttt{0xb961} \right)$

For clarity, we provide an equivalent representation of the λ given in Table II, as shown in Table III.

TABLE III: Illustrations for the LFSR2 corresponding to Table II

No.	Illustration
1	$\lambda_0 = (h_7 \ll 5 \oplus h_6 \gg 11, h_6 \ll 5, h_5 \ll 5 \oplus h_4 \gg 11, h_4 \gg 6, h_3 \gg 6, h_2 \ll 5, h_1 \ll 5 \oplus h_0 \gg 11, h_0 \gg 6)$
	$\lambda_1 = (h_7 \ll 5 \oplus h_6 \gg 11, h_6 \gg 6, h_5 \ll 5 \oplus h_4 \gg 11, h_4 \gg 6, h_3 \ll 5 \oplus h_2 \gg 11, h_2 \gg 6, h_1 \gg 6, h_0 \ll 5)$
2	$\lambda_0 = (h_7 \gg 4,\ h_6 \ll 7,\ h_5 \gg 4,\ h_4 \ll 7,\ h_3 \gg 4,\ h_2 \ll 7,\ h_1 \ll 7 \oplus h_0 \gg 9, h_0 \ll 7)$
	$\lambda_1 = (h_7 \gg 4, h_6 \gg 4, h_5 \ll 7 \oplus h_4 \gg 9, h_4 \gg 4, h_3 \gg 4, h_2 \ll 7, h_1 \gg 4, h_0 \gg 4)$
3	$\lambda_0 = (h_7 \gg 2, h_6 \gg 2, h_5 \ll 9 \oplus h_4 \gg 7, h_4 \ll 9, h_3 \gg 2, h_2 \gg 2, h_1 \gg 2, h_0 \ll 9)$
	$\lambda_1 = (h_7 \ll 9 \oplus h_6 \gg 7,\ h_6 \ll 9,\ h_5 \gg 2,\ h_4 \ll 9,\ h_3 \gg 2, h_2 \gg 2, h_1 \ll 9 \oplus h_0 \gg 7, h_0 \ll 9)$
4	$\lambda_0 = (h_7 \ll 10 \oplus h_6 \gg 6, h_6 \gg 13, h_5 \gg 13, h_4 \ll 10, h_3 \ll 10 \oplus h_2 \gg 6, h_2 \ll 10, h_1 \ll 10 \oplus h_0 \gg 6, h_0 \gg 13)$
	$\lambda_1 = (h_7 \gg 13, h_6 \ll 10, h_5 \gg 13, h_4 \gg 13, h_3 \gg 13\ h_2 \ll 10, h_1 \ll 10 \oplus h_0 \gg 6, h_0 \gg 13)$

To make SCMAC embrace good resistance against collision attacks, we propose a series of collision resisting $(\boldsymbol{E}, \texttt{updE})$ designs, inspired by the ideas in [19], [20]. The *MacState* $\boldsymbol{E}$ consists of μ 128-bit state blocks as defined in Equation (15)

$$\boldsymbol{E} = (E_0, \ldots, E_{\mu-1}). \tag{15}$$

The update function updE of *MacState* $\boldsymbol{E}$ absorbs a data block D consisting of ν 128-bit state blocks as defined in Equation (16):

$$D = (D_{\nu-1}, \ldots, D_0) \tag{16}$$

each clock. The process of updating *MacState* $\boldsymbol{E} \leftarrow \texttt{updE}(\boldsymbol{E}, D)$ adheres to the structure illustrated in Figure 4, where $\mathcal{R}$ represents the SPN function following the aforementioned definition. The definition of updE is determined by 3

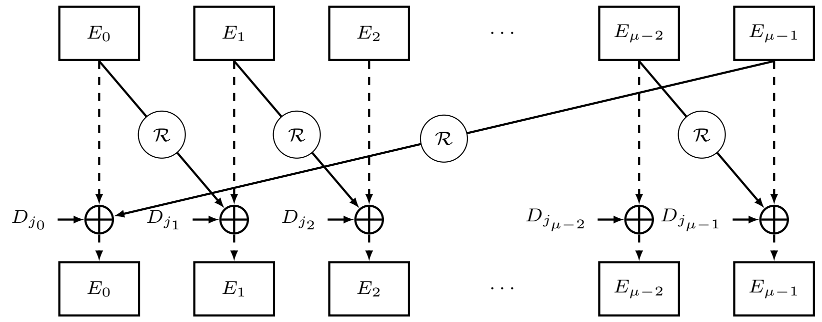

Fig. 4: The *MacState* updating function updE [19], [20]

parameters: the $\mathcal{R}$ mask $\Gamma_{\mathcal{R}} \in \mathbb{F}_2^{\mu}$, the feed mask $\Gamma_{\texttt{FD}} \in \mathbb{F}_2^{\mu}$ and the message sequence $\texttt{MSG} \in [0, \nu]^{\mu}$. The $\mathcal{R}$ mask $\Gamma_{\mathcal{R}}$ determines whether the $\mathcal{R}$ function in the slashes of Figure 4 is applied: $\mathcal{R}(E_i)$ is executed only if $\Gamma_{\mathcal{R}}[i] = 1$; otherwise, E_i is directly fed into the updating process of E_{i+1}. The feed mask $\Gamma_{\texttt{FD}}$ controls the forward feeding operations represented by the horizontal dashed lines: if $\Gamma_{\texttt{FD}}[i] = 1$, the computation of E_i can be expressed as the form $E_i \leftarrow E_i \oplus \star$. The message sequence $\texttt{MSG} \in [0, \nu]^{\mu}$ should satisfy the following criterion:

- All integers $1, \ldots, \nu$ are included in the sequence $\texttt{MSG}[i],\ i = 1, \ldots, \mu$
- If $\texttt{MSG}[i] = 0\ (i = 0, \ldots, \mu - 1)$, then $D_{j_i} = 0^{128}$; otherwise, $D_{j_i} = D_{\texttt{MSG}[i]-1}$

The detailed definition of updE is given in Algorithm 4. For non-negative integer e, we define $\mathcal{R}^e$ as e consecutive $\mathcal{R}$ operations as follow:

$$\mathcal{R}^e(x) = \begin{cases} \mathcal{R} \circ \ldots \circ \mathcal{R}(x) & e \geq 1 \\ x & e = 0 \end{cases},\ x \in \mathbb{F}_2^{128} \tag{17}$$

In addition to the 3 parameters above, there is also a *rate* parameter defined as

$$\rho = \frac{HW(\Gamma_{\mathcal{R}})}{\nu} \tag{18}$$

As extensively studied, the collision resistance of such updE's is bounded by the minimum number of active S-boxes, denoted as #ASB, in the differential characteristic of the following form:

$$\mathbf{0} = \Delta \boldsymbol{E}^0 \xrightarrow[\texttt{updE}]{\Delta D^0} \Delta \boldsymbol{E}^1 \xrightarrow[\texttt{updE}]{\Delta D^1} \ldots \xrightarrow[\texttt{updE}]{\Delta D^{r-1}} \Delta \boldsymbol{E}^r = \mathbf{0} \tag{19}$$

Algorithm 4: The definition of $\texttt{updE}$ with predefined parameters $\Gamma_{\mathcal{R}}, \Gamma_{\texttt{FD}}, \texttt{MSG}$.

```
  Input: The MacState $\boldsymbol{E} = (E_0, \ldots, E_{\mu-1}) \in (\mathbb{F}_2^{128})^\mu$. The data block $\boldsymbol{D} = (D_{\nu-1}, \ldots, D_0) \in (\mathbb{F}_2^{128})^\nu$
  Output: The updated MacState $\boldsymbol{E}' = (E'_0, \ldots, E'_{\mu-1}) \in (\mathbb{F}_2^{128})^\mu$
1 for $i = 0, \ldots, \mu-1$ do
2     Define
          $M_i \leftarrow \begin{cases} 0^{128} & \texttt{MSG}[i] = 0 \\ D_{\texttt{MSG}[i]-1} & \texttt{MSG}[i] > 0 \end{cases}$
3     Define
          $T_i \leftarrow \begin{cases} 0^{128} & \Gamma_{\texttt{FD}}[i] = 0 \\ E_i & \Gamma_{\texttt{FD}}[i] = 1 \end{cases}$
4     Compute $E'_i \leftarrow \mathcal{R}^{\Gamma_{\mathcal{R}}[i]}(E_{(i-1) \bmod \mu}) \oplus M_i \oplus T_i$
5 Define $\boldsymbol{E}' = (E'_0, \ldots, E'_{\mu-1})$
6 return $\boldsymbol{E}'$
```

With the knowledge of $\#ASB$ and the differential properties of the AES Sbox, we can bound the complexity of finding $\boldsymbol{E}$ collisions as $\mathcal{C}_{\boldsymbol{E}} \geq 2^{6\times\#ASB}$. The $\#ASB$ for different $\Gamma_{\mathcal{R}}, \Gamma_{\texttt{FD}}, \texttt{MSG}$ parameters are given in Table IV.

The statistics in Table IV are acquired with the MILP based automatic cryptanalysis method. Specifically, for a given $\texttt{updE}$ definition, we construct MILP model $\mathcal{M}_r$'s capturing the truncated differential characteristic in Equation (19) for $r = 1, 2, \ldots$. By solving $\mathcal{M}_r$, we can acquire the minimum number of active S-boxes, denoted as $\#ASB_r$, of all r-round truncated differential characteristics. We find that when r grows to a specific threshold, denoted as $\hat{r}$, the $\#ASB_r$ value can no longer change. Therefore, the $\#ASB$'s in Table IV are set to

$$\#ASB = \min_{r\in[1,\hat{r}]} \#ASB_r.$$

TABLE IV: Some collision resisting *MacState* updating functions. Note that, following [20], the entries of $\Gamma_{\mathcal{R}}$, Γ_{FD}, $\texttt{MSG}$ are indexed from 0 to $\mu - 1$ in a left-to-right manner.

No.	μ	ν	$\Gamma_{\mathcal{R}}$	Γ_{FD}	MSG	#ASB	ρ	Source
1	3	1	111	**1	100	14	3	This paper
2	4	1	1111	1111	1000	35	4	
3	4	1	1111	0111	1000	35	4	
4	4	1	1111	***1	1000	34	4	
5	4	1	1111	**10	1000	30	4	
6	4	1	1111	*100	1000	26	4	
7	4	1	1111	*000	1000	25	4	
8	5	1	11111	11111	10000	56	5	
9	5	2	11111	10010	10201	15	2.5	
10	6	2	111111	011100	102110	27	3	[20]
11	6	2	111111	100100	011022	22	3	
12	6	2	111111	100100	122020	26	3	

B. $\texttt{LOL2.0-Mini}$ Cipher

1) As a stream cipher: $\texttt{LOL2.0-Mini}$ adopts the single mode of the $\texttt{LOL2.0}$ framework. It supports a 256-bit key denoted as as $K = (K_h, K_\ell)$ and a 128-bit IV. The internal state of $\texttt{LOL2.0-Mini}$ consists of six 128-bit blocks: 2 for the LFSR denoted as (H, L), 1 for the NFSR denoted as N and 3 for the FSM denoted as (S_0, S_1, S_2). The internal state of $\texttt{LOL2.0-Mini}$ is naturally defined as

$$\boldsymbol{S} = (L, H, N, S_0, S_1, S_2) \tag{20}$$

$\texttt{LOL2.0-Mini}$ cipher outputs a 128-bit keystream block at each step.

The LFSR of $\texttt{LOL2.0-Mini}$ cipher consist of sixteen 16-bit cells denoted as $a_{15}, \ldots, a_1, a_0$ (equivalent to the $\ell = m = 16$ situation), which are stored in the two 128-bit registers (H, L) as

$$H = (a_{15}, \ldots, a_9, a_8), L = (a_7, \ldots, a_1, a_0). \tag{21}$$

The updating function $f(H, L) = \lambda(H) \oplus \sigma(L)$, of the same form as Equation (5), has two options denoted as $\texttt{LFSR1}$ and $\texttt{LFSR2}$ respectively, sharing the same σ definition as Equation (22):

$$(x_7, \ldots, x_1, x_0) \xrightarrow{\sigma} (x_5, x_0, x_3, x_6, x_4, x_7, x_2, x_1) \tag{22}$$

They only differs in the λ settings: The λ of `LFSR1` is identical to that used in `LOL-Mini` in [6]. To ensure the completeness of the design, we provide a detailed definition of λ of `LFSR1` in Section A. For `LFSR2`, the λ employs parameter No.1 in Table II, i.e., defined as Equation (23)

$$\lambda(x) = \mathtt{blend}_{16}\left(x|_{32}^{\lll 5}, x|_{16}^{\ggg 6}, \mathtt{0x19}\right) \tag{23}$$

which is equivalent to

$$\lambda(x) = (h_7 \lll 5 \oplus h_6 \ggg 11, h_6 \lll 5, h_5 \lll 5 \oplus h_4 \ggg 11, h_4 \ggg 6, h_3 \ggg 6, h_2 \lll 5, h_1 \lll 5 \oplus h_0 \ggg 11, h_0 \ggg 6). \tag{24}$$

The encryption/decryption process of `LOL2.0-Mini` has been summarized in Algorithm 1. The concrete definitions of the subroutines `Load`, `scInit` and `scOut` are given in Algorithm 5. As can be seen, the secret key and public IV are loaded to the FSM registers while setting all nonlinear driver bits to 0 or other constants. Then, `LOL2.0-Mini` runs 12 initialization rounds when the 128-bit keystream block Z's are not output but instead fed back to the nonlinear driver for thorough diffusion. Subsequently, `LOL2.0-Mini` adopts the FP-(1) mode [18] by XORing the key to the internal state at the end of `scInit` making it irreversible. Thereafter, `LOL2.0-Mini` starts the generation of the keystream. Taking the convention of existing stream cipher designs, `LOL2.0-Mini` also upper bounds the keystream length to 2^{64} for a single key-IV pair, and each key may be used with a maximum of 2^{64} different IVs. The initialization and keystream generation phases of `LOL2.0-Mini` are illustrated as Figure 5 and Figure 6 respectively.

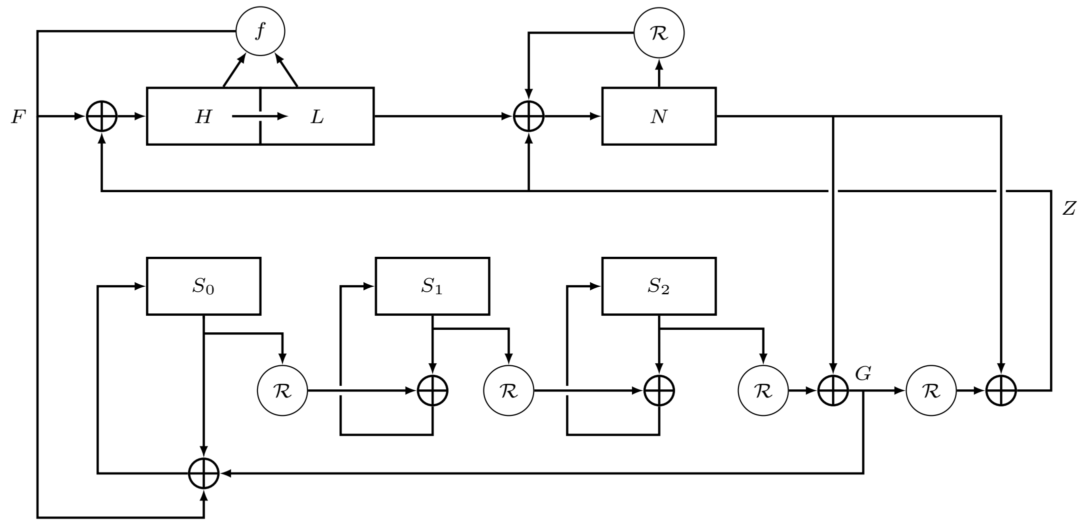

Fig. 5: The schematic of `LOL2.0-Mini` cipher in the initialization phase

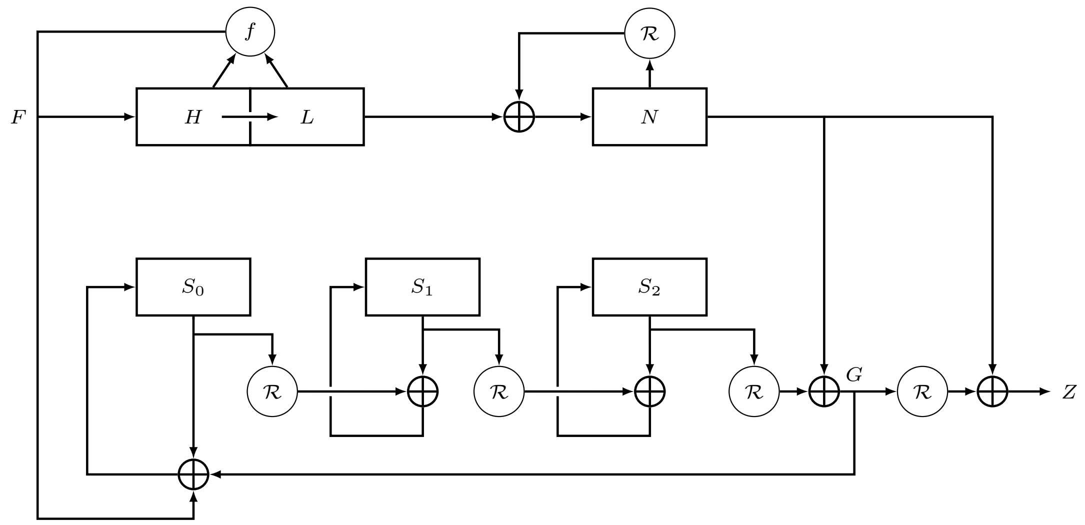

Fig. 6: The schematic of `LOL2.0-Mini` cipher in the keystream generation phase

Algorithm 5: The subroutines `LOL2.0-Mini` where the secret key K is of 256 bits and the public vector IV is of 128 bits. The internal state $\boldsymbol{S}$ is denoted as $\boldsymbol{S} = (L, H, N, S_0, S_1, S_2)$, and the *MacState* $\boldsymbol{E}$ is denoted as $\boldsymbol{E} = (E_0, E_1, E_2, E_3)$.

```
1  procedure Load($K, IV$)
2      Split the 256-bit key $K$ as $K = (K_h, K_\ell)$
3      Initialize $\boldsymbol{S} = (L, H, N, S_0, S_1, S_2) \leftarrow (0, 0, 0, K_h, K_\ell, IV) \in (\mathbb{F}_2^{128})^6$
4      return $\boldsymbol{S}$
5  procedure scOut($\boldsymbol{S}$)
6      Compute the intermediate value $G \leftarrow \mathcal{R}(S_2) \oplus N$
7      Compute the LFSR feedback $F \leftarrow f(H, L)$
8      Compute the keystream block $Z \leftarrow \mathcal{R}(G) \oplus N$
9      Update the nonlinear driver by taking the following three steps sequentially:
           $N \leftarrow \mathcal{R}(N) \oplus L$
           $L \leftarrow H$
           $H \leftarrow F$
10     Update the FSM by taking the following three steps sequentially:
           $S_2 \leftarrow \mathcal{R}(S_1) \oplus S_2$
           $S_1 \leftarrow \mathcal{R}(S_0) \oplus S_1$
           $S_0 \leftarrow F \oplus G \oplus S_0$
       return $(Z, \boldsymbol{S})$
11 procedure scInit($\boldsymbol{S}, K$)
12     Split the 256-bit key $K$ as $K = (K_h, K_\ell)$
13     for $i = 0, \ldots 12$ do
14         $(Z, \boldsymbol{S}) \leftarrow$ scOut($\boldsymbol{S}$)
15         Update the $H$ and $N$ states of $\boldsymbol{S}$ as $H \leftarrow H \oplus Z$ and $N \leftarrow N \oplus Z$
16     Update $H$ of $\boldsymbol{S}$ as $H \leftarrow H \oplus K_h$
17     Update $S_0$ of $\boldsymbol{S}$ as $S_0 \leftarrow S_0 \oplus K_\ell$
18     return $\boldsymbol{S}$
19 procedure COPY($\boldsymbol{S}$)
20     $\boldsymbol{E} = (E_0, E_1, E_2, E_3) \leftarrow (N, S_0, S_1, S_2)$
21     return $\boldsymbol{E}$
22 procedure updE($\boldsymbol{E}, D$)
23     Compute the intermediate value $U \leftarrow \mathcal{R}(E_3) \oplus E_0$
24     Update the MacState and absorb the data by taking the following four steps sequentially:
           $E_3 \leftarrow \mathcal{R}(E_2) \oplus E_3;$
           $E_2 \leftarrow \mathcal{R}(E_1) \oplus E_2;$
           $E_1 \leftarrow \mathcal{R}(E_0) \oplus E_1;$
           $E_0 \leftarrow U \oplus D.$
       return $\boldsymbol{E}$
25 procedure Feedback($\boldsymbol{E}, \boldsymbol{S}$)
26     $\boldsymbol{S} = (L, H, N, S_0, S_1, S_2) \leftarrow (L, H, E_0, E_1, E_2, E_3)$
27     return $\boldsymbol{S}$
```

2) As an AEAD scheme: The encryption/decryption process of `LOL2.0-Mini` works as Algorithm 2 further combined with the *MacState*, which employs parameter No.2 in Table IV. The concrete definitions of the subroutines `COPY`, `updE` and `Feedback` are given in Algorithm 5. As can be seen, the *MacState* consists of four 128-bit state blocks, which is initialized by $\boldsymbol{E} \leftarrow \texttt{COPY}(\boldsymbol{S})$. Subsequently, the *MacState* absorbs the 128-bit data block and updates as `updE`. The lengths $|AD|$ and $|M|$ are encoded as 64-bit words making the final block $\Theta = |AD| \| |M|$ of the length 128 bits. Thereafter, the *MacState* is fed back to the state $\boldsymbol{S}$ as `Feedback`.

`LOL2.0-Mini` supports up to 128-bit variable-length tags ($\tau \leq 128$), and provides 256-bit security against key-recovery attacks and 128-bit security against forgery attacks in the nonce-respecting setting. It limits the keystream length to at most 2^{64} for a single key-IV pair, and each key can be used with a maximum of 2^{64} different IVs. The number of different messages produced for a fixed key is no more than $2^{\tau/2}$ ($\leq 2^{64}$), while the associated data length is no longer than 2^{64}.

C. `LOL2.0-Double` Cipher

1) As a stream cipher: `LOL2.0-Double` adopts the *parallel-dual mode* of the `LOL2.0` framework. It supports a 256-bit key denoted as $K = (K_h, K_\ell)$ and a 256-bit IV denoted as $IV = (IV_h, IV_\ell)$. The internal state of `LOL2.0-Double` consists of ten 128-bit blocks: 4 for the LFSR denoted as (H, L) where $H = (H_1, H_0)$ and $L = (L_1, L_0)$, 2 for two NFSRs denoted as N_0 and N_1, 4 for two FSMs denoted as (S_0, S_1) and (S_2, S_3) forming a circular. As can be seen, `LOL2.0-Double` is expanded by two underlying single mode primitives, denoted as (H_0, L_0, N_0, S_0, S_1), and (H_1, L_1, N_1, S_2, S_3) respectively; the 256-bit output keystream block of `LOL2.0-Double` is the concatenation of the two 128-bit keystream blocks of its underlying primitives in the reversed order, denoted as $Z = (Z_0, Z_1)$. The internal state $\boldsymbol{S}$ of `LOL2.0-Double` is defined as

$$\boldsymbol{S} = (L_0, L_1, H_0, H_1, N_0, N_1, S_0, \ldots, S_3) \tag{25}$$

The LFSR of `LOL2.0-Double` cipher consist of 32 16-bit cells denoted as $a_{31}, \ldots, a_1, a_0$ (equivalent to the $\ell = 32$, $m = 16$ situation), which are stored in the two 256-bit registers (H, L) as

$$H = (a_{31}, \ldots, a_{17}, a_{16}), L = (a_{15}, \ldots, a_1, a_0). \tag{26}$$

Same with `LOL2.0-Mini`, the updating function $f(H, L) = \lambda(H) \oplus \sigma(L)$, of the same form as Equation (5), has two options denoted as `LFSR1` and `LFSR2` respectively, sharing the same σ definition as Equation (27).

$$(x_{15}, \ldots, x_0) \xrightarrow{\sigma} (x_{11}, x_6, x_{15}, x_{14}, x_2, x_8, x_0, x_9, x_4, x_7, x_{10}, x_{13}, x_1, x_5, x_{12}, x_3). \tag{27}$$

They only differs in the λ settings: The λ of `LFSR1` is identical to that used in `LOL-Double` in [6]. To ensure the completeness of the design, we provide a detailed definition of λ of `LFSR1` in Section A. For `LFSR2`, the λ employs parameter No.1 in Table II, i.e., defined as Equation (23)

$$\lambda(x) = \texttt{blend}_{16}(x|_{32}^{\lll 5}, x|_{16}^{\ggg 6}, \texttt{0x5619}) \tag{28}$$

which is equivalent to

$$\begin{aligned}\lambda = (&h_7 \lll 5 \oplus h_6 \ggg 11, h_6 \ggg 6, h_5 \lll 5 \oplus h_4 \ggg 11, h_4 \ggg 6, h_3 \lll 5 \oplus h_2 \ggg 11, h_2 \ggg 6, h_1 \ggg 6, h_0 \lll 5,\\ &h_7 \lll 5 \oplus h_6 \ggg 11, h_6 \lll 5, h_5 \lll 5 \oplus h_4 \ggg 11, h_4 \ggg 6, h_3 \ggg 6, h_2 \lll 5, h_1 \lll 5 \oplus h_0 \ggg 11, h_0 \ggg 6)\end{aligned} \tag{29}$$

As can be seen, the definition of `LFSR2` in `LOL2.0-Mini` is still part of that in `LOL2.0-Double`.

The encryption/decryption process of `LOL2.0-Double` has been summarized in Algorithm 1. The concrete definitions of the subroutines `Load`, `scInit` and `scOut` are given in Algorithm 6. As can be seen, the secret key and public IV are loaded to the FSM registers while setting all nonlinear driver bits to 0 or other constants. Then, `LOL2.0-Double` runs 12 initialization rounds when the 256-bit keystream block Z's are not output but instead fed back to the nonlinear driver for thorough diffusion. Subsequently, `LOL2.0-Double` adopts the FP-(1) mode [18] by XORing the key to the internal state at the end of `scInit` making it irreversible. Thereafter, `LOL2.0-Double` starts the generation of the keystream. It also upper bounds the keystream length to 2^{64} for a single key-IV pair, and each key may be used with a maximum of 2^{64} different IVs. The initialization and keystream generation phases of `LOL2.0-Double` are illustrated as Figure 7 and Figure 8 respectively.

2) As an AEAD scheme: The encryption/decryption process of `LOL2.0-Double` works as Algorithm 2 further combined with the *MacState*, which employs parameter No.10 in Table IV. The concrete definitions of the subroutines `COPY`, `updE` and `Feedback` are given in Algorithm 6. As can be seen, the *MacState* consists of six 128-bit state blocks, which is initialized by $\boldsymbol{E} \leftarrow \texttt{COPY}(\boldsymbol{S})$. Subsequently, the *MacState* absorbs the 256-bit data block and updates as `updE`. The lengths $|AD|$ and $|M|$ are encoded as 128-bit state making the final block $\Theta = |AD| \| |M|$ of the length 256 bits. Thereafter, the *MacState* is fed back to the state $\boldsymbol{S}$ as `Feedback`.

Algorithm 6: The subroutines `LOL2.0-Double` where the secret key K and pubic vector IV are both of 256 bits. The internal state $\boldsymbol{S}$ is denoted as $\boldsymbol{S} = (L_0, L_1, H_0, H_1, N_0, N_1, S_0, \ldots, S_3)$, and the *MacState* $\boldsymbol{E}$ is denoted as $\boldsymbol{E} = (E_0, E_1, E_2, E_3, E_4, E_5)$. We further define the $\boldsymbol{S}$-entry combinations $H = (H_1, H_0)$, $L = (L_1, L_0)$ and $N = (N_1, N_0)$.

```
1  procedure Load($K, IV$)
2      Split the 256-bit key $K$ as $K = (K_h, K_\ell)$ and $IV$ as $IV = (IV_h, IV_\ell)$
3      Initialize $(L, H, N) \leftarrow (0, 0, 0)$, $(S_2, S_0) \leftarrow (K_h, K_\ell)$ and $(S_3, S_1) \leftarrow (IV_h, IV_\ell)$
4      return $\boldsymbol{S}$
5  procedure scOut($\boldsymbol{S}$)
6      Compute the intermediate values $G_0 \leftarrow \mathcal{R}(S_1) \oplus N_0$ and $G_1 \leftarrow \mathcal{R}(S_3) \oplus N_1$
7      Compute the LFSR feedback $F = (F_1, F_0) \leftarrow f(H, L)$ where $F_1, F_2 \in \mathbb{F}_2^{128}$
8      Compute the keystream block $Z = (Z_0, Z_1) \leftarrow (\mathcal{R}(G_0) \oplus N_0, \mathcal{R}(G_1) \oplus N_1)$
9      Update the nonlinear driver by taking the following three steps sequentially:
           $N = (N_1, N_0) \leftarrow (\mathcal{R}(N_1) \oplus L_1, \mathcal{R}(N_0) \oplus L_0)$
           $L \leftarrow H$
           $H \leftarrow F$
10     Update the FSM by taking the following two steps sequentially:
           $(S_3, S_1) \leftarrow (\mathcal{R}(S_2) \oplus S_3, \mathcal{R}(S_0) \oplus S_1)$
           $(S_2, S_0) \leftarrow (F_1 \oplus G_0 \oplus S_2, F_0 \oplus G_1 \oplus S_0)$
       return $(Z, \boldsymbol{S})$
11 procedure scInit($\boldsymbol{S}, K$)
12     Split the 256-bit key $K$ as $K = (K_h, K_\ell)$
13     for $i = 0, \ldots 12$ do
14         $(Z, \boldsymbol{S}) \leftarrow$ scOut($\boldsymbol{S}$)
15         Update the $H$ and $N$ states of $\boldsymbol{S}$ as $H \leftarrow H \oplus Z = (H_1 \oplus Z_0, H_0 \oplus Z_1)$ and
             $N \leftarrow N \oplus Z = (N_1 \oplus Z_0, N_0 \oplus Z_1)$
16     Update $H$ of $\boldsymbol{S}$ as $H \leftarrow H \oplus K$
17     return $\boldsymbol{S}$
18 procedure COPY($\boldsymbol{S}$)
19     $\boldsymbol{E} = (E_0, E_1, E_2, E_3, E_4, E_5) \leftarrow (N_0, N_1, S_0, S_1, S_2, S_3)$
20     return $\boldsymbol{E}$
21 procedure updE($\boldsymbol{E}, D$)
22     Compute the intermediate value $U \leftarrow \mathcal{R}(E_5) \oplus E_0$
23     Update the MacState and absorb the data by taking the following six steps sequentially:
           $E_5 \leftarrow \mathcal{R}(E_4) \oplus D_1;$
           $E_4 \leftarrow \mathcal{R}(E_3) \oplus D_1;$
           $E_3 \leftarrow \mathcal{R}(E_2) \oplus E_3;$
           $E_2 \leftarrow \mathcal{R}(E_1) \oplus D_0;$
           $E_1 \leftarrow \mathcal{R}(E_0) \oplus D_0;$
           $E_0 \leftarrow U.$
       return $\boldsymbol{E}$
24 procedure Feedback($\boldsymbol{E}, \boldsymbol{S}$)
25     $\boldsymbol{S} = (L_0, L_1, H_0, H_1, N_0, N_1, S_0, \ldots, S_3) \leftarrow (L_0, L_1, H_0, H_1, E_0, \ldots, E_5)$
26     return $\boldsymbol{S}$
```

LOL2.0-Double-SCMAC also supports up to 128-bit variable-length tags ($\tau \leq 128$), and provides 256-bit security against key-recovery attacks and 128-bit security against forgery attacks in the nonce-respecting setting. It limits the keystream length to at most 2^{64} for a single key-IV pair, and each key can be used with a maximum of 2^{64} different IVs. The number of different messages for a fixed key is limited to no more than $2^{\tau/2}$ ($\leq 2^{64}$), while the associated data length for a fixed key is up to 2^{64}.

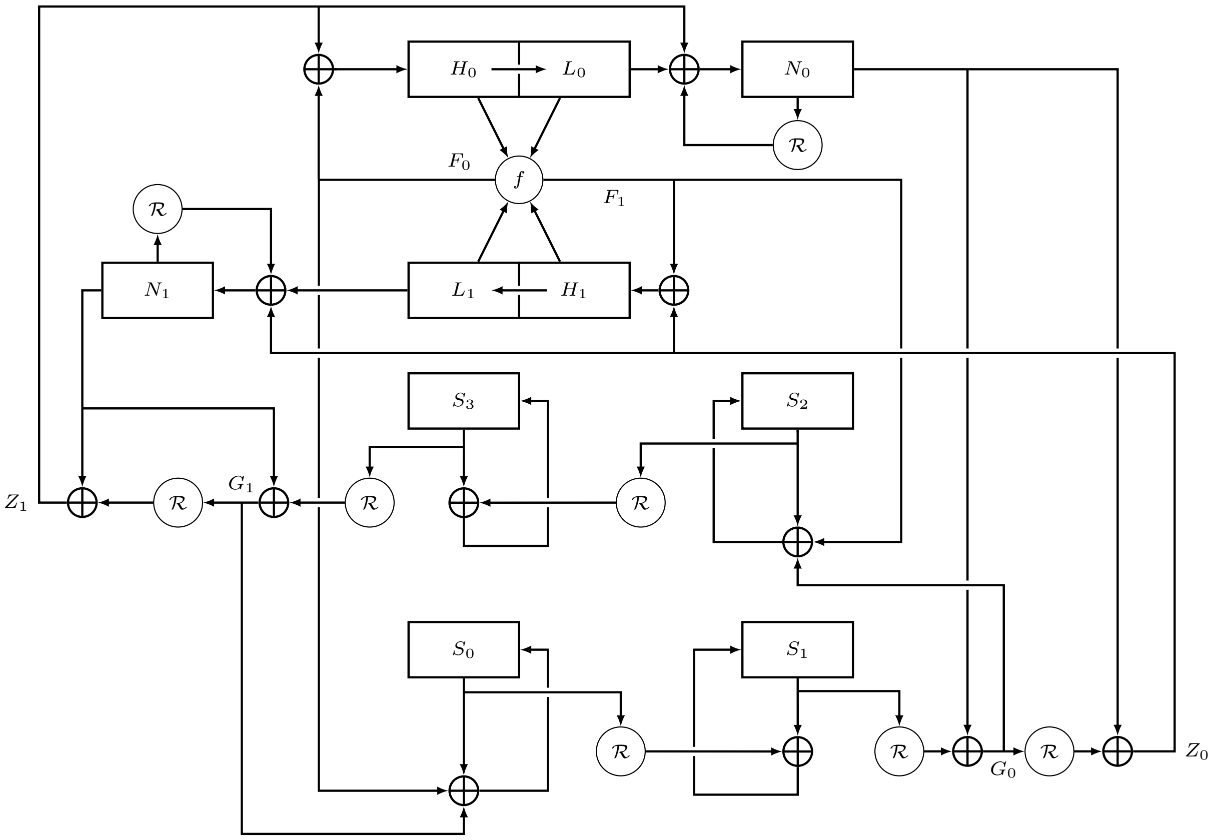

Fig. 7: The schematic of LOL2.0-Double cipher in initialization phase

VI. Security Analysis of LOL2.0 ciphers

A. The Confidentiality of LOL2.0 ciphers

As stream ciphers, the resistance of LOL2.0-Mini and LOL2.0-Double against common cryptanalysis methods is evaluated so as to prove their 256-bit security.

1) Maximum period proof of the LFSR: For LOL2.0 framework, the maximum period proof method of LFSRs has been demonstrated in Section IV-B.

For LOL2.0-Mini, the characteristic polynomial of LFSR2 is primitives of degree 256 as follows; therefore, the corresponding LFSR has the maximum periods of $2^{256} - 1$.

LFSR2: $\eta(x) = x^{256} + x^{230} + x^{217} + x^{214} + x^{211} + x^{204} + x^{202} + x^{191} + x^{188} + x^{184} + x^{182} + x^{180} + x^{179} + x^{175} + x^{174} + x^{172} + x^{171} + x^{170} + x^{167} + x^{164} + x^{163} + x^{162} + x^{161} + x^{160} + x^{159} + x^{154} + x^{152} + x^{150} + x^{149} + x^{148} + x^{144} + x^{137} + x^{135} + x^{132} + x^{130} + x^{127} + x^{126} + x^{125} + x^{124} + x^{122} + x^{116} + x^{115} + x^{114} + x^{106} + x^{104} + x^{103} + x^{95} + x^{94} + x^{90} + x^{89} + x^{88} + x^{80} + x^{78} + x^{71} + x^{68} + x^{57} + 1.$

For LOL2.0-Double, the characteristic polynomial of LFSR2 is primitives of degree 512 as follows; therefore, the corresponding LFSR has the maximum periods of $2^{512} - 1$.

LFSR2: $\eta(x) = x^{512} + x^{484} + x^{479} + x^{473} + x^{469} + x^{466} + x^{465} + x^{464} + x^{463} + x^{460} + x^{458} + x^{456} + x^{455} + x^{452} + x^{450} + x^{448} + x^{447} + x^{446} + x^{445} + x^{442} + x^{441} + x^{438} + x^{437} + x^{436} + x^{435} + x^{431} + x^{429} + x^{428} + x^{427} + x^{426} + x^{425} + x^{423} + x^{422} + x^{418} + x^{417} + x^{416} + x^{414} + x^{413} + x^{407} + x^{406} + x^{405} + x^{401} + x^{400} + x^{397} + x^{395} + x^{392} + x^{391} + x^{389} + x^{388} + x^{387} + x^{382} + x^{381} + x^{380} + x^{379} + x^{375} + x^{374} + x^{373} + x^{372} + x^{370} + x^{367} + x^{366} + x^{363} + x^{362} + x^{360} + x^{358} + x^{357} + x^{355} + x^{352} + x^{351} + x^{348} + x^{347} + x^{344} + x^{342} + x^{335} + x^{334} + x^{333} + x^{331} + x^{330} + x^{327} + x^{326} + x^{325} + x^{322} + x^{321} + x^{320} + x^{318} + x^{317} + x^{311} + x^{310} + x^{309} + x^{306} + x^{305} + x^{302} + x^{301} + x^{298} + x^{296} + x^{295} + x^{293} + x^{291} + x^{288} + x^{286} + x^{284} + x^{283} + x^{282} + x^{281} + x^{278} + x^{277} + x^{272} + x^{265} + x^{264} + x^{262} + x^{260} + x^{259} + x^{258} + x^{253} + x^{252} + x^{250} + x^{249} + x^{247} + x^{244} + x^{243} + x^{240} + x^{237} + x^{232} + x^{230} + x^{229} + x^{227} + x^{226} + x^{225} + x^{224} + x^{221} + x^{219} + x^{218} + x^{217} + x^{216} + x^{215} + x^{213} + x^{209} + x^{207} + x^{205} + x^{200} + x^{195} + x^{193} + x^{191} + x^{184} + x^{183} + x^{182} + x^{181} + x^{178} + x^{177} + x^{176} + x^{175} + x^{171} +$

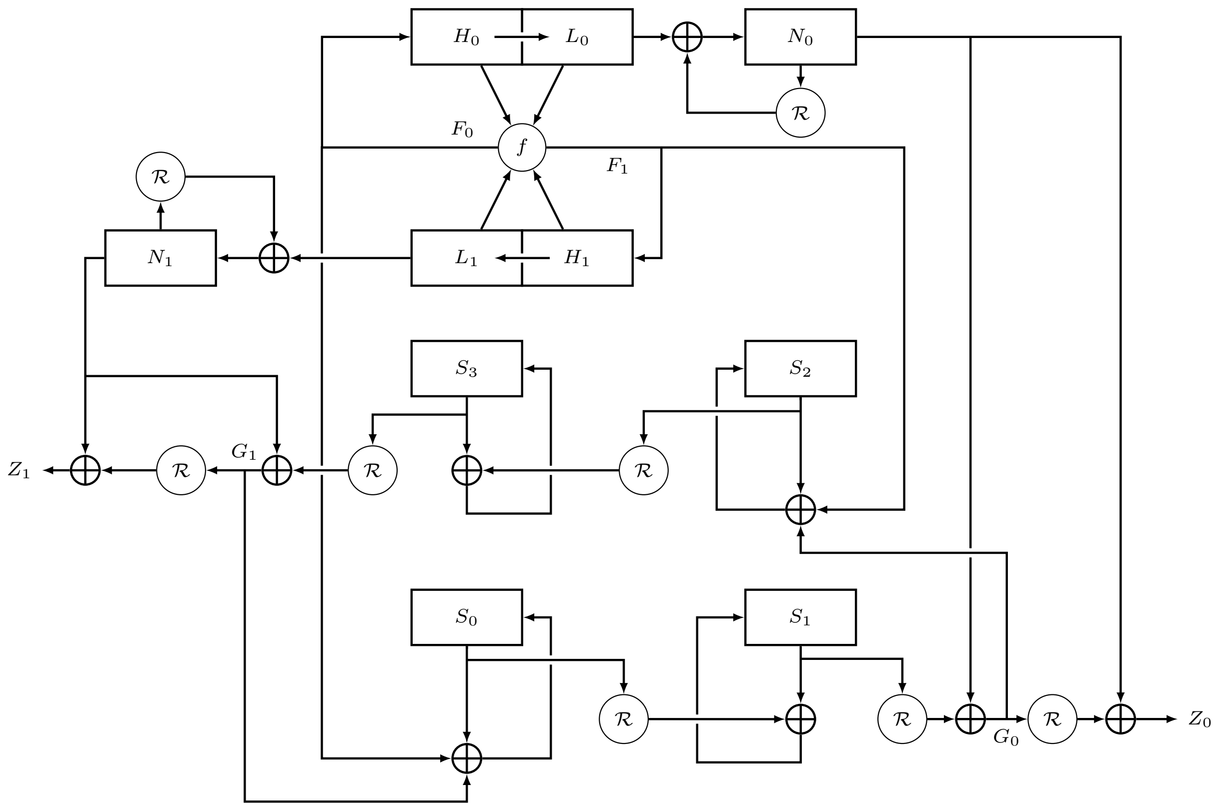

Fig. 8: The schematic of `LOL2.0-Double` cipher in keystream generation phase

$$x^{168}+x^{166}+x^{163}+x^{162}+x^{161}+x^{160}+x^{159}+x^{156}+x^{155}+x^{151}+x^{150}+x^{149}+x^{148}+x^{146}+x^{143}+x^{140}+x^{138}+x^{136}+x^{135}+x^{133}+x^{132}+x^{131}+$$
$$x^{130}+x^{129}+x^{128}+x^{126}+x^{125}+x^{123}+x^{122}+x^{121}+x^{120}+x^{119}+x^{118}+x^{115}+x^{114}+x^{113}+x^{112}+x^{110}+x^{107}+x^{106}+x^{104}+x^{102}+x^{101}+x^{100}+$$
$$x^{99}+x^{94}+x^{93}+x^{90}+x^{89}+x^{88}+x^{86}+x^{83}+x^{78}+x^{74}+x^{69}+x^{68}+x^{66}+x^{65}+x^{63}+x^{61}+x^{60}+x^{55}+x^{51}+x^{49}+x^{46}+x^{43}+x^{42}+x^{31}+x^{26}+x^{20}+1.$$

2) Full diffusion round analysis: The number of full diffusion rounds in the initialization of a stream cipher is the minimum number of initialization iterations required for each state bit of the input to influence all state bits.

For `LOL2.0` framework, the number of full diffusion rounds is influenced by structural parameters and component selection. Here, we propose loading the key/IV into the rapidly diffusing FSM instead of the conventional LFSR.

For `LOL2.0-Mini` and `LOL2.0-Double`, achieving full diffusion necessitates 5 initialization rounds (with either `LFSR1` or `LFSR2`). Table V shows the full diffusion round count for each state block.

TABLE V: Full diffusion round for `LOL2.0-Mini` and `LOL2.0-Double`

`LOL2.0-Mini`						`LOL2.0-Double`									
L	H	N	S_0	S_1	S_2	L_0	L_1	H_0	H_1	N_0	N_1	S_0	S_1	S_2	S_3
5	5	4	4	4	3	5	5	5	5	4	4	4	3	4	3

3) Differential attack on intialization: The differential attacks are considered under the related-key chosen-IV model, where differences can be injected in both key and IV bits.

For LOL2.0 framework, the resistance against differential analysis depends on factors such as the DDT of S-box and branch number of linear layer in $\mathcal{R}$ function, as well as the structural parameters. With truncated differential propagation rules and MILP-based automatic search techniques, we can determine an upper bound for the probability of differential characteristics with the minimum number of active S-boxes: XOR and SPN $\mathcal{R}$ functions can be directly modeled; LFSR can be modeled by H-representation technique detailed in [21]; some inter-round relationships in the form of XORing $\mathcal{R}$ functions should be considered by combining their linear layer operations.

For `LOL2.0-Mini` and `LOL2.0-Double`, it is sufficient to have at least 43 active S-boxes to ensure security against differential attacks, since the maximal differential probability of AES's S-box is 2^{-6}. The results are shown in Table VI, which indicate that there are 45 active S-boxes over 6 rounds for `LOL2.0-Mini` and 51 active S-boxes over 5 rounds for `LOL2.0-Double` in the initialization phase in related-key setting. Thus the secure bound for `LOL2.0-Mini` (`LOL2.0-Double`) against differential attacks is 6 (5).

r	Minimum number of active S-boxes	
	`LOL2.0-Mini` (with `LFSR1` or `LFSR2`)	`LOL2.0-Double` (with `LFSR1` or `LFSR2`)
2	6	6
3	26	27
4	33	39
5	41	51
6	45	-

TABLE VI: The active S-box lower bounds for r initialization rounds.

Since the update function in `LOL2.0-Mini` and `LOL2.0-Double` is not a permutation, state collisions may occur, meaning that the number of active S-boxes does not continue to increase with additional initialization rounds. However, the results presented in Table VI indicate that the complexity of identifying a state collision during the initialization phase remains at or above the security threshold.

4) Integral attack: For `LOL2.0` framework, its blockwise structure determines that the keystream can be expressed in a simplified form based on the IV. Then if there is no key-independent integral distinguisher for n iterated $\mathcal{R}$ function calls, the `LOL2.0` cipher can be proved secure against integral attacks as long as the first keystream output after initialization has undergone n $\mathcal{R}$ function iterations.

For `LOL2.0-Mini` and `LOL2.0-Double`, since they both have 12 initialization rounds, we denote the not-output keystream blocks as $Z^{-12}, \ldots, Z^{-1}$. For `LOL2.0-Mini`, IV needs to go through 5 AES rounds for Z^{-9}, while for `LOL2.0-Double`, IV_h and IV_ℓ need to go through 5 AES rounds for the keystream block $Z^{-9} = (Z_0^{-9}, Z_1^{-9})$. According to [22], there are no integral distinguishers on 5 or more AES rounds [22]. Our evaluations demonstrate that it is impossible to prevent IV from going through ≤ 5 AES rounds for Z^{-9}, making it impossible to conduct integral attacks.

Additionally, we establish bit-based division MILP models for `LOL2.0-Mini` and `LOL2.0-Double`, and examine the algebraic degree of each round output with respect to IV, as well as the algebraic degree of the key given a specific IV. The detailed findings are summarized below: For `LOL2.0-Mini` (`LOL2.0-Double`), each bit of Z^{-9} includes a 128-(256-)th degree monomial in terms of IV, along with 128(256) instances of 127-(255-)th degree monomials in terms of IV, which all achieve an algebraic degree of 256(256) with respect to the key. Since our analysis employs two-subset models, the results only indicate that there is a division character for the detected monomials, but cannot guarantee that the monomials definitely exist.

5) Slide attacks: Slide attacks aim to form the same state at shifted instants for different key/IV pairs, which can be detected by observing similar keystreams and recover certain key information.

For `LOL2.0` framework, it can be secure against slide attacks by adding a feeding-key-forward operation in initialization, or using distinct update functions for initialization and key generation.

For `LOL2.0-Mini` and `LOL2.0-Double`, such sliding properties would be difficult to form also due to their large state updates at each step additionally.

6) Correlation attack: FCA leverages strong linear correlations between keystream and LFSR bits for internal state recoveries.

For `LOL2.0` framework, it employs a combination of nonlinear drivers and FSMs with memories to avoid direct LFSR-FSM connections in the SNOW stream cipher family, thereby eliminating high linear correlations utilized in FCAs. The large memory of FSM prevents FCAs on nonlinear-driver based Grain-like stream ciphers, while the use of nonlinear drivers resists linear distinguishing attacks in `AEGIS`.

Similar to differential attacks, resistance against correlation analysis depends on the LAT of S-boxes, branch number of linear layers in $\mathcal{R}$ functions, and structural parameters. With truncated linear propagation rules and MILP-based automatic search techniques, we can upper bound the linear correlations with minimum active S-boxes: XOR and SPN $\mathcal{R}$ functions can be directly modeled; LFSRs can be modeled using H-representation technique from [21]. For correlation `Cor`, the complexity of correlation attacks is evaluated as $\mathcal{O}(\texttt{Cor}^{-2})$. The minimum consecutive time instants required to construct an effective linear approximation between keystream and LFSR bits can be determined by

$$\frac{\text{The number of nonlinear state units}}{\text{The number of keystream units per time instant}}.$$

For `LOL2.0-Mini` and `LOL2.0-Double`, it is sufficient to guarantee the security against correlation attacks if there are at least 43 active S-boxes, since the maximal correlation `Cor` of AES's S-box is 2^{-3},. It reveals that at least 4 (3) consecutive time instants are necessary for a correlation attack on `LOL2.0-Mini` (`LOL2.0-Double`). The result is shown in Table VII, which shows that there are at least 48 (52) active S-boxes for `LOL2.0-Mini` (`LOL2.0-Double`) for a linear approximation. Therefore, neither effective linear distinguishers nor correlation attacks can be applied to `LOL2.0-Mini` and `LOL2.0-Double`.

r	Minimum number of active S-boxes	
	`LOL2.0-Mini` (with `LFSR1` or `LFSR2`)	`LOL2.0-Double` (with `LFSR1` or `LFSR2`)
3	-	62
4	48	52
5	48	52
6	48	-

TABLE VII: The active S-box lower bounds for r encryption rounds.

Furthermore, our analysis has identified a linear path for `LOL2.0-Mini` wherein the number of active S-boxes is precisely 59, as well as a linear path for `LOL2.0-Double` wherein the number of active S-boxes is precisely 80.

7) Guess-and-determine attack: Guess-and-determine attacks act as a common tool to achieve state recovery during the keystream generation phase.

For `LOL2.0` framework, attackers can determine the minimum number of keystream blocks required to form a guess-and-determine attack by which the whole internal state is involved. Due to the full diffusion property of $\mathcal{R}$ function, attackers need to guess some complete state units. Combined with the large state unit of `LOL2.0` stream ciphers, the complexity of the byte-based guess-and-determine attack can be bounded.

For `LOL2.0-Mini` (resp. `LOL2.0-Double`), a keystream block at each step only involves 2 (resp. 4) state units. Since the internal state consists of 6 (resp. 10) units, attackers must consider at least 3 (resp. 3) consecutive steps to involve the information about the whole internal state. We first consider the attack based on 128-bit units. It is necessary to guess N^t,N^{t+1},N^{t+2} according to Z^t,Z^{t+1},Z^{t+2} with a time complexity of 2^{384} on `LOL2.0-Mini` and another 3 keystream blocks are needed for filtering. For `LOL2.0-Double`, it is necessary to guess N_0^t, N_0^{t+1}, N_0^{t+2}, N_1^t according to Z_0^t, Z_1^t, Z_0^{t+1}, Z_1^{t+1}, Z_0^{t+2}, Z_1^{t+2} with a time complexity of 2^{512} and another 4 keystream blocks are needed for filtering.

Then we consider the guess-and-determine attack based on bytes. For `LOL2.0-Mini`, the formation of a guess-and-determine attack requires 3 consecutive keystream blocks:

$$\begin{aligned}
Z^t &= \mathcal{R}(\mathcal{R}(S_2^t) \oplus N^t) \oplus N^t,\\
Z^{t+1} &= \mathcal{R}(\mathcal{R}(\mathcal{R}(S_1^t) \oplus S_2^t) \oplus \mathcal{R}(N^t) \oplus L^t) \oplus \mathcal{R}(N^t) \oplus L^t\\
Z^{t+2} &= \mathcal{R}(\mathcal{R}(\mathcal{R}(\mathcal{R}(S_0^t) \oplus S_1^t) \oplus \mathcal{R}(S_1^t) \oplus S_2^t) \oplus \mathcal{R}(\mathcal{R}(N^t) \oplus L^t) \oplus H^t) \oplus \mathcal{R}(\mathcal{R}(N^t) \oplus L^t) \oplus H^t
\end{aligned}$$

Once two or more AES rounds are involved, it implies that the attackers need to guess at least a complete 128-bit unit due to the full diffusion property of AES round function. Therefore, we believe that the complexity of a guess-and-determine attack on `LOL2.0-Mini` cannot be lower than 2^{256}. The analysis and security claim hold similar for `LOL2.0-Double`.

8) Time/Memory/Data tradeoff attacks: The time memory data tradeoff attacks have two phases: during the preprocessing phase, a mapping table from different secret keys or internal states to keystreams is computed and stored with time complexity P and memory M; in the real-time phase, attackers intercept D keystreams and search for matches in the table with time complexity T, aiming to recover the corresponding input. To balance the parameters, the most popular tradeoffs are Babbage-Golic's [23], [24] with $TM = N, P = M, T \leq D$ and Biryukov-Shamir's [25] with $MT^2D^2 = N^2, P = N/D, T \leq D^2$, where N is the input space.

To reconstruct the internal state, `LOL2.0-Mini` and `LOL2.0-Double` provide natural immunity due to their internal state sizes being more than twice the secret key size. Additionally, we use FP(1)-mode in the initialization phase to prevent attacks from benefiting by reconstructing the internal state in order to recover the key directly. To recover the secret key directly, there is a trivial (yet impractical) attack on `LOL2.0-Mini` since its IV size is smaller than the key size. In contrast, this vulnerability is mitigated by `LOL2.0-Double`.

9) Algebraic attacks: For `LOL2.0` framework, an algebraic representation of the update function can be given as follows: the S-boxes in the SPN functions are expressed as a set of nonlinear boolean functions by computing the annihilating implicit equations of the S-box, while other operations are expressed as a set of linear boolean functions through redefining some variables. By observing how the number of equations and variables grow over time instants and estimating their algebraic degree, attackers can determine whether an `LOL2.0` cipher's algebraic representation is solvable.

For `LOL2.0-Mini` and `LOL2.0-Double`, we construct a system of equations and compute the total number of terms. It is well-known that each AES S-box can be completely defined by a system of 23 linearly independent bi-affine quadratic equations involving input and output bits in 80 terms, which includes 64 quadratic terms and 16 linear terms [26]. We find that the number of terms and equations expands significantly with rounds. Moreover, regardless if it is internal states or secret keys being recovered, solving these equations using methods like linearization or XL algorithm has complexity higher than 2^{256}. Therefore, `LOL2.0-Mini` and `LOL2.0-Double` are immune to algebraic attacks.

B. The integrity of `LOL2.0` *ciphers*

As AEAD schemes, we evaluate 128-bit security of `LOL2.0-Mini` and `LOL2.0-Double` against forgery attacks in the nonce-respecting setting.

1) MacState *collision during the encryption phase:* As mentioned in Section V-A, our security measure relies on the $\#ASB$ value, which gives an upper bound on the success probability of a differential attack that may lead to collisions in $\boldsymbol{E}$. We transform the search of the differential characteristic of the for Equation (19) into MILP problems. Since we use $\mathcal{R}$ defined as the AES round function, we only require the differential characteristic to have at least 22 active S-boxes ($\#ASB \geq 22$) to ensure the 128-bit security against forgery attacks. The evaluations to $\#ASB$ are shown in Table VIII. The number of active S-boxes tends to stabilize as the number of rounds increases.

r	Minimum number of # ASB	
	`LOL2.0-Mini` (with `LFSR1` or `LFSR2`)	`LOL2.0-Double` (with `LFSR1` or `LFSR2`)
3	-	31
4	-	27
5	35	27
6	35	27
7	35	27
8	35	27

TABLE VIII: The active S-box lower bounds for *MacState* collision.

2) Tag collision in the tag generation phase: The differential resistance of `LOL2.0-Mini` and `LOL2.0-Double` in **Finalization** process should be reevaluated. In the standard initialization, non-zero differences can only be injected to the 3 (`LOL2.0-Mini`) or 4 (`LOL2.0-Double`) 128-bit registers that load the key and IV bits. In the **Finalization** process, the non-zero differences can appear in 4 (`LOL2.0-Mini`) or 6 (`LOL2.0-Double`) 128-bit registers after the `Feedback` operation. We construct an MILP model describing the truncated differential characteristic as follows:

$$\boldsymbol{0} \neq \Delta \boldsymbol{S}^0 \rightarrow \Delta \boldsymbol{S}^r. \tag{30}$$

By solving this MILP model $\mathcal{M}$, which captures the truncated differential characteristic mentioned in Equation (30), we are able to identify the minimum number of active Sboxes given in Table IX.

r	Minimum number of active S-boxes	
	`LOL2.0-Mini` (with `LFSR1` or `LFSR2`)	`LOL2.0-Double` (with `LFSR1` or `LFSR2`)
2	5	6
3	12	17
4	33	28
5	41	28

TABLE IX: The active S-box lower bounds for tag collision.

Therefore, we can conclude that the maximum tag length is $\tau_{\max} = 210$ for `LOL2.0-Mini`, and $\tau_{\max} = 162$ for `LOL2.0-Double` according to Equation (4). However, these estimates do not account for additional constant factors of improvements and optimizations, e.g. using clustering effect. This is why we reduce our security claims. Consequently, `LOL2.0-Mini` and `LOL2.0-Double` can provide 128-bit security against the forgery attack.

3) General state-recovery attack [14]: As discussed in Section III-C, `SCMAC` design framework is capable of directly resisting this attack, which was targeted at Rocca [8], while also supporting variable-length tags.

4) Committing attacks: Committing security is the notion formalized for AEAD in [27] under the name of robust AE. It is utilized in many real world applications [28], [29], [30], [31]. Let $\mathcal{H}$ be an AEAD scheme. Its encryption process, denoted as $\mathcal{H}$.`enc` takes (K, IV, AD, M) as its input (each letter corresponds to the secret key, the public nonce, the associated data and the message respectively) and output (C, T) (C is the ciphertext and T is the tag for authentication). On receiving (C, T), the decryption process $\mathcal{H}$.`dec` will do the authentication and decryption simultaneously by taking (K, IV, AD, C, T) as inputs: if (C, T) is legal, the $\mathcal{H}$.`dec` will return the corresponding message

M; otherwise, it will output the specific symbol $\perp$ indicating the failure of authentication and decryption. Committing security considers the adversary with full control over (K, IV, M, T). The goal of the adversary is to find distinct (K, IV, AD, M) and (K', IV', AD', M') making

$$\mathcal{H}.\texttt{enc}(K, IV, AD, M) = \mathcal{H}.\texttt{enc}(K', IV', AD', M') \tag{31}$$

The levels of the committing security is specified as CMT-ℓ where $\ell = 1, 2, 3, 4$ specifies the distinctness condition over the first ℓ input elements of Equation (31): specifically, the CMT-4 notion requires $(K, IV, AD, M) \neq (K', IV', AD', M')$; CMT-3 requires $(K, IV, AD) \neq (K', IV', AD')$; CMT-2 requires $(K, IV) \neq (K', IV')$; CMT-1 requires $K \neq K'$. The FROB security imposes even stricter constraints with $K \neq K'$ and $IV = IV'$.

We claim that `LOL2.0-Mini` and `LOL2.0-Double` AEAD schemes reach CMT-1, CMT-2 and FROB security. For $|T| = 128$, the generic committing security bounds for `LOL2.0-Mini` and `LOL2.0-Double` are both 2^{128}. To make $(C, T) = (C', T')$, the attacker can make $C = C'$ by $\Delta M = \Delta Z$. To make $T = T'$, in this context, the construction of the model differs slightly. After the initialization process `scInit`, which thoroughly mixes Δk and ΔIV, the *ScState* difference is known but no longer controllable. Subsequently, after keystream encryption, the non-feedback *ScState* differences ΔH and ΔL are also uncontrollable and cannot be considered as free difference variables. In contrast, the *ScState* differences fed back by *MacState* can be treated as free variables since ΔAD remains controllable. The schematic is presented in Figure 9. The attack complexity consists of two parts: first, the probability that $(\Delta H, \Delta L)$ reaches a certain value

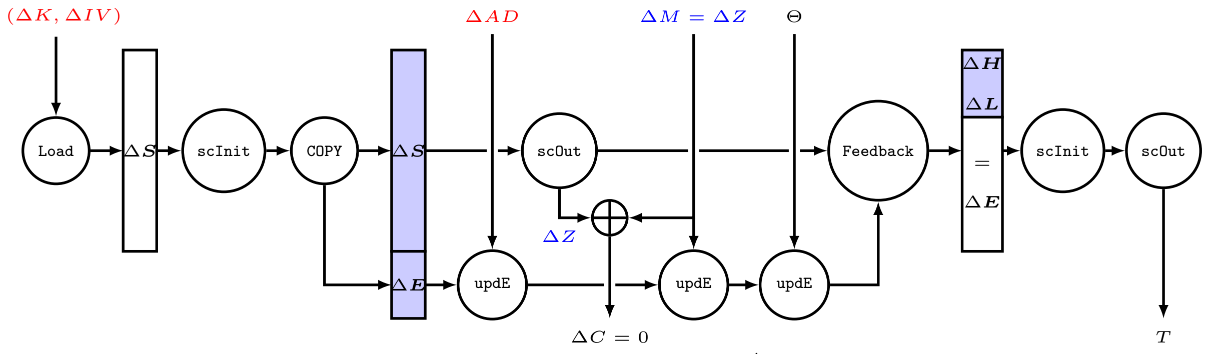

Fig. 9: The schematic of the committing attacks for CMT-1, CMT-2 and FROB security
Note: The blue-purple section indicates the uncontrollable state difference.

after keystream encryption under a pair of (k, IV) and (k', IV'); second, the probability bound by the number of active S-boxes, denoted by $\#ASB_r$, in the final `scInit` operation when $(\Delta H, \Delta L)$ assumes this value. Given that the probability of one byte being 0 is 2^{-8}, while the probability of it not being 0 is approximately 1, the probability that $(\Delta H, \Delta L)$ equals a certain value can be relaxed to $2^{-8\#D_0}$, where $\#D_0$ represents the number of bytes with a value of 0 in $(\Delta L, \Delta H)$. With the same technique as Section VI-A3 but but with no initial conditions imposed on $(\Delta H, \Delta L)$, we construct an MILP model $\mathcal{M}$ with the objective function defined as

$$\mathcal{M}.\texttt{obj} : \min\{8\#D_0 + 6\#ASB\},$$

upper bounding the probability of *ScState* collisions occurring in each round of final `scInit` when specific values of $(\Delta H, \Delta L)$ are assumed. The optimum solutions of the MILP models corresponding to r-round $(r = 2, \ldots, 7)$ `LOL2.0-Mini` and `LOL2.0-Double` are shown in Table X. As can be seen, for 128-bit tags, 5-round `LOL2.0-Mini` and 3-round `LOL2.0-Double` can reach the generic bound of 2^{128} .

TABLE X: The minimum $(8\#D_0 + 6\#ASB)$ values for r-round `LOL2.0-Mini-SCMAC` and `LOL2.0-Double-SCMAC`.

r	`LOL2.0-Mini-SCMAC`	`LOL2.0-Double-SCMAC`
2	44	48
3	84	132
4	116	192
5	150	-
6	180	-
7	192	-

Next, we check for CMT-3 and CMT-4 security. By manipulating AD, the differences at $\boldsymbol{E}$ can be freely modified since the knowledge of K and K' in `LOL2.0-Double`, thus `LOL2.0-Double` AEAD schemes cannot achieve CMT-3 and

CMT-4 security by setting $K = K', IV = IV', AD \neq AD'$ as well as $M = M'$. In `LOL2.0-Mini`, it is not immediately apparent that manipulating AD can eliminate the differences at $\boldsymbol{E}$. In fact, both `LOL2.0-Mini` and `LOL2.0-Double` act a stream cipher and are therefore inherently susceptible to IV reuse attacks.

VII. Software Implementation and Performance

In this section, we present the software performance of `LOL2.0-Mini` and `LOL2.0-Double` implemented in C++ (Visual Studio 2022) utilizing AVX2/AVX512/AES-NI/PCLMULQDQ intrinsic on a laptop with Intel i7-11800H CPU 2.30GHz with Turbo Boost up to 4.6GHz. All experiments are conducted in a single process/thread manner. The software performance is measured in Gbps using message lengths ranging from 32 to 16384 bytes. Test vectors and reference implementations of `LOL2.0-Mini`/`LOL2.0-Double` can be found in the Appendix. Although this section is written with Intel intrinsic notation, similar implementations can be applied to other CPUs such as AMD and ARM.

A. Stream Cipher Speeds

We also performed identical experiments on `SNOW-V`, `AEGIS`, and `Rocca` on the same platform using exactly the C++ source codes from the origins [4], [7], [8] for fair comparisons. The results are shown in Table XI.

TABLE XI: The software performance (Gbps) of `AEGIS`, `SNOW-V`, `Rocca`, `LOL2.0-Mini` and `LOL2.0-Double` in the pure encryption mode. Note that the `LOL2.0-Double` are implemented in both AVX2 and AVX512 manners while others only use the AVX2 instructions.

Bytes	`AEGIS`	`SNOW-V`	`Rocca`	`LOL2.0-Mini`		`LOL2.0-Double`		`LOL2.0-Double`†	
				`LFSR1`	`LFSR2`	`LFSR1`	`LFSR2`	`LFSR1`	`LFSR2`
32	11.17	5.31	12.23	8.16	4.77	6.16	6.65	4.98	7.67
64	21.31	9.67	23.03	15.65	16.68	11.35	13.15	13.86	14.53
96	25.67	13.50	32.22	20.92	21.97	17.27	19.07	19.18	19.57
128	30.60	16.51	40.24	25.71	26.70	21.21	24.32	22.95	23.68
160	35.64	19.49	48.07	30.02	31.39	25.15	28.13	26.87	27.89
192	39.55	21.19	54.41	33.78	35.27	29.59	32.00	30.67	31.09
224	43.48	23.85	60.01	36.71	38.07	32.92	35.67	34.49	35.36
256	47.08	25.57	66.10	40.25	41.68	36.67	39.37	38.30	39.14
1024	72.34	41.06	112.97	67.27	69.39	67.27	72.85	81.72	85.17
2048	80.88	46.70	130.85	75.58	78.59	79.84	84.91	105.19	108.01
4096	86.06	50.10	144.63	83.13	84.86	91.43	92.19	122.70	127.93
8192	88.92	52.10	151.86	85.88	88.87	96.52	98.47	132.96	137.14
16384	90.71	53.13	154.03	87.70	90.31	97.97	100.61	139.08	143.56

†: Implemented with the AVX512 instructions.

As can be seen, the encryption/decryption speeds of `LOL2.0-Mini` and `LOL2.0-Double` can reach 90 Gbps and 100 Gbps with AVX2 instructions respectively; the speed of `LOL2.0-Double` can reach 143 Gbps with AVX512 instructions, meeting the requirements not only for 5G but also for 6G networks. For speed comparison, we draw the following conclusions:

- `LOL2.0-Mini` and `AEGIS` exhibit similar efficiency levels, which are slower than `LOL2.0-Double` but significantly faster than `SNOW-V`. `Rocca` still maintains the highest speed, while `LOL2.0-Double` is only slightly slower than `Rocca`.
- The performance of both `LOL2.0-Mini` and `LOL2.0-Double` demonstrates that structures incorporating non-AES components can remain competitive with pure AES-based primitives. Moreover, considering the security concerns highlighted in the introduction and embracing diverse cipher designs (not solely relying on AES encryption round function), both `LOL2.0-Mini` and `LOL2.0-Double` offer enhanced competitiveness.
- Notably, the `AEGIS` performance on our platform is notably superior to the original 56 Gbps in [7], indicating significant progress in CPU support for AES-NI instructions. These advancements can be fully utilized in the frequent utilization of parallel AES operations. In contrast, `SNOW-V` maintains the same speed as the original 58 Gbps, highlighting the importance of optimal non-AES component designs.
- Since the λ in `LFSR2` requires one fewer instruction than that in `LFSR1`, Table XI shows that both `LOL2.0-Mini` and `LOL2.0-Double` achieve higher speeds with `LFSR2` compared to `LFSR1`. Consequently, in the AEAD speeds comparison, we have only included the data for `LFSR2`.

B. AEAD Speeds

In addition to the AEAD schemes combined with `SCMAC`, we also combine `LOL2.0-Mini` and `LOL2.0-Double` with standard GCM using the same method as [6]. For more detailed information, please refer to Section D. We conduct identical experiments on `SNOW-V` and `Rocca` on the same platform using exactly the C++ source codes from the origins [4], [7], [8] for fair comparisons. The results are shown in Table XI. As can be seen, the `SCMAC` AEAD mode exhibits higher efficiency compared to its GCM counterpart for both `LOL2.0-Mini` and `LOL2.0-Double`. Moreover, `LOL2.0-Double` AEAD scheme achieve the requirements for 6G networks.

Bytes	Rocca	SNOW-V-GCM	LOL2.0-Mini				LOL2.0-Double†			
			LFSR1		LFSR2		LFSR1		LFSR2	
			GCM	SCMAC	GCM	SCMAC	GCM	SCMAC	GCM	SCMAC
32	6.19	3.26	4.39	4.58	4.48	4.48	3.59	3.33	3.60	3.26
64	12.04	6.17	8.44	8.59	8.70	8.41	7.15	6.45	7.20	6.38
96	17.56	7.95	10.66	12.09	10.89	11.84	9.21	9.04	9.32	9.18
128	22.67	10.47	14.02	15.34	14.59	15.12	12.64	11.94	12.73	12.20
160	27.47	11.48	15.18	18.08	15.62	17.88	13.79	14.49	14.02	14.80
192	32.17	13.55	18.06	20.59	18.96	20.44	16.94	16.87	17.11	17.35
224	36.72	14.08	18.45	22.82	19.16	22.77	17.49	18.92	17.80	19.56
256	40.68	15.83	21.02	23.97	22.10	24.87	20.39	21.38	20.68	22.18
1024	87.79	25.90	33.37	43.61	35.67	43.45	37.91	53.19	38.75	57.35
2048	112.83	28.82	36.97	50.45	39.53	50.36	44.47	70.52	45.35	77.00
4096	129.95	30.73	39.29	54.93	41.84	54.89	48.26	83.99	49.57	92.96
8192	141.23	31.71	40.41	57.29	43.08	57.54	50.34	92.42	51.71	102.43
16384	148.31	31.88	40.98	58.69	43.67	59.06	51.71	97.82	53.10	109.70

VIII. Conclusions

In this paper, we propose `LOL2.0` framework for designing high-performance, high-security, and highly flexible stream ciphers through a rich component library and extension patterns; `SCMAC` framework for transforming stream ciphers to AEAD schemes. We give 2 concrete stream cipher designs named `LOL2.0-Mini` and `LOL2.0-Double`, which support 256-bit key and achieve software performances of 90 Gbps and 144 Gbps, satisfying the speed and security requirements in beyond 5G/6G mobile system. By combining `LOL2.0-Mini` and `LOL2.0-Double` with `SCMAC`, we propose two AEAD schemes, `LOL2.0-Mini-SCMAC` and `LOL2.0-Double-SCMAC`, supporting 128-bit tags equipped with thorough security evaluations and achieving speeds of 59 Gbps and 110 Gbps respectively, exhibiting higher efficiencies compared to their standard GCM counterparts.

References

[1] Matti Latva-aho and Kari Leppänen. Key drivers and research challenges for 6G ubiquitous wireless intelligence. 2019.

[2] Cheng Feng. Support of 256-bit algorithm in 5G mobile communication system. *Journal of Information Security Reserach*, 6(8):716–721, 2020.

[3] ARM. ARM architecture reference manual ARMv8, for ARMv8-a architecture profile. 2021.

[4] Patrik Ekdahl, Thomas Johansson, Alexander Maximov, and Jing Yang. A new SNOW stream cipher called SNOW-V. *IACR Trans. Symm. Cryptol.*, 2019(3):1–42, 2019.

[5] Patrik Ekdahl, Alexander Maximov, Thomas Johansson, and Jing Yang. SNOW-Vi : An extreme performance variant of SNOW-V for lower grade cpus. In *WiSec 2021*, pages 261–272. (ACM), 06 2021.

[6] Dengguo Feng, Lin Jiao, Yonglin Hao, Qunxiong Zheng, Wenling Wu, Wenfeng Qi, Lei Zhang, Liting Zhang, Siwei Sun, and Tian Tian. Lol: a highly flexible framework for designing stream ciphers. *Science China Information Sciences*, 67(9):192101, Aug 2024.

[7] Hongjun Wu and Bart Preneel. AEGIS: A fast authenticated encryption algorithm. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 185–201. Springer, Berlin, Heidelberg, August 2014.

[8] Kosei Sakamoto, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. Rocca: An efficient AES-based encryption scheme for beyond 5g. *IACR Trans. Symm. Cryptol.*, 2021(2):1–30, 2021.

[9] CAESAR. CAESAR: Competition for authenticated encryption: Security, applicability, and robustness, 2014.

[10] Ravi Anand, Subhadeep Banik, Andrea Caforio, Kazuhide Fukushima, Takanori Isobe, Shinsaku Kiyomoto, Fukang Liu, Yuto Nakano, Kosei Sakamoto, and Nobuyuki Takeuchi. An ultra-high throughput aes-based authenticated encryption scheme for 6g: Design and implementation. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part I*, volume 14344 of *Lecture Notes in Computer Science*, pages 229–248. Springer, 2023.

[11] Michel Mouly, Marie-Bernadette Pautet, and Thomas Haug. *The GSM System for Mobile Communications.* Telecom Publishing, 1992.

[12] ETSI SAGE. Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2, document 2: SNOW 3G specification, version 1.1, 2006. 2006.

[13] ETSI SAGE. pecification of the 3gpp confidentiality and integrity algorithms 128-EEA3 & 128-EIA3 , document 2: ZUC specification. version 1.6, 2011. *Security architecture and procedures for 5G System*, 3GPP TS 33.501 version 15.2.0 Release 15, 2011.

[14] Akinori Hosoyamada, Akiko Inoue, Ryoma Ito, Tetsu Iwata, Kazuhiko Mimematsu, Ferdinand Sibleyras, and Yosuke Todo. Cryptanalysis of Rocca and feasibility of its security claim. *IACR Transactions on Symmetric Cryptology*, 2022, Issue 3:123–151, 2022.

[15] Maria Eichlseder, Marcel Nageler, and Robert Primas. Analyzing the linear keystream biases in AEGIS. *IACR Trans. Symm. Cryptol.*, 2019(4):348–368, 2019.

[16] Zhen Shi, Chenhui Jin, Jiyan Zhang, Ting Cui, Lin Ding, and Yu Jin. A correlation attack on full SNOW-V and SNOW-vi. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 34–56. Springer, Cham, May / June 2022.

[17] Zhaocun Zhou, Dengguo Feng, and Bin Zhang. Efficient and extensive search for precise linear approximations with high correlations of full SNOW-V. *Des. Codes Cryptogr.*, 90(10):2449–2479, 2022.

[18] Matthias Hamann and Matthias Krause. On stream ciphers with provable beyond-the-birthday-bound security against time-memory-data tradeoff attacks. *Cryptography and Communications*, 10:959–1012, 2018.

[19] Jérémy Jean and Ivica Nikolic. Efficient design strategies based on the AES round function. In Thomas Peyrin, editor, *FSE 2016*, volume 9783 of *LNCS*, pages 334–353. Springer, Berlin, Heidelberg, March 2016.

[20] Ivica Nikolic. How to use metaheuristics for design of symmetric-key primitives. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part III*, volume 10626 of *LNCS*, pages 369–391. Springer, Cham, December 2017.

[21] Siwei Sun, Lei Hu, Peng Wang, Kexin Qiao, Xiaoshuang Ma, and Ling Song. Automatic security evaluation and (related-key) differential characteristic search: Application to SIMON, PRESENT, LBlock, DES(L) and other bit-oriented block ciphers. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 158–178. Springer, Berlin, Heidelberg, December 2014.

[22] Bing Sun, Zhiqiang Liu, Vincent Rijmen, Ruilin Li, Lei Cheng, Qingju Wang, Hoda AlKhzaimi, and Chao Li. Links among impossible differential, integral and zero correlation linear cryptanalysis. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 95–115. Springer, Berlin, Heidelberg, August 2015.

[23] Steve Babbage. Improved "exhaustive search" attacks on stream ciphers. 1995.

[24] Jovan Dj. Golic. Cryptanalysis of alleged A5 stream cipher. In *EUROCRYPT*, 1997.

[25] Alex Biryukov and Adi Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In *ASIACRYPT*, 2000.

[26] Nicolas T. Courtois and Josef Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Yuliang Zheng, editor, *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer, 2002.

[27] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. *IACR Trans. Symm. Cryptol.*, 2017(1):449–473, 2017.

[28] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 66–97. Springer, Cham, August 2017.

[29] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 155–186. Springer, Cham, August 2018.

[30] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3291–3308. USENIX Association, August 2022.

[31] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning oracle attacks. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 195–212. USENIX Association, August 2021.

[32] Morris Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. 2007.

Appendix A
The LFSR1 Setting

The λ of `LFSR1` used in `LOL2.0-Mini` cipher divides the 128-bit input x into 8 16-bit elements over the finite field $\mathbb{F}_{2^{16}}$ and computes the 128-bit output $\lambda(x)$ Equation (32)

$$(x_7,\dots,x_0) \xrightarrow{\lambda} (\alpha_7 \cdot x_7,\dots,\alpha_0 \cdot x_0) \tag{32}$$

where α_0,α_1,...,α_7 are the roots of 16-degree irreducible polynomials $g_0(y)$,$g_1(y)$,...,$g_7(y) \in \mathbb{F}_2[y]$ defined as Equation (33) and the $\alpha_i \cdot x_i$ operation denotes the multiplication the finite field $\mathbb{F}_{2^{16}} = \mathbb{F}_2[y]/g_i(y)$;

$$\begin{aligned}
g_0(y) =& y^{16}+y^{13}+y^{12}+y^{10}+y^8+y^7+y^6+y^3+1,\\
g_1(y) =& y^{16}+y^{15}+y^{12}+y^{10}+y^8+y^5+y^3+y+1,\\
g_2(y) =& y^{16}+y^{15}+y^{14}+y^{12}+y^{10}+y^7+y^5+y^4+1,\\
g_3(y) =& y^{16}+y^{14}+y^{11}+y^9+y^7+y^5+y^4+y^2+1,\\
g_4(y) =& y^{16}+y^{15}+y^{13}+y^9+y^7+y^4+1,\\
g_5(y) =& y^{16}+y^{14}+y^{13}+y^{12}+y^{11}+y^{10}+y^9+y^7+y^6+y^5+y^3+y^2+1,\\
g_6(y) =& y^{16}+y^{15}+y^{13}+y^9+y^8+y^4+y^3+y+1,\\
g_7(y) =& y^{16}+y^{14}+y^{13}+y^{12}+y^{11}+y^{10}+y^7+y^5+1.
\end{aligned} \tag{33}$$

The characteristic polynomial of `LFSR1` in `LOL2.0-Mini` is primitives of degree 256 as follows; therefore, the corresponding LFSR has the maximum periods of $2^{256}-1$.

`LFSR1:` $\eta(x) = x^{256}+x^{252}+x^{251}+x^{250}+x^{246}+x^{245}+x^{242}+x^{241}+x^{238}+x^{234}+x^{229}+x^{227}+x^{225}+x^{222}+x^{220}+x^{219}+x^{217}+x^{215}+x^{214}+x^{213}+$
$x^{212}+x^{211}+x^{210}+x^{208}+x^{207}+x^{204}+x^{203}+x^{198}+x^{196}+x^{194}+x^{191}+x^{186}+x^{185}+x^{184}+x^{183}+x^{182}+x^{179}+x^{178}+x^{177}+x^{175}+x^{170}+x^{168}+$
$x^{167}+x^{166}+x^{165}+x^{161}+x^{160}+x^{157}+x^{154}+x^{153}+x^{152}+x^{151}+x^{150}+x^{148}+x^{147}+x^{146}+x^{145}+x^{144}+x^{141}+x^{140}+x^{139}+x^{138}+x^{134}+x^{133}+$
$x^{131}+x^{130}+x^{126}+x^{123}+x^{122}+x^{121}+x^{119}+x^{118}+x^{116}+x^{115}+x^{113}+x^{110}+x^{109}+x^{108}+x^{107}+x^{104}+x^{103}+x^{102}+x^{101}+x^{100}+x^{98}+x^{97}+$
$x^{94}+x^{93}+x^{92}+x^{89}+x^{88}+x^{86}+x^{84}+x^{83}+x^{82}+x^{81}+x^{80}+x^{79}+x^{78}+x^{77}+x^{76}+x^{74}+x^{72}+x^{71}+x^{70}+x^{69}+x^{67}+x^{66}+x^{65}+x^{63}+x^{62}+x^{61}+$
$x^{60}+x^{56}+x^{53}+x^{52}+x^{51}+x^{50}+x^{48}+x^{47}+x^{44}+x^{42}+x^{39}+x^{38}+x^{37}+x^{34}+x^{32}+x^{31}+x^{30}+x^{29}+x^{28}+x^{26}+x^{25}+x^{24}+x^{21}+x^{20}+x^9+x^8+1.$

The λ of `LFSR1` used in `LOL2.0-Double` cipher divides the 256-bit input x into 16 16-bit cells $x = (x_{15}, \ldots, x_0)$ and compute the 256-bit output $\lambda(x)$ as Equation (34)

$$(x_{15}, \ldots, x_0) \xrightarrow{\lambda} (\alpha_{15} \cdot x_{15}, \ldots, \alpha_0 \cdot x_0) \tag{34}$$

where α_0,α_1,. . .,α_{15} are the roots of 16-degree irreducible polynomials $g_0(y)$,$g_1(y)$,. . .,$g_{15}(y) \in \mathbb{F}_2[y]$ whose definitions are given in Equation (33) and Equation (35).

$$\begin{aligned}
g_8(y) =& y^{16} + y^{15} + y^{14} + y^{10} + y^8 + y^6 + y^4 + y + 1 \\
g_9(y) =& y^{16} + y^{14} + y^{13} + y^{12} + y^{11} + y^{10} + y^8 + y^7 + y^6 + y^2 + 1 \\
g_{10}(y) =& y^{16} + y^{11} + y^{10} + y^8 + y^7 + y + 1 \\
g_{11}(y) =& y^{16} + y^{15} + y^{13} + y^{12} + y^9 + y^7 + y^6 + y^5 + y^3 + y + 1 \\
g_{12}(y) =& y^{16} + y^{15} + y^{14} + y^{12} + y^{10} + y^8 + y^5 + y^3 + y^2 + y + 1 \\
g_{13}(y) =& y^{16} + y^{15} + y^{12} + y^{11} + y^{10} + y^9 + y^8 + y^7 + y^5 + y^4 + y^2 + y + 1 \\
g_{14}(y) =& y^{16} + y^{14} + y^{10} + y^7 + y^6 + y^5 + 1 \\
g_{15}(y) =& y^{16} + y^{15} + y^{14} + y^{13} + y^{12} + y^6 + y^5 + y^3 + 1
\end{aligned} \tag{35}$$

The characteristic polynomial of `LFSR1` in`LOL2.0-Double` is primitives of degree 512 as follows; therefore, the corresponding LFSR ha the maximum periods of $2^{512} - 1$.

`LFSR1:` $\eta(x) = x^{512} + x^{511} + x^{510} + x^{509} + x^{505} + x^{504} + x^{501} + x^{500} + x^{499} + x^{498} + x^{497} + x^{495} + x^{494} + x^{492} + x^{488} + x^{486} + x^{483} + x^{482} + x^{481} + x^{479} +$
$x^{477} + x^{476} + x^{475} + x^{474} + x^{472} + x^{471} + x^{470} + x^{466} + x^{464} + x^{461} + x^{460} + x^{457} + x^{455} + x^{454} + x^{452} + x^{449} + x^{448} + x^{442} + x^{441} + x^{440} + x^{439} +$
$x^{437} + x^{435} + x^{432} + x^{429} + x^{426} + x^{425} + x^{424} + x^{423} + x^{422} + x^{421} + x^{420} + x^{419} + x^{417} + x^{416} + x^{414} + x^{411} + x^{410} + x^{408} + x^{407} + x^{405} + x^{403} +$
$x^{400} + x^{396} + x^{393} + x^{390} + x^{389} + x^{388} + x^{386} + x^{381} + x^{379} + x^{378} + x^{377} + x^{374} + x^{372} + x^{369} + x^{368} + x^{367} + x^{360} + x^{358} + x^{357} + x^{355} + x^{354} +$
$x^{349} + x^{346} + x^{344} + x^{343} + x^{341} + x^{338} + x^{337} + x^{336} + x^{335} + x^{334} + x^{333} + x^{331} + x^{330} + x^{328} + x^{325} + x^{324} + x^{322} + x^{319} + x^{318} + x^{316} + x^{315} +$
$x^{312} + x^{309} + x^{307} + x^{304} + x^{302} + x^{300} + x^{297} + x^{295} + x^{293} + x^{292} + x^{291} + x^{290} + x^{285} + x^{284} + x^{283} + x^{280} + x^{276} + x^{268} + x^{267} + x^{263} + x^{258} + x^{257} +$
$x^{255} + x^{254} + x^{252} + x^{249} + x^{248} + x^{245} + x^{242} + x^{241} + x^{240} + x^{236} + x^{234} + x^{232} + x^{231} + x^{230} + x^{227} + x^{226} + x^{224} + x^{219} + x^{210} + x^{209} + x^{207} + x^{206} +$
$x^{205} + x^{203} + x^{200} + x^{197} + x^{196} + x^{193} + x^{188} + x^{187} + x^{186} + x^{184} + x^{180} + x^{173} + x^{172} + x^{171} + x^{167} + x^{166} + x^{162} + x^{160} + x^{159} + x^{157} + x^{155} + x^{151} +$
$x^{150} + x^{149} + x^{147} + x^{146} + x^{145} + x^{144} + x^{143} + x^{137} + x^{134} + x^{133} + x^{127} + x^{126} + x^{125} + x^{124} + x^{123} + x^{118} + x^{116} + x^{115} + x^{112} + x^{111} + x^{109} + x^{108} +$
$x^{103} + x^{102} + x^{101} + x^{100} + x^{99} + x^{98} + x^{95} + x^{94} + x^{92} + x^{91} + x^{90} + x^{86} + x^{84} + x^{81} + x^{80} + x^{79} + x^{78} + x^{72} + x^{69} + x^{68} + x^{65} + x^{63} + x^{62} + x^{61} + x^{60} +$
$x^{58} + x^{57} + x^{55} + x^{54} + x^{52} + x^{51} + x^{49} + x^{46} + x^{41} + x^{40} + x^{39} + x^{38} + x^{37} + x^{35} + x^{31} + x^{30} + x^{29} + x^{27} + x^{25} + x^{23} + x^{22} + x^{20} + x^{18} + x^{17} + x^{16} + 1.$

Appendix B
Reference implementations

Similar to [4], we present the reference implementations of `LOL2.0-Mini` and `LOL2.0-Double` with AVX2 (**Listing 1** for `LOL2.0-Mini` with `LFSR1`, **Listing 2** for `LOL2.0-Mini` with `LFSR2`, **Listing 3** for `LOL2.0-Double` with `LFSR1` and **Listing 4** for `LOL2.0-Double` with `LFSR2`). The `LOL2.0-Double` implementations with AVX512 are given as **Listing 5** for `LOL2.0-Double` with `LFSR1` and **Listing 6** for `LOL2.0-Double` with `LFSR2`.

A. Implement `LOL2.0-Mini` with `LFSR1` *with AVX2*

Listing 1: Implement `LOL2.0-Mini` with `LFSR1` with AVX2

```
#include <immintrin.h>
        const __m128i lol_mul8 = _mm_setr_epi16(0x35c9, 0x952b, 0xd4b1, 0x4ab5, 0xa291, 0x7eed,
            0xa31b, 0x7ca1);
const __m128i lol_shu = _mm_setr_epi8(2, 3, 4, 5, 14, 15, 8, 9, 12, 13, 6, 7, 0, 1, 10, 11);
const __m128i ZERO = _mm_setzero_si128();
#define LFSR_MINI(F,H,L)\
{       F =  _mm_xor_si128(_mm_slli_epi16(H, 1),_mm_and_si128(lol_mul8, _mm_srai_epi16(H, 15)))
    ;\
        F =_mm_xor_si128(F, _mm_shuffle_epi8(L, lol_shu));}


struct LOL2.0_MINI_LFSR1
{
        __m128i S0, S1, S2, F, G, Z, L, H, N;

        inline __m128i keystream(void)
        {

                G = _mm_aesenc_si128(S2, N);

                LFSR_MINI(F, H, L);
                Z = _mm_aesenc_si128(G, N);
                N = _mm_aesenc_si128(N, L);

                L = H;
                H = F;

                S2 = _mm_aesenc_si128(S1, S2);
                S1 = _mm_aesenc_si128(S0, S1);
                S0 = _mm_xor_si128(S0, _mm_xor_si128(F, G));

                return Z;
        }

        inline void keyiv_setup(const unsigned char* key, const unsigned char* iv)
        {
                S2 = _mm_load_si128((__m128i*)iv);
                S1 = _mm_loadu_si128(&((__m128i*)key)[0]);
                S0 = _mm_loadu_si128(&((__m128i*)key)[1]);
                H = L = N = Z = ZERO;

                for (int i = 0; i < 12; i++) {
                        LFSR_MINI(F, H, L);

                        G = _mm_aesenc_si128(S2, N);
                        Z = _mm_aesenc_si128(G, N);
                        N = _mm_xor_si128(Z, _mm_aesenc_si128(N, L));
                        L = H;
                        H = _mm_xor_si128(Z, F);
                        S2 = _mm_aesenc_si128(S1, S2);
                        S1 = _mm_aesenc_si128(S0, S1);
                        S0 = _mm_xor_si128(S0, _mm_xor_si128(F, G));

                }

                //reload
                H = _mm_xor_si128(H, _mm_loadu_si128(&((__m128i*)key)[1]));
                S0 = _mm_xor_si128(S0, _mm_loadu_si128(&((__m128i*)key)[0]));
        }
};
```

B. Implement `LOL2.0-Mini` LFSR2 *using AVX2*

Listing 2: Implement `LOL2.0-Mini` `LFSR2` using AVX2

```
#include <immintrin.h>
const __m128i lol_shu = _mm_setr_epi8(2, 3, 4, 5, 14, 15, 8, 9, 12, 13, 6, 7, 0, 1, 10, 11);
const __m128i ZERO = _mm_setzero_si128();


#define type4blend(x) _mm_blend_epi16(_mm_slli_epi32((x), 5), _mm_srli_epi16((x), 6), 0x19)
#define LFSR_MINIT4(F,H,L)\
{       F =  type4blend(H);\
        F =_mm_xor_si128(F, _mm_shuffle_epi8(L, lol_shu));}


struct LOL2.0_MINI_LFSR2
{
        __m128i S0, S1, S2, F, G, Z, L, H, N;

        inline __m128i keystream(void)
        {

                G = _mm_aesenc_si128(S2, N);

                LFSR_MINIT4(F, H, L);
                Z = _mm_aesenc_si128(G, N);
                N = _mm_aesenc_si128(N, L);

                L = H;
                H = F;
                S2 = _mm_aesenc_si128(S1, S2);
                S1 = _mm_aesenc_si128(S0, S1);
                S0 = _mm_xor_si128(S0, _mm_xor_si128(F, G));

                return Z;
        }

        inline void keyiv_setup(const unsigned char* key, const unsigned char* iv)
        {

                S2 = _mm_load_si128((__m128i*)iv);
                S1 = _mm_loadu_si128(&((__m128i*)key)[0]);
                S0 = _mm_loadu_si128(&((__m128i*)key)[1]);
                H = L = N = Z = ZERO;

                for (int i = 0; i < 12; i++) {
                        LFSR_MINIT4(F, H, L);

                        G = _mm_aesenc_si128(S2, N);
                        Z = _mm_aesenc_si128(G, N);
                        N = _mm_xor_si128(Z, _mm_aesenc_si128(N, L));
                        L = H;
                        H = _mm_xor_si128(Z, F);
                        S2 = _mm_aesenc_si128(S1, S2);
                        S1 = _mm_aesenc_si128(S0, S1);
                        S0 = _mm_xor_si128(S0, _mm_xor_si128(F, G));

                }

                //reload
                H = _mm_xor_si128(H, _mm_loadu_si128(&((__m128i*)key)[1]));
                S0 = _mm_xor_si128(S0, _mm_loadu_si128(&((__m128i*)key)[0]));
        }
};
```

C. Implement `LOL2.0-Double` *with* `LFSR1` *using AVX2*

Listing 3: Implement `LOL2.0-Double` with `LFSR1` using AVX2

```
#include <immintrin.h>
const __m256i lol_mul16 = _mm256_setr_epi16(0x35c9, 0x952b, 0xd4b1, 0x4ab5, 0xa291, 0x7eed, 0
    xa31b, 0x7ca1,
0xc553, 0x7dc5, 0x0d83, 0xb2eb, 0xd52f, 0x9fb7, 0x44e1, 0xf069);
const __m128i ZERO = _mm_setzero_si128();
const __m256i lol_shu0 = _mm256_setr_epi8(6, 7, 0xff, 0xff, 10, 11, 2, 3, 0xff, 0xff, 0xff, 0
    xff, 14, 15, 8, 9, 2, 3,
0xff, 0xff, 0, 1, 0xff, 0xff, 12, 13, 14, 15, 0xff, 0xff, 6, 7);
const __m256i lol_shu1 = _mm256_setr_epi8(0xff, 0xff, 0, 1, 0xff, 0xff, 4, 5, 0xff, 0xff, 0xff,
     0xff, 12, 13, 0xff, 0xff, 0xff, 0xff,
8, 9, 0xff, 0xff, 0xff, 0xff, 10, 11, 4, 5, 0xff, 0xff, 0xff, 0xff);
#define LFSR_DOUBLE(H,L)\
{       __m256i F = _mm256_xor_si256(_mm256_slli_epi16(H, 1),_mm256_and_si256(lol_mul16,
    _mm256_srai_epi16(H, 15)));\
        F = _mm256_xor_si256(_mm256_xor_si256(_mm256_shuffle_epi8(L, lol_shu0),
            _mm256_permute4x64_epi64(_mm256_shuffle_epi8( L, lol_shu1),0x4e)),F);\
        L = H;\
        H = F;}

struct LOL2.0_DOUBLE_LFSR1
{
        __m256i H, L;
        __m128i S0, S1, S2, S3, N0, N1;

        inline __m256i keystream(void)
        {
                __m128i G0, G1, Z0, Z1;
                G0 = _mm_aesenc_si128(S1, N0);
                G1 = _mm_aesenc_si128(S3, N1);
                Z0 = _mm_aesenc_si128(G0, N0);
                Z1 = _mm_aesenc_si128(G1, N1);

                N0 = _mm_aesenc_si128(N0, _mm256_castsi256_si128(L));
                N1 = _mm_aesenc_si128(N1, _mm256_extracti128_si256(L, 1));

                LFSR_DOUBLE(H, L);

                S1 = _mm_aesenc_si128(S0, S1);
                S0 = _mm_xor_si128(_mm_xor_si128(G1, S0), _mm256_castsi256_si128(H));
                S3 = _mm_aesenc_si128(S2, S3);
                S2 = _mm_xor_si128(_mm_xor_si128(G0, S2), _mm256_extracti128_si256(H, 1));

                return _mm256_set_m128i(Z0, Z1);
        }

        inline void keyiv_setup(const unsigned char* key, const unsigned char* iv)
        {
                __m128i G0, G1, Z0, Z1;
                S1 = _mm_loadu_si128(&((__m128i*)iv)[0]);
                S3 = _mm_loadu_si128(&((__m128i*)iv)[1]);
                S0 = _mm_loadu_si128(&((__m128i*)key)[0]);
                S2 = _mm_loadu_si128(&((__m128i*)key)[1]);
                H = L = _mm256_setzero_si256();
                N0 = N1 = ZERO;

                for (int i = 0; i < 12; i++)
                {
                        G0 = _mm_aesenc_si128(S1, N0);
                        G1 = _mm_aesenc_si128(S3, N1);
                        Z0 = _mm_aesenc_si128(G0, N0);
                        Z1 = _mm_aesenc_si128(G1, N1);

                        N0 = _mm_xor_si128(_mm_aesenc_si128(N0, _mm256_castsi256_si128(L)), Z1)
                            ;
                        N1 = _mm_xor_si128(_mm_aesenc_si128(N1, _mm256_extracti128_si256(L, 1))
                            , Z0);

                        LFSR_DOUBLE(H, L);

                        S1 = _mm_aesenc_si128(S0, S1);
                        S0 = _mm_xor_si128(_mm_xor_si128(G1, S0), _mm256_castsi256_si128(H));
                        S3 = _mm_aesenc_si128(S2, S3);
                        S2 = _mm_xor_si128(_mm_xor_si128(G0, S2), _mm256_extracti128_si256(H,
                            1));

                        H = _mm256_xor_si256(H, _mm256_set_m128i(Z0, Z1));
                }

                //reload
                H = _mm256_xor_si256(H, _mm256_loadu_si256((__m256i*)key));
        }
};
```

D. Implement `LOL2.0-Double` *with* `LFSR2` *using AVX2*

Listing 4: Implement `LOL2.0-Double` with `LFSR2` using AVX2

```
#include <immintrin.h>
const __m256i Co = _mm256_set_epi8(0, 0, 0x80, 0x80, 0, 0, 0x80, 0x80, \
0, 0, 0x80, 0x80, 0x80, 0x80, 0, 0, \
0, 0, 0, 0, 0, 0, 0x80, 0x80, \
0x80, 0x80, 0, 0, 0, 0, 0x80, 0x80);
const __m128i ZERO = _mm_setzero_si128();
const __m256i lol_shu0 = _mm256_setr_epi8(6, 7, 0xff, 0xff, 10, 11, 2, 3, 0xff, 0xff, 0xff, 0
    xff, 14, 15, 8, 9, 2, 3,0xff, 0xff, 0, 1, 0xff, 0xff, 12, 13, 14, 15, 0xff, 0xff, 6, 7);
const __m256i lol_shu1 = _mm256_setr_epi8(0xff, 0xff, 0, 1, 0xff, 0xff, 4, 5, 0xff, 0xff, 0xff,
     0xff, 12, 13, 0xff, 0xff, 0xff, 0xff,8, 9, 0xff, 0xff, 0xff, 0xff, 10, 11, 4, 5, 0xff, 0
    xff, 0xff, 0xff);

#define LFSR_DOUBLE2401T4(H,L)\
{       __m256i  F=_mm256_blendv_epi8(_mm256_slli_epi32(H, 5), _mm256_srli_epi16(H, 6), Co);\
        F = _mm256_xor_si256(_mm256_xor_si256(_mm256_shuffle_epi8(L, lol_shu0),
            _mm256_permute4x64_epi64(_mm256_shuffle_epi8( L, lol_shu1),0x4e)),F);\
        L = H;\
        H = F;}

        struct LOL2.0_DOUBLE_LFSR2
        {
                __m256i H, L;
                __m128i S0, S1, S2, S3, N0, N1;

                inline __m256i keystream(void)
                {
                        __m128i G0, G1, Z0, Z1;
                        G0 = _mm_aesenc_si128(S1, N0);
                        G1 = _mm_aesenc_si128(S3, N1);
                        Z0 = _mm_aesenc_si128(G0, N0);
                        Z1 = _mm_aesenc_si128(G1, N1);

                        N0 = _mm_aesenc_si128(N0, _mm256_castsi256_si128(L));
                        N1 = _mm_aesenc_si128(N1, _mm256_extracti128_si256(L, 1));

                        LFSR_DOUBLE2401T4(H, L);

                        S1 = _mm_aesenc_si128(S0, S1);
                        S0 = _mm_xor_si128(_mm_xor_si128(G1, S0), _mm256_castsi256_si128(H));
                        S3 = _mm_aesenc_si128(S2, S3);
                        S2 = _mm_xor_si128(_mm_xor_si128(G0, S2), _mm256_extracti128_si256(H,
                            1));

                        return _mm256_set_m128i(Z0, Z1);
                }

                inline void keyiv_setup(const unsigned char* key, const unsigned char* iv)
                {
                        __m128i G0, G1, Z0, Z1;
                        S1 = _mm_loadu_si128(&((__m128i*)iv)[0]);
                        S3 = _mm_loadu_si128(&((__m128i*)iv)[1]);
                        S0 = _mm_loadu_si128(&((__m128i*)key)[0]);
                        S2 = _mm_loadu_si128(&((__m128i*)key)[1]);
                        H = L = _mm256_setzero_si256();
                        N0 = N1 = ZERO;

                        for (int i = 0; i < 12; i++)
                        {
                                G0 = _mm_aesenc_si128(S1, N0);
                                G1 = _mm_aesenc_si128(S3, N1);
                                Z0 = _mm_aesenc_si128(G0, N0);
                                Z1 = _mm_aesenc_si128(G1, N1);

                                N0 = _mm_xor_si128(_mm_aesenc_si128(N0, _mm256_castsi256_si128(
                                    L)), Z1);
                                N1 = _mm_xor_si128(_mm_aesenc_si128(N1,
                                    _mm256_extracti128_si256(L, 1)), Z0);

                                LFSR_DOUBLE2401T4(H, L);

                                S1 = _mm_aesenc_si128(S0, S1);
                                S0 = _mm_xor_si128(_mm_xor_si128(G1, S0),
                                    _mm256_castsi256_si128(H));
                                S3 = _mm_aesenc_si128(S2, S3);
                                S2 = _mm_xor_si128(_mm_xor_si128(G0, S2),
                                    _mm256_extracti128_si256(H, 1));

                                H = _mm256_xor_si256(H, _mm256_set_m128i(Z0, Z1));
                        }

                        //reload
                        H = _mm256_xor_si256(H, _mm256_loadu_si256((__m256i*)key));
                }
        };
```

E. Implement `LOL2.0-Double` *with* `LFSR1` *using AVX-512*

Listing 5: Implement `LOL2.0-Double` with `LFSR1` using AVX-512

```
#include <immintrin.h>
const __m256i lol_mul16 = _mm256_setr_epi16(0x35c9, 0x952b, 0xd4b1, 0x4ab5, 0xa291, 0x7eed, 0
    xa31b, 0x7ca1,
0xc553, 0x7dc5, 0x0d83, 0xb2eb, 0xd52f, 0x9fb7, 0x44e1, 0xf069);

const __m256i Co = _mm256_set_epi8(0, 0, 0x80, 0x80, 0, 0, 0x80, 0x80, \
0, 0, 0x80, 0x80, 0x80, 0x80, 0, 0, \
0, 0, 0, 0, 0, 0, 0x80, 0x80, \
0x80, 0x80, 0, 0, 0, 0, 0x80, 0x80);

const __m256i lol_shu16 = _mm256_set_epi8(23, 22, 13, 12, 31, 30, 29, 28, 5, 4, 17, 16, 1, 0,
    19, 18,
9, 8, 15, 14, 21, 20, 27, 26, 3, 2, 11, 10, 25, 24, 7, 6);
const __m256i ZERO256 = _mm256_setzero_si256();
#define LFSR_DOUBLE(H,L)\
{       F = _mm256_xor_si256(_mm256_slli_epi16(H, 1),_mm256_and_si256(lol_mul16,
    _mm256_srai_epi16(H, 15)));\
        F = _mm256_xor_si256( _mm256_permutexvar_epi8(lol_shu16,L),F);\
        L = H;\
        H = F;}
#define AES_ENC_256(C,M) C= _mm256_aesenc_epi128(M,ZERO256);
#define AES_ENC_KEY_256(C,M,R) C= _mm256_aesenc_epi128(M,R);

struct LOL2.0_DOUBLE_LFSR1
{
        __m256i F, H, L, G, S0, S1, Z, N;
        inline __m256i keystream(void)
        {
                AES_ENC_KEY_256(G, S1, N);
                AES_ENC_KEY_256(Z, G, N);


                AES_ENC_KEY_256(N, N, L);
                LFSR_DOUBLE(H, L);

                AES_ENC_KEY_256(S1, S0, S1);
                S0 = _mm256_xor_si256(S0, _mm256_xor_si256(H, _mm256_permute4x64_epi64(G, 0x4e)
                    ));

                Z = _mm256_permute4x64_epi64(Z, 0x4e);
                return Z;
        }

        inline void keyiv_setup(const unsigned char* key, const unsigned char* iv)
        {

                S0 = _mm256_loadu_si256((__m256i*)key);
                S1 = _mm256_loadu_si256((__m256i*)iv);
                L = H = N = _mm256_setzero_si256();

                for (int i = 0; i < 12; i++)
                {
                        AES_ENC_KEY_256(G, S1, N);
                        AES_ENC_KEY_256(Z, G, N);
                        Z = _mm256_permute4x64_epi64(Z, 0x4e);

                        AES_ENC_KEY_256(N, N, L);
                        N = _mm256_xor_si256(N, Z);

                        LFSR_DOUBLE(H, L);

                        AES_ENC_KEY_256(S1, S0, S1);
                        S0 = _mm256_xor_si256(S0, _mm256_xor_si256(H, _mm256_permute4x64_epi64(
                            G, 0x4e)));

                        H = _mm256_xor_si256(H, Z);
                }

                //reload
                H = _mm256_xor_si256(H, _mm256_loadu_si256((__m256i*)key));
        }
};
```

F. Implement `LOL2.0-Double` *with* `LFSR2` *using AVX-512*

Listing 6: Implement `LOL2.0-Double` with `LFSR2` using AVX-512

```
#include <immintrin.h>

const __m256i Co = _mm256_set_epi8(0, 0, 0x80, 0x80, 0, 0, 0x80, 0x80, \
        0, 0, 0x80, 0x80, 0x80, 0x80, 0, 0, \
        0, 0, 0, 0, 0, 0x80, 0x80, \
        0x80, 0x80, 0, 0, 0, 0, 0x80, 0x80);

const __m256i lol_shu16 = _mm256_set_epi8(23, 22, 13, 12, 31, 30, 29, 28, 5, 4, 17, 16, 1, 0,
    19, 18,9, 8, 15, 14, 21, 20, 27, 26, 3, 2, 11, 10, 25, 24, 7, 6);
const __m256i ZERO256 = _mm256_setzero_si256();

#define LFSR_DOUBLE2401T4(H,L)\
{       __m256i F=_mm256_blendv_epi8(_mm256_slli_epi32(H, 5), _mm256_srli_epi16(H, 6), Co);\
        F = _mm256_xor_si256( _mm256_permutexvar_epi8(lol_shu16,L),F);\
        L = H;\
        H = F;}

#define AES_ENC_256(C,M) C= _mm256_aesenc_epi128(M,ZERO256);
#define AES_ENC_KEY_256(C,M,R) C= _mm256_aesenc_epi128(M,R);


struct LOL_DOUBLEV2
{
        __m256i F, H, L, G, S0, S1, Z, N;
        inline __m256i keystream(void)
        {
                AES_ENC_KEY_256(G, S1, N);

                AES_ENC_KEY_256(Z, G, N);

                AES_ENC_KEY_256(N, N, L);
                LFSR_DOUBLE2401T4(H, L);

                AES_ENC_KEY_256(S1, S0, S1);
                S0 = _mm256_xor_si256(S0, _mm256_xor_si256(H, _mm256_permute4x64_epi64(G, 0x4e)
                    ));

                Z = _mm256_permute4x64_epi64(Z, 0x4e);
                return Z;
        }

        inline void keyiv_setup(const unsigned char* key, const unsigned char* iv)
        {
                S0 = _mm256_loadu_si256((__m256i*)key);
                S1 = _mm256_loadu_si256((__m256i*)iv);
                N = L = H = ZERO256;
                for (int i = 0; i < 12; i++)
                {
                        AES_ENC_KEY_256(G, S1, N);
                        AES_ENC_KEY_256(Z, G, N);
                        //Z = _mm256_xor_si256(G, N);
                        Z = _mm256_permute4x64_epi64(Z, 0x4e);

                        AES_ENC_KEY_256(N, N, L);
                        N = _mm256_xor_si256(N, Z);
                    LFSR_DOUBLE2401T4(H, L);

                    AES_ENC_KEY_256(S1, S0, S1);
                    S0 = _mm256_xor_si256(S0, _mm256_xor_si256(H, _mm256_permute4x64_epi64(
                        G, 0x4e)));

                    H = _mm256_xor_si256(H, Z);
                }

                //reload
                H = _mm256_xor_si256(H, _mm256_loadu_si256((__m256i*)key));
        }
};
```

G. *Implement* `updE` *of* `SCMAC` *following strategy No.10 of Table IV*

Listing 7: Implement `updE` of `SCMAC` following strategy No.10 of Table IV

```
        struct SCMAC6 {
                __m128i E[6];
                inline void upd(__m128i c2[2]) {
                        __m128i tmp = _mm_aesenc_si128(E[5], c2[0]);
                        E[5] = _mm_aesenc_si128(E[4], ZERO);
                        E[3] = _mm_aesenc_si128(E[2], E[3]);
                        E[2] = _mm_aesenc_si128(E[1], E[2]);
                        E[1] = _mm_aesenc_si128(E[0], E[1]);
                        E[0] = tmp;
                        E[3] = _mm_xor_si128(E[3], c2[0]);
                        E[2] = _mm_xor_si128(E[2], c2[1]);
                }

        };
```

H. *Implement* `updE` *of* `SCMAC` *following strategy No.2 of Table IV*

Listing 8: Implement `updE` of `SCMAC` following strategy No.2 of Table IV

```
struct SMAC4 {
        __m128i E[4];
        inline void upd(__m128i msg) {
                __m128i tmp = _mm_aesenc_si128(E[3], E[0]);
                E[3] = _mm_aesenc_si128(E[2], E[3]);
                E[2] = _mm_aesenc_si128(E[1], E[2]);
                E[1] = _mm_aesenc_si128(E[0], E[1]);
                E[0] = _mm_xor_si128(msg, tmp);
        }
};
```

Appendix C
Test vectors

The 8-bit bytes are separated with _ symbols. The 128/256-bit states are represented from the most significant byte to the least significant one from left to right. The test vectors here aim at showing the detailed the implementations of both the stream ciphers and the corresponding `SCMAC` AEAD schemes so we use the all-zero plaintexts to make the keystream bits identical to the ciphertexts for easy verifications. For generating test vectors of ciphertexts and tags, the bit length the of all-zero plaintext for `LOL2.0-Mini` is set $128 * 16 = 2048$ and that of `LOL2.0-Double` is $256 * 16 = 4096$; the bit length of associated data is set to 0.

A test vector for `LOL2.0-Mini-LFSR1` is presented as follows.

```
key256: 4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b_4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
iv128: 8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
t= -12:
H:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
S0:4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b
S1:4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
S2:8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
Z:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
t= -11:
H:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
```

```
N:87_22_b5_a5_5a_46_ab_3b_3a_82_51_3a_e2_e5_15_54
S0:58_5d_d0_cd_eb_63_90_2b_f3_04_b2_0a_1e_57_0d_a3
S1:a5_5e_e4_33_c1_79_bc_d9_7a_9a_3c_c3_99_1e_11_e1
S2:83_4d_51_b5_45_3f_63_8b_21_47_46_cb_7c_95_52_72
Z:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37
t= -10:
H:29_37_f0_b5_d4_20_48_e0_74_4b_e9_32_02_c8_21_8a
L:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37
N:ba_b4_c1_68_8c_6f_d9_23_86_e5_ff_36_51_93_ad_81
S0:73_60_2b_80_9e_cb_0e_f4_63_4d_5b_59_46_4a_73_6e
S1:95_eb_08_2b_f8_59_4d_e6_1b_93_3d_93_9f_70_09_3d
S2:17_ef_26_75_ae_f2_c8_f2_a9_95_a4_8e_3e_fb_a0_0f
Z:9d_14_fe_22_a6_6a_7a_c1_c7_89_8d_80_94_ef_cd_e4
t= -9:
H:0a_e5_bf_f9_30_9b_9c_8a_ed_b4_b9_a1_4e_97_a6_43
L:29_37_f0_b5_d4_20_48_e0_74_4b_e9_32_02_c8_21_8a
N:ce_3a_3d_ea_c0_c3_fe_88_9a_c2_5b_65_a1_8e_25_98
S0:07_81_02_7a_58_e6_2e_06_a6_ff_5c_87_81_2d_88_33
S1:a3_69_f5_c8_e8_b6_91_a6_92_a6_8a_72_06_62_4a_c3
S2:40_7f_eb_12_03_04_7a_5b_71_15_5e_21_ef_08_a5_c9
Z:61_ae_8b_bf_bf_d7_db_8c_cd_7a_5b_35_79_5e_64_d1
t= -8:
H:b3_a5_95_4e_fb_6d_af_2f_32_24_1c_1f_d5_68_5f_7a
L:0a_e5_bf_f9_30_9b_9c_8a_ed_b4_b9_a1_4e_97_a6_43
N:bc_5f_b4_53_38_32_95_a4_17_e9_f2_17_73_19_c8_15
S0:e1_a1_97_52_42_16_b8_7f_54_e2_69_bf_90_de_bf_9e
S1:d0_e5_ec_4c_be_69_47_1d_8e_8c_e7_0f_0d_b8_51_cb
S2:bc_bf_43_07_bd_58_23_c7_17_8c_78_af_b9_da_af_53
Z:72_4f_68_2d_ee_10_c4_1f_eb_19_92_db_a1_74_24_fd
t= -7:
H:49_07_3c_17_81_b2_ed_cf_00_a8_cb_16_14_ef_2d_b4
L:b3_a5_95_4e_fb_6d_af_2f_32_24_1c_1f_d5_68_5f_7a
N:bc_a2_af_d1_ad_d6_ed_49_fe_0b_21_a9_57_1b_2b_e0
S0:cb_74_ed_c1_f8_de_95_f3_b2_7c_d2_b5_55_b7_d2_f9
S1:18_b2_63_e0_06_69_99_1c_08_2c_01_f9_45_d7_82_72
S2:35_66_d1_f7_da_c1_ad_a8_41_91_f2_b6_15_28_e3_43
Z:62_77_13_d3_e4_31_ae_f9_f8_6a_f9_cd_92_b5_dd_d7
t= -6:
H:e6_60_54_8d_56_db_ae_42_d7_d8_22_c9_f8_a2_40_15
L:49_07_3c_17_81_b2_ed_cf_00_a8_cb_16_14_ef_2d_b4
N:95_9b_c9_5d_14_ca_75_c5_fc_9c_a3_cd_e1_4f_f5_39
S0:98_2c_e0_71_50_af_f5_91_24_5e_71_76_df_af_7a_43
S1:b7_9c_1b_85_83_21_e9_67_37_f3_d1_58_63_e0_52_9d
S2:b1_c7_ea_05_f0_ea_fe_a3_f4_3a_9b_ef_11_11_e6_13
Z:8f_03_73_d9_19_76_42_03_79_a7_d3_f1_cd_63_ce_15
t= -5:
H:e2_c1_3e_42_03_09_b0_5b_ab_32_47_80_34_93_c6_6a
L:e6_60_54_8d_56_db_ae_42_d7_d8_22_c9_f8_a2_40_15
N:99_0d_9e_76_fc_06_bf_89_1d_4b_d8_32_93_8b_cf_7f
S0:8c_15_d6_d1_97_33_7f_ed_c8_66_bf_19_0e_a6_14_36
S1:6d_a0_ac_a6_ef_e9_0c_09_cf_76_33_f6_1b_61_d8_68
S2:dd_65_86_d1_05_ac_9c_23_67_ff_4f_3c_79_70_71_b9
Z:d3_12_ba_ec_ae_17_72_59_a3_f8_4b_15_9b_ea_52_af
t= -4:
H:34_ec_29_f1_c4_2b_46_8b_e5_2a_02_86_f3_cc_cf_72
L:e2_c1_3e_42_03_09_b0_5b_ab_32_47_80_34_93_c6_6a
N:37_70_08_1b_d4_1b_2c_b2_99_80_c4_7a_e4_c5_81_45
S0:db_ec_46_c9_d4_ab_8e_9a_a5_d0_23_04_a0_71_b6_9d
S1:ed_81_a0_74_09_21_2a_6e_6e_24_18_da_17_5d_f7_dd
S2:4d_2a_f7_dd_77_7c_04_ec_fa_72_7e_ee_13_8e_b7_4f
Z:db_14_15_60_15_e1_d0_21_57_b9_6b_e6_b8_23_8e_cd
t= -3:
H:dc_6b_3d_37_d0_31_2e_41_aa_b3_81_b2_79_ac_2d_52
L:34_ec_29_f1_c4_2b_46_8b_e5_2a_02_86_f3_cc_cf_72
N:ef_81_25_29_86_c8_65_38_b6_89_83_fb_56_8e_52_4b
S0:c6_a1_66_96_3f_da_09_77_99_66_9e_76_d8_ee_02_88
S1:3f_2a_39_3f_2f_94_de_4e_6f_73_71_20_d8_31_fa_d0
S2:bc_bf_ee_6d_44_21_0b_21_11_d3_1c_88_b1_fc_7e_92
Z:b6_ba_a8_bf_8d_b8_9d_15_9a_09_66_7f_4c_9f_b2_ec
t= -2:
H:24_2e_5d_c1_35_fa_93_b1_7d_bf_81_d8_30_ef_a3_9d
L:dc_6b_3d_37_d0_31_2e_41_aa_b3_81_b2_79_ac_2d_52
N:cb_a2_d2_9e_79_ac_bc_51_e8_5f_50_13_5d_13_49_e2
S0:ce_52_86_a8_19_84_fd_97_89_0b_bc_9a_05_23_ec_12
S1:3a_34_67_db_f7_2b_db_2c_d6_8d_7c_00_a0_97_65_bd
S2:ab_dd_5e_6a_ab_6a_fa_26_e9_0b_c4_5e_4a_f4_40_db
Z:24_72_e8_dd_0e_5f_e6_c2_24_e7_62_e1_c1_31_0a_f5
t= -1:
H:8f_fc_b1_78_0e_db_e4_a4_47_5b_82_2a_5c_4d_03_c9
L:24_2e_5d_c1_35_fa_93_b1_7d_bf_81_d8_30_ef_a3_9d
N:55_8e_9b_bd_92_85_ce_ba_78_6f_bd_d1_8a_5e_ac_94
S0:7d_66_cd_b0_78_87_7a_be_07_e6_6b_33_b2_5b_4b_82
S1:31_09_8b_f2_43_59_16_6a_87_dc_07_fc_d7_03_0c_4f
S2:bb_2b_f2_e1_c2_94_ef_2e_36_da_06_b0_3e_fc_92_6e
Z:17_91_27_a8_cf_9c_5c_60_92_64_89_40_bc_21_08_96
End of Initialization (t=0):
```

```
H:61_71_05_4c_be_00_ba_9b_a3_fb_4b_f5_49_b0_ed_ca
L:8f_fc_b1_78_0e_db_e4_a4_47_5b_82_2a_5c_4d_03_c9
N:74_b5_95_48_42_bd_4b_b3_68_2e_87_a5_4b_26_04_10
S0:71_2b_4e_18_d8_54_2c_e6_30_9b_1e_51_24_c3_98_10
S1:5d_85_24_db_bf_6c_57_0d_5e_fa_09_10_6f_5b_27_00
S2:77_65_17_6d_ac_64_0d_01_a7_3d_35_0d_32_62_06_25
Z:79_ea_27_32_5f_c8_50_29_39_94_60_e0_13_bb_2f_2c
Output Keystream Z's for t=0...15 (also the ciphertext of the 2048-bit all-zero plaintext):
78_e2_b6_49_c3_34_77_9b_4b_ba_ae_68_ad_e7_fd_c1
a7_01_22_5e_05_62_c6_bb_49_6e_b9_23_18_1a_9d_ad
6a_b8_b9_11_b5_23_86_bf_d7_d3_e2_2c_57_89_d9_95
b9_10_0b_98_de_20_5f_d1_ad_ac_66_76_39_a8_41_e9
49_91_31_06_38_28_6a_4e_2d_08_d3_d1_2e_89_08_6f
d4_43_09_b2_bb_c9_86_4e_77_78_86_f6_92_b9_a3_d1
b8_76_12_cc_23_20_56_65_b9_da_92_68_b6_be_5a_0e
d0_5d_cd_05_12_04_bf_c3_76_f6_d6_d2_55_a6_ea_52
4f_b4_b1_6b_b0_09_1e_ea_fe_89_3c_82_b4_c0_e1_84
7d_88_fa_25_59_61_98_86_15_bc_df_e1_45_e5_81_d7
21_84_35_c0_5c_a4_32_c3_c9_5e_8e_b0_91_09_4a_01
d0_b7_2e_54_60_11_01_5f_ff_e9_d5_7f_99_0e_95_8c
6f_11_03_1b_67_d4_11_59_7b_9c_aa_87_ca_9b_61_c8
06_ae_e7_de_58_c0_1b_40_5c_d7_d8_6f_21_98_ee_cc
09_c1_d3_db_ea_0b_96_2a_b1_db_3f_7f_ef_30_59_fc
b3_8f_c5_1b_06_0c_8b_49_46_d8_44_b0_d7_73_05_d2
The 128-bit tag T:
33_63_90_15_1b_a7_b5_fc_24_67_d7_c4_7a_a0_32_f7
```

A test vector for `LOL2.0-Mini-LFSR2` is presented as follows.

```
key256: 4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b_4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
iv128: 8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
t= -12:
H:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
S0:4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b
S1:4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
S2:8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
Z:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
t= -11:
H:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N:87_22_b5_a5_5a_46_ab_3b_3a_82_51_3a_e2_e5_15_54
S0:58_5d_d0_cd_eb_63_90_2b_f3_04_b2_0a_1e_57_0d_a3
S1:a5_5e_e4_33_c1_79_bc_d9_7a_9a_3c_c3_99_1e_11_e1
S2:83_4d_51_b5_45_3f_63_8b_21_47_46_cb_7c_95_52_72
Z:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37
t= -10:
H:15_2e_26_e2_82_d3_79_e0_c6_ee_c6_a0_a4_21_cc_3c
L:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37
N:ba_b4_c1_68_8c_6f_d9_23_86_e5_ff_36_51_93_ad_81
S0:4f_79_fd_d7_c8_38_3f_f4_d1_e8_74_cb_e0_a3_9e_d8
S1:95_eb_08_2b_f8_59_4d_e6_1b_93_3d_93_9f_70_09_3d
S2:17_ef_26_75_ae_f2_c8_f2_a9_95_a4_8e_3e_fb_a0_0f
Z:9d_14_fe_22_a6_6a_7a_c1_c7_89_8d_80_94_ef_cd_e4
t= -9:
H:fd_4f_21_c8_bc_59_0c_ad_06_39_6b_74_cf_3e_e6_67
L:15_2e_26_e2_82_d3_79_e0_c6_ee_c6_a0_a4_21_cc_3c
N:ce_3a_3d_ea_c0_c3_fe_88_9a_c2_5b_65_a1_8e_25_98
S0:cc_32_4a_1c_82_d7_8f_21_ff_d7_a1_c0_a6_6d_25_a1
S1:dc_e0_4e_59_00_e4_7a_4b_20_b6_e8_65_ba_1a_da_d1
S2:40_7f_eb_12_03_04_7a_5b_71_15_5e_21_ef_08_a5_c9
Z:61_ae_8b_bf_bf_d7_db_8c_cd_7a_5b_35_79_5e_64_d1
t= -8:
H:59_78_9d_11_a3_df_e2_cf_92_e1_e9_75_80_08_83_45
L:fd_4f_21_c8_bc_59_0c_ad_06_39_6b_74_cf_3e_e6_67
N:80_46_62_04_6e_c1_a4_a4_a5_4c_dd_85_d5_f0_25_a3
S0:c0_cf_d7_6b_c0_95_54_b8_ad_0f_61_92_e2_fe_ce_33
S1:74_6a_ad_26_da_8d_62_e1_ed_55_f5_98_46_db_d5_a6
S2:d9_da_3f_4b_35_05_dd_42_6e_5c_70_db_48_ee_8b_ec
Z:72_4f_68_2d_ee_10_c4_1f_eb_19_92_db_a1_74_24_fd
t= -7:
H:c7_a9_14_1a_0f_a1_c5_6a_bd_80_44_5c_f1_e4_5d_e6
L:59_78_9d_11_a3_df_e2_cf_92_e1_e9_75_80_08_83_45
N:97_89_19_9d_3f_d7_64_79_02_30_14_70_72_c9_a0_4b
S0:5d_78_70_ce_df_22_f3_2f_e4_85_e4_cc_44_7c_c2_89
S1:67_a7_ff_92_ea_26_51_05_bd_86_e7_74_b2_76_17_81
S2:8a_34_86_10_a0_fe_91_e1_b8_5a_25_ea_a0_3b_0d_5a
Z:54_e3_50_5d_72_64_e7_29_b3_66_97_b3_9b_80_90_d5
t= -6:
H:5d_e3_8a_fe_2b_cd_5e_3e_15_6d_34_62_0e_74_ce_6c
L:c7_a9_14_1a_0f_a1_c5_6a_bd_80_44_5c_f1_e4_5d_e6
N:be_4f_eb_1c_c8_91_2c_4f_f0_32_6a_6f_6a_54_0a_b8
S0:6b_4d_2f_e5_eb_fb_28_fb_d6_81_74_31_42_25_8b_5e
S1:74_85_02_f9_46_f9_bf_a5_e7_8e_72_52_22_51_80_b0
S2:fa_91_79_46_c7_7d_e4_4f_f6_f8_dd_d6_c6_c2_b5_97
Z:0b_1e_8a_fb_4d_14_c0_3a_f5_54_e6_9a_db_8a_4f_13
```

```
t= -5:
H:99_ae_60_09_67_1c_55_bb_b2_de_b5_a8_a8_2f_77_5c
L:5d_e3_8a_fe_2b_cd_5e_3e_15_6d_34_62_0e_74_ce_6c
N:dd_49_a9_74_f1_34_24_2c_d8_ad_65_43_a3_90_86_96
S0:8b_8e_28_88_67_ff_cf_63_79_cd_5d_ce_45_6f_55_ba
S1:fc_59_5c_f2_8a_9c_2d_4d_3a_d4_cf_54_a9_04_89_dd
S2:2a_19_10_60_58_12_f0_9f_54_6b_fc_ca_12_e1_f0_89
Z:2a_7e_62_2f_a3_37_40_d9_77_e1_fe_41_22_ea_85_81
t= -4:
H:39_db_3c_23_ed_ab_9b_95_7d_03_2d_cf_50_04_c3_99
L:99_ae_60_09_67_1c_55_bb_b2_de_b5_a8_a8_2f_77_5c
N:5f_26_66_56_d2_40_43_8e_03_df_c0_11_08_49_e1_69
S0:04_da_76_91_64_b6_07_d7_b0_9e_09_14_8d_d9_a5_37
S1:fb_94_22_46_77_d9_eb_e2_f3_55_0e_ab_8e_d7_e2_d1
S2:b9_25_6a_ad_c5_df_0a_ee_9a_70_e2_e1_65_c6_f7_fb
Z:27_da_f3_6f_1b_4c_10_3d_21_f6_c5_2c_61_88_cc_30
t= -3:
H:55_b4_a7_dd_3f_af_21_ff_9f_83_a8_52_a9_3c_58_86
L:39_db_3c_23_ed_ab_9b_95_7d_03_2d_cf_50_04_c3_99
N:4f_cb_62_a9_8a_cf_5f_04_69_2f_b9_42_92_38_c6_01
S0:cd_29_f3_85_14_d1_75_64_36_38_d3_7b_8b_ea_24_1a
S1:e8_c0_f1_1d_bd_b7_2d_64_ce_23_c1_1a_8b_dd_5a_41
S2:c2_00_f1_a3_f1_df_b0_68_c5_94_36_3e_a7_28_d4_5b
Z:09_cf_54_e1_38_02_43_98_cb_cc_88_1c_1c_0c_f3_a7
t= -2:
H:20_6e_94_fa_69_ac_dd_82_5c_f1_b3_8d_7d_fd_21_22
L:55_b4_a7_dd_3f_af_21_ff_9f_83_a8_52_a9_3c_58_86
N:71_45_71_f1_45_a6_37_57_82_15_6a_9b_6c_0f_62_d1
S0:f4_23_f4_5a_1b_84_f0_24_37_8b_c7_d0_89_e2_1c_bb
S1:94_7a_f9_8f_f8_e3_e3_77_ef_89_08_7c_4f_b9_cb_37
S2:59_d8_22_75_59_22_79_5c_20_c6_d7_95_c5_2c_dc_fc
Z:7b_51_ac_c3_e1_4b_e1_26_c5_1a_80_16_77_b9_70_44
t= -1:
H:ff_02_39_e4_75_6c_ad_e4_ab_19_3d_3d_2f_7e_66_5a
L:20_6e_94_fa_69_ac_dd_82_5c_f1_b3_8d_7d_fd_21_22
N:4f_c0_32_74_fe_fa_a7_9d_7c_b4_68_d8_a8_3e_62_99
S0:b4_35_c4_a2_bd_c3_30_24_6b_bd_d1_0d_29_de_1e_e3
S1:a9_c8_8c_c5_aa_f4_99_c3_80_cb_b6_92_96_f1_e8_49
S2:f6_22_dc_54_37_98_80_94_c8_ff_14_d8_15_c2_37_b0
Z:cd_7f_fe_22_df_74_09_4f_8b_95_19_29_38_88_cf_e2
End of Initialization (t=0):
H:fa_de_99_df_16_21_15_97_61_19_55_af_c3_2b_c9_6d
L:ff_02_39_e4_75_6c_ad_e4_ab_19_3d_3d_2f_7e_66_5a
N:fa_8c_ba_63_5c_1a_47_89_bf_a1_70_6c_d3_b8_f6_11
S0:ff_2b_84_26_8e_03_eb_41_b6_44_ed_cb_ca_e3_05_fd
S1:1b_fa_a6_27_0f_65_cf_e9_21_db_6d_18_b4_0e_7a_8a
S2:c0_ce_da_06_01_c7_53_08_24_3e_2f_17_14_bc_99_78
Z:3d_0d_c4_75_66_84_5f_70_39_5f_0d_bf_fc_23_40_92
Output Keystream Z's for t=0...15 (also the ciphertext of the 2048-bit all-zero plaintext):
57_9d_a5_0a_db_a2_42_69_bd_6a_37_3a_56_88_5f_67
cd_b3_bd_aa_11_e5_9a_1c_c8_6f_55_ef_b8_41_7d_77
6d_2a_dd_f1_68_f0_cf_cd_45_01_4b_0d_78_13_d8_cf
bc_d5_d5_17_dd_1f_36_52_ad_1d_79_0d_d8_76_5a_64
e6_0f_86_c4_32_58_07_23_d7_d8_69_55_45_47_f7_08
51_10_e6_36_c8_7b_84_d7_ac_b5_64_df_48_2a_cf_7d
ff_3f_76_24_a4_d3_82_84_eb_c3_3a_2f_58_e9_57_73
65_96_71_14_af_15_3d_f0_bb_88_1e_42_65_81_f5_3f
87_bc_9a_eb_c4_4b_d9_c0_58_3c_94_fc_6f_31_d0_02
59_6c_56_5b_1b_71_6b_bd_fa_ec_37_1d_ec_d0_e4_a1
97_5e_c7_58_0c_56_13_68_f4_ee_6a_f4_d8_32_20_3c
54_9b_7b_92_61_02_28_5c_f1_92_e1_9b_fc_be_9a_7c
a3_25_72_6f_8f_55_d4_07_0f_24_ea_9f_24_dd_04_83
e5_02_c7_4f_c5_97_4c_4a_04_14_ec_bf_f0_96_97_17
65_35_51_e3_40_1a_bb_4a_72_4e_63_d2_20_c0_36_4f
e8_be_fd_b6_ea_ce_cb_e6_8a_88_3d_c6_6c_39_4c_65
The 128-bit tag T:
b8_47_2d_8a_51_64_a1_cd_06_2c_d3_77_da_05_88_3c
```

A test vector for `LOL2.0-Double-LFSR1` is presented as follows.

```
key256: 4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b_4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
iv256: 30_62_1b_29_44_f7_cd_9a_b4_6d_3f_5d_a8_3b_0b_48_8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
t= -12:
H:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N0:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N1:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
S0:4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
S1:8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
S2:4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b
S3:30_62_1b_29_44_f7_cd_9a_b4_6d_3f_5d_a8_3b_0b_48
t= -11:
H:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37_33_8d_3c_91_72_79_4a_bd_eb_c8_6d_74_97_0f_bb_07
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N0:50_ee_5f_f2_11_1a_29_de_88_ab_0e_17_f4_6c_d8_64
N1:87_22_b5_a5_5a_46_ab_3b_3a_82_51_3a_e2_e5_15_54
S0:9b_cd_6c_20_ad_d7_15_43_6d_fa_ce_b8_fe_ee_81_70
```

```
S1:83_4d_51_b5_45_3f_63_8b_21_47_46_cb_7c_95_52_72
S2:58_5d_d0_cd_eb_63_90_2b_f3_04_b2_0a_1e_57_0d_a3
S3:db_2a_3f_31_b7_1d_29_ee_ff_ee_c7_0a_2c_30_9f_8e
t= -10:
H:28_98_09_7c_3d_7d_6e_b3_a0_c1_e6_b4_4a_7d_22_91_7a_7e_47_b0_5f_6f_c0_31_1d_57_a2_20_10_14_d2_e1
L:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37_33_8d_3c_91_72_79_4a_bd_eb_c8_6d_74_97_0f_bb_07
N0:81_d5_51_d8_90_b8_f9_af_63_ad_1e_50_c1_63_31_b7
N1:37_d3_df_5b_65_32_88_ce_52_6f_f0_b0_f1_c8_ae_9a
S0:93_59_1b_c5_36_0c_f3_df_c1_dd_36_c4_87_62_17_9d
S1:51_08_1c_6c_6b_61_9d_5b_1c_c5_ff_fe_22_6f_08_4c
S2:28_64_26_2d_d5_97_fb_af_d1_64_04_74_b8_2d_be_5e
S3:eb_9f_d3_29_8e_3d_d8_d1_9e_e7_c6_5a_2a_5e_87_52
t= -9:
H:29_7a_56_62_bf_bd_90_23_b2_e4_24_03_07_6a_55_cb_65_bb_d7_f6_b0_30_94_df_02_27_a0_72_e0_f3_83_27
L:28_98_09_7c_3d_7d_6e_b3_a0_c1_e6_b4_4a_7d_22_91_7a_7e_47_b0_5f_6f_c0_31_1d_57_a2_20_10_14_d2_e1
N0:d9_40_67_ef_02_9d_e8_2e_db_0a_28_e3_79_9b_b6_5c
N1:58_ab_ea_6d_e8_fd_dc_87_bc_0f_df_c3_fa_d6_a3_05
S0:5f_61_4d_96_b5_4c_71_42_8c_55_cb_06_c4_00_31_cd
S1:9b_c8_1c_f4_27_97_db_44_d0_60_a8_0d_98_ac_99_dd
S2:39_d6_e7_b5_87_c5_6c_59_a9_ff_68_d4_6d_21_b5_1b
S3:73_6a_8a_2b_56_5a_19_c7_a7_48_e3_5b_d0_a3_97_53
t= -8:
H:e6_5a_c9_8e_b7_c9_5e_01_2f_8a_7c_8f_8f_62_a6_9c_10_1f_ed_bf_59_9c_e9_be_b4_22_36_0e_ee_9c_1c_14
L:29_7a_56_62_bf_bd_90_23_b2_e4_24_03_07_6a_55_cb_65_bb_d7_f6_b0_30_94_df_02_27_a0_72_e0_f3_83_27
N0:dc_e0_41_e1_55_86_3a_86_6b_76_ce_3f_1a_6b_15_86
N1:29_9b_02_0d_c0_29_aa_88_36_53_42_2e_ba_c9_bf_b7
S0:a7_98_6c_eb_38_c0_63_d4_59_79_23_98_82_b9_25_db
S1:ba_18_d3_b4_1b_32_eb_25_15_4d_ac_de_f7_16_ae_7b
S2:75_06_b4_7e_8d_23_98_0a_b8_99_f1_6b_f2_bb_2b_f4
S3:fa_7f_c3_ff_c4_8a_2e_c3_a1_e3_36_f1_fd_28_51_6d
t= -7:
H:bf_68_86_8d_94_21_79_46_c7_74_e0_dc_a9_1f_a8_68_1c_ca_f6_61_6b_01_56_80_6d_fb_7f_84_48_92_63_b6
L:e6_5a_c9_8e_b7_c9_5e_01_2f_8a_7c_8f_8f_62_a6_9c_10_1f_ed_bf_59_9c_e9_be_b4_22_36_0e_ee_9c_1c_14
N0:5c_5e_c2_8d_82_b2_ab_19_0c_0f_f7_e8_64_87_d9_52
N1:f0_e0_81_1e_05_de_50_d7_92_44_d2_67_76_73_d2_9f
S0:f8_23_f3_71_63_ec_1b_24_1e_fb_47_16_85_f2_39_47
S1:1d_b0_82_8d_48_c0_32_d8_44_49_de_11_03_02_d2_9e
S2:e9_81_3a_ee_d3_c6_2b_26_9f_84_c3_5e_2a_07_38_4e
S3:51_5d_bd_b9_d2_1b_01_07_00_84_fd_23_81_c6_40_4b
t= -6:
H:87_35_50_46_52_b3_d3_ce_c0_db_41_e0_f5_23_9f_d9_45_81_d7_e8_c3_2e_19_3b_87_77_20_2c_98_f4_d9_c3
L:bf_68_86_8d_94_21_79_46_c7_74_e0_dc_a9_1f_a8_68_1c_ca_f6_61_6b_01_56_80_6d_fb_7f_84_48_92_63_b6
N0:07_05_db_11_82_f9_b3_79_0c_b3_44_26_72_95_75_5c
N1:2e_10_78_dc_bb_03_65_05_b3_1b_74_f5_94_75_22_5c
S0:05_cc_3d_f4_90_d6_c4_f2_fb_03_35_6d_b0_6e_b8_69
S1:13_09_ef_12_72_64_ec_58_31_4f_21_c4_14_46_21_b1
S2:d9_84_6f_30_f1_6a_7e_a6_36_7f_d3_f7_94_e1_bb_d2
S3:bd_66_1c_d9_38_8b_62_77_c4_01_ad_72_56_96_ea_cd
t= -5:
H:96_f1_6f_1c_9d_23_9b_a8_ad_90_33_ad_ab_1b_74_75_d0_43_73_d5_3b_d3_de_1a_0f_3d_bf_3f_96_f6_9a_66
L:87_35_50_46_52_b3_d3_ce_c0_db_41_e0_f5_23_9f_d9_45_81_d7_e8_c3_2e_19_3b_87_77_20_2c_98_f4_d9_c3
N0:5a_fb_92_68_fd_fd_71_be_ea_83_3b_8b_ce_cd_1b_81
N1:72_9b_04_80_55_4d_cf_29_d4_7f_e3_31_a3_4c_46_90
S0:5c_1d_f6_aa_03_ce_66_62_64_95_81_26_79_42_c0_03
S1:1e_8c_65_d8_54_fe_9e_a8_f7_cb_1f_34_b2_63_85_64
S2:5e_11_1d_9d_c7_ce_df_d2_72_d0_de_f3_15_75_0c_d9
S3:d0_cf_1b_87_0e_e3_1b_21_94_42_fb_d1_9c_1e_a0_21
t= -4:
H:78_8b_20_cb_47_71_53_b7_e0_7e_9d_fd_ff_e2_77_20_df_dc_d2_9d_42_49_db_b0_c3_3d_9c_8e_d4_37_76_42
L:96_f1_6f_1c_9d_23_9b_a8_ad_90_33_ad_ab_1b_74_75_d0_43_73_d5_3b_d3_de_1a_0f_3d_bf_3f_96_f6_9a_66
N0:7b_94_dc_86_b9_33_e5_31_24_c2_2f_43_e2_55_b8_3c
N1:c5_d9_91_1c_88_db_5b_7d_31_39_da_90_2a_9d_c9_60
S0:53_d3_43_93_e5_15_e9_06_f6_83_2b_34_53_47_a0_70
S1:97_43_a3_1f_59_8b_87_4e_15_9d_cf_86_2c_c7_2a_47
S2:72_12_d3_7a_23_2d_3a_8e_78_36_88_85_73_f0_76_85
S3:2b_de_f4_94_73_13_01_7a_2d_25_15_a9_e5_d3_71_bf
t= -3:
H:70_ce_08_e7_65_bd_de_29_89_85_bc_3c_39_06_b8_69_1a_74_34_b8_e3_e1_ad_cb_8b_ef_7c_73_96_95_32_44
L:78_8b_20_cb_47_71_53_b7_e0_7e_9d_fd_ff_e2_77_20_df_dc_d2_9d_42_49_db_b0_c3_3d_9c_8e_d4_37_76_42
N0:fe_ac_34_40_2b_f5_02_cd_20_df_58_cc_14_8e_f5_8b
N1:59_8b_9a_31_b0_1d_4e_17_94_3a_65_c6_58_65_81_1a
S0:74_7c_b5_e7_dd_2b_f7_62_3b_66_f2_27_1a_94_46_de
S1:4d_cc_52_b2_d6_63_65_5c_84_7b_ac_1f_b6_f9_84_59
S2:38_df_05_85_8d_54_24_12_b2_28_d0_10_ce_11_eb_89
S3:2f_5c_77_50_d1_b0_e7_d2_c3_31_56_41_db_eb_46_45
t= -2:
H:ac_20_2f_20_f4_69_54_b7_5d_25_95_08_d4_ad_dd_28_fb_17_62_be_bb_63_66_d9_bd_af_dc_ee_1c_6d_92_e0
L:70_ce_08_e7_65_bd_de_29_89_85_bc_3c_39_06_b8_69_1a_74_34_b8_e3_e1_ad_cb_8b_ef_7c_73_96_95_32_44
N0:51_53_90_c3_75_ad_12_f8_23_de_0d_ab_85_ba_20_8e
N1:de_ab_6f_5e_ba_5d_20_f1_81_6b_cd_7b_b6_fb_e4_8c
S0:d6_71_12_cb_cd_1f_4d_b1_0f_53_fa_07_67_72_50_97
S1:1a_3c_e6_9d_1b_65_fa_b7_ec_02_22_96_d6_39_0a_80
S2:17_ba_ee_ce_ec_d2_ca_3b_6e_51_43_4e_2f_58_53_5c
S3:ea_ad_28_2c_16_54_52_53_23_6c_c2_81_77_75_66_a7
t= -1:
H:db_8b_cf_1c_8c_53_a9_16_b2_02_61_eb_71_04_9b_c4_73_87_36_5f_9c_90_dd_95_4b_ee_b2_7f_aa_ec_6a_db
```

```
L:ac_20_2f_20_f4_69_54_b7_5d_25_95_08_d4_ad_dd_28_fb_17_62_be_bb_63_66_d9_bd_af_dc_ee_1c_6d_92_e0
N0:b7_a8_3c_f1_d0_de_78_a1_d0_b7_10_7e_0a_1d_1a_6d
N1:87_21_52_83_a9_46_21_0f_d7_3e_fe_81_c9_bf_a9_ed
S0:51_b5_02_3d_0a_c7_da_69_59_7f_e4_88_c3_d1_de_8d
S1:e4_6d_ce_06_3d_7e_0d_4a_e9_ed_41_1f_4f_06_75_ca
S2:d0_78_b2_38_65_9e_2f_56_11_c7_cb_4e_81_00_b0_17
S3:d7_3b_c4_12_4c_f0_c1_01_f8_fb_17_a2_fd_5f_2c_58
End of Initialization (t=0):
H:67_cb_bd_1d_51_c9_90_32_e2_ba_72_5b_58_45_69_8d_e9_0b_fb_34_c2_49_8f_0c_98_66_30_fd_27_de_b9_eb
L:db_8b_cf_1c_8c_53_a9_16_b2_02_61_eb_71_04_9b_c4_73_87_36_5f_9c_90_dd_95_4b_ee_b2_7f_aa_ec_6a_db
N0:ce_07_a0_75_60_66_24_3c_21_18_89_2c_68_ce_fb_bc
N1:6c_a7_25_9e_81_24_d0_e6_87_9a_0a_57_af_2a_a6_79
S0:3b_9e_d4_2b_a4_62_59_e6_79_91_b2_6f_40_a1_0f_24
S1:8c_41_d9_da_e9_ca_93_cf_60_bd_61_8e_be_cd_7a_dc
S2:4a_bb_a0_be_27_d0_8b_94_68_89_70_8f_c5_44_82_e0
S3:65_9b_0a_a5_cf_2d_39_c0_38_b6_c1_73_02_36_8f_f9
Output Keystream Z's for t=0...15 (also the ciphertext of the 4096-bit all-zero plaintext):
09_2a_c4_d5_94_90_69_3f_bb_a2_19_bb_29_5e_44_2c_ff_bd_87_73_71_d0_c6_49_23_b1_e8_4e_7b_74_07_3b
93_f7_ec_13_05_af_3d_92_5a_57_46_28_81_1e_78_9f_42_04_ca_a5_33_6f_6c_63_7b_7b_bb_d8_fb_e9_b8_82
89_d8_27_34_a0_0a_11_09_e5_cb_6d_b6_0c_79_99_4c_48_a6_56_e4_2a_f3_9d_82_26_65_7e_70_86_ff_71_92
81_c3_13_4e_f6_d9_94_15_f6_54_07_48_fe_d2_9f_b0_69_7a_53_7b_8a_dc_34_9b_d8_22_4d_19_37_f8_e5_b4
f5_fc_33_2f_91_48_e1_95_b6_00_f8_4e_03_7a_a8_2c_2c_ea_d7_f4_e6_43_35_2d_55_f2_f7_49_af_43_85_d5
67_b4_23_ee_9a_3f_ce_e9_fa_e4_a0_b5_e1_90_a6_01_27_19_38_80_d8_71_ef_b7_11_05_ba_41_25_da_12_39
49_fd_c9_16_bf_e4_b6_4d_03_10_2d_ac_bd_06_8d_af_50_d2_5c_79_36_e8_cc_c0_30_c6_ca_7e_b5_b4_21_fc
55_58_45_31_61_e7_08_8c_8d_80_7a_d5_65_c4_49_f0_9f_97_c4_35_b3_6a_3b_ae_95_cd_d6_84_b5_da_97_bc
f5_24_a6_39_77_9c_0c_b0_a4_f2_0a_7e_ae_a9_fe_f2_43_52_4b_0e_bf_e6_34_c1_d4_b8_68_30_a0_89_a3_3b
ed_48_73_7a_22_2c_86_0d_89_20_a9_23_2d_2d_39_f1_f7_dc_69_59_9a_41_56_85_21_bd_21_e4_9d_24_63_fe
23_ae_4c_5e_0e_63_02_c0_20_e9_6e_bd_37_5a_91_3c_33_7a_37_38_bc_be_0e_15_c9_c7_75_18_65_77_6a_24
ae_a8_b9_4a_e2_15_22_a7_bc_00_19_91_0d_5b_4f_b8_4f_f1_4a_bf_4b_b3_b8_81_04_95_7d_5a_b8_df_bc_7c
f5_89_92_9d_74_57_ab_57_18_84_4a_03_20_f8_3a_09_1b_63_06_5c_99_53_f3_05_71_3c_b7_eb_62_2d_23_5f
2d_bc_46_79_71_21_b4_2c_ac_39_ac_61_92_f0_db_2b_87_31_47_1f_71_2f_4c_4e_66_40_0b_d5_06_b6_70_10
86_bb_4b_2a_13_4d_bb_e5_b5_4c_13_ce_0a_84_83_f6_36_38_7f_ca_b8_80_18_ea_3e_ac_36_99_42_c5_18_61
cd_f7_c5_ec_65_e3_a4_43_55_41_19_61_bc_90_fb_16_79_59_28_58_ec_0d_39_f8_48_e4_28_59_fb_04_6e_ba
The 128-bit tag T:
8d_0c_0e_d0_33_79_9f_05_0a_58_46_a8_7f_66_f9_99
```

A test vector for `LOL2.0-Double-LFSR2` is presented as follows.

```
key256: 4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b_4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
iv256: 30_62_1b_29_44_f7_cd_9a_b4_6d_3f_5d_a8_3b_0b_48_8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
t= -12:
H:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N0:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N1:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
S0:4e_16_c0_2b_32_93_58_ad_31_19_c4_94_1d_15_85_27
S1:8e_fd_d1_19_97_f0_b5_f1_39_76_dd_d2_ad_97_f6_26
S2:4e_38_40_08_81_c1_dc_aa_87_68_df_de_63_49_f5_9b
S3:30_62_1b_29_44_f7_cd_9a_b4_6d_3f_5d_a8_3b_0b_48
t= -11:
H:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37_33_8d_3c_91_72_79_4a_bd_eb_c8_6d_74_97_0f_bb_07
L:00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00_00
N0:50_ee_5f_f2_11_1a_29_de_88_ab_0e_17_f4_6c_d8_64
N1:87_22_b5_a5_5a_46_ab_3b_3a_82_51_3a_e2_e5_15_54
S0:9b_cd_6c_20_ad_d7_15_43_6d_fa_ce_b8_fe_ee_81_70
S1:83_4d_51_b5_45_3f_63_8b_21_47_46_cb_7c_95_52_72
S2:58_5d_d0_cd_eb_63_90_2b_f3_04_b2_0a_1e_57_0d_a3
S3:db_2a_3f_31_b7_1d_29_ee_ff_ee_c7_0a_2c_30_9f_8e
t= -10:
H:98_49_e3_4a_6b_8e_28_0d_2f_25_82_cf_36_b2_08_1f_6c_c3_ac_b2_f4_b4_54_61_83_dd_d6_48_4a_d6_93_ca
L:e4_41_d6_c6_39_25_c8_58_59_e1_32_59_81_86_76_37_33_8d_3c_91_72_79_4a_bd_eb_c8_6d_74_97_0f_bb_07
N0:81_d5_51_d8_90_b8_f9_af_63_ad_1e_50_c1_63_31_b7
N1:37_d3_df_5b_65_32_88_ce_52_6f_f0_b0_f1_c8_ae_9a
S0:85_e4_f0_c7_9d_d7_67_8f_5f_57_42_ac_dd_a0_56_b6
S1:51_08_1c_6c_6b_61_9d_5b_1c_c5_ff_fe_22_6f_08_4c
S2:98_b5_cc_1b_83_64_bd_11_5e_80_60_0f_c4_e2_94_d0
S3:eb_9f_d3_29_8e_3d_d8_d1_9e_e7_c6_5a_2a_5e_87_52
t= -9:
H:71_76_47_17_b4_82_4d_e5_a5_3d_e6_e3_93_4a_13_09_09_32_ce_d6_98_64_b7_7d_3a_86_f9_83_9a_09_11_63
L:98_49_e3_4a_6b_8e_28_0d_2f_25_82_cf_36_b2_08_1f_6c_c3_ac_b2_f4_b4_54_61_83_dd_d6_48_4a_d6_93_ca
N0:d9_40_67_ef_02_9d_e8_2e_db_0a_28_e3_79_9b_b6_5c
N1:58_ab_ea_6d_e8_fd_dc_87_bc_0f_df_c3_fa_d6_a3_05
S0:25_55_bf_b4_36_c3_c6_b0_2a_7e_e6_9f_e4_38_e2_a2
S1:56_2b_10_36_17_0f_19_1d_c1_5b_05_fb_e7_b3_9b_3d
S2:d1_0b_1c_f6_da_09_f7_21_31_c2_ce_4f_85_ce_d9_57
S3:2d_bc_98_2b_70_37_27_d3_93_7e_3c_f1_a5_b2_cf_dc
t= -8:
H:ca_9f_0f_77_c2_6f_1a_8b_65_d3_1c_f4_e3_a8_f1_45_f0_bd_2a_4c_ac_35_95_72_df_69_60_e9_ca_30_b8_fb
L:71_76_47_17_b4_82_4d_e5_a5_3d_e6_e3_93_4a_13_09_09_32_ce_d6_98_64_b7_7d_3a_86_f9_83_9a_09_11_63
N0:53_80_ad_9a_7d_94_0d_1b_c0_d1_e3_5e_37_54_5d_0a
N1:46_57_68_18_23_29_b6_d4_f7_19_27_3d_e7_7e_74_99
S0:dc_b3_42_04_28_56_1c_89_81_e0_76_fd_3c_f0_e4_4c
S1:bb_cc_bd_72_fb_da_02_32_2d_6c_db_04_77_1a_eb_d1
S2:d0_ce_40_48_70_a8_19_71_9c_b9_cd_34_60_ba_e2_ad
S3:55_78_88_82_ee_09_b8_c1_72_66_82_e6_be_22_a4_d3
t= -7:
H:ec_37_25_47_3a_1a_2d_02_5f_8f_72_f0_09_db_5e_57_ac_46_43_55_21_16_73_e7_cd_8b_44_71_a5_7d_d0_6b
```

```
L:ca_9f_0f_77_c2_6f_1a_8b_65_d3_1c_f4_e3_a8_f1_45_f0_bd_2a_4c_ac_35_95_72_df_69_60_e9_ca_30_b8_fb
N0:41_3e_50_4f_e9_8c_36_36_b3_c2_03_33_6f_98_37_5f
N1:8d_7b_7f_a5_0c_45_83_5e_f6_2a_45_3a_7c_e1_35_c3
S0:25_78_96_48_34_6f_19_fe_f1_e5_f3_25_8b_14_17_b4
S1:56_d2_91_e3_b1_46_7e_93_ed_58_f0_95_92_e3_fc_fc
S2:12_8a_55_c6_27_dc_a1_a5_df_9f_09_c8_26_0f_40_5c
S3:cf_ed_5a_92_4a_29_d0_ad_be_cc_7f_ea_85_05_7d_8c
t= -6:
H:c5_25_55_c2_d7_6d_aa_c0_11_2e_00_7b_b1_0f_ad_97_47_59_c5_51_76_83_48_a2_fb_e9_33_06_06_ce_c2_e2
L:ec_37_25_47_3a_1a_2d_02_5f_8f_72_f0_09_db_5e_57_ac_46_43_55_21_16_73_e7_cd_8b_44_71_a5_7d_d0_6b
N0:be_28_af_9d_24_e1_1a_a4_4f_dc_dd_72_31_2d_b0_de
N1:72_d5_8b_da_da_7f_09_ef_93_e4_dc_04_22_00_68_47
S0:67_fe_d2_e9_30_04_b8_2d_4c_36_9b_4d_96_a6_5b_d5
S1:b9_f1_50_8d_54_7a_4a_0e_be_97_9e_0d_c5_69_b7_7d
S2:d2_b5_33_1b_02_0a_3b_2f_5b_23_a6_d5_95_0b_b2_54
S3:59_b2_46_ed_ed_86_31_56_24_e8_86_fe_55_19_6d_54
t= -5:
H:cb_61_43_29_97_d7_b4_6a_7b_0d_5d_36_c7_85_e5_46_46_52_99_25_10_bb_08_13_66_16_b9_10_de_33_dc_2e
L:c5_25_55_c2_d7_6d_aa_c0_11_2e_00_7b_b1_0f_ad_97_47_59_c5_51_76_83_48_a2_fb_e9_33_06_06_ce_c2_e2
N0:9d_3f_f0_28_fa_38_0c_89_ce_d7_38_39_d6_4a_fb_f1
N1:a7_4a_15_33_2b_14_62_d9_77_68_17_fc_33_a8_9a_72
S0:7f_4f_91_75_62_e0_de_8f_bc_4d_8d_6b_ad_d6_52_11
S1:af_d0_37_fb_e4_24_a6_6c_6c_de_e3_4f_37_15_14_30
S2:11_27_69_76_2c_91_b5_88_37_e7_1b_22_11_28_73_c5
S3:4c_ff_9a_a0_48_4d_fd_30_42_c6_14_3a_07_17_19_aa
t= -4:
H:cc_f4_55_7f_b2_ef_e6_d9_83_e4_76_05_45_b5_7c_6c_8d_44_3c_23_fd_3e_af_1b_b2_ae_e3_07_85_bc_00_c8
L:cb_61_43_29_97_d7_b4_6a_7b_0d_5d_36_c7_85_e5_46_46_52_99_25_10_bb_08_13_66_16_b9_10_de_33_dc_2e
N0:4f_bc_8f_d1_39_bd_7e_73_04_16_18_3c_ab_83_c2_ba
N1:2e_2e_52_1a_6f_a6_34_bc_ea_94_87_16_47_fd_3a_19
S0:a5_45_21_e5_42_dd_23_1c_84_df_19_20_56_35_64_66
S1:bf_75_3e_93_cc_8e_a8_81_45_0b_a4_87_a9_7e_c6_fb
S2:d5_6a_f3_7c_7a_f2_2a_0e_47_68_49_6d_df_6c_94_8f
S3:fd_e8_71_bf_fd_ad_16_1d_ea_9e_fb_2d_c7_9a_e0_14
t= -3:
H:0e_d2_7f_14_2d_99_b6_96_ec_3a_9f_fc_d7_56_52_9b_35_ed_5a_66_65_93_3c_3a_4f_e6_36_3d_21_ab_e3_57
L:cc_f4_55_7f_b2_ef_e6_d9_83_e4_76_05_45_b5_7c_6c_8d_44_3c_23_fd_3e_af_1b_b2_ae_e3_07_85_bc_00_c8
N0:d7_88_ae_37_65_f9_15_cc_81_7d_24_b2_3f_48_e6_8d
N1:34_b1_d3_03_c2_8c_5c_57_e9_6f_a2_63_fb_33_5a_47
S0:71_fc_29_ba_2e_66_2f_59_8a_51_c8_52_35_1a_2c_55
S1:4a_bc_f4_4d_a6_39_2b_77_b6_8e_5c_47_e5_0a_18_3d
S2:09_b2_7d_d9_5c_23_07_87_5d_7a_52_f2_f8_bb_a2_e4
S3:05_25_30_66_1d_50_f7_b7_6e_fa_8e_fd_e6_de_66_35
t= -2:
H:88_6e_54_fc_6c_8b_f4_99_71_b9_c5_cc_83_da_41_6f_a1_83_56_7b_da_78_e7_a5_83_43_6a_2d_f0_84_0b_e8
L:0e_d2_7f_14_2d_99_b6_96_ec_3a_9f_fc_d7_56_52_9b_35_ed_5a_66_65_93_3c_3a_4f_e6_36_3d_21_ab_e3_57
N0:60_51_61_c8_90_3b_98_5e_9b_0c_14_80_f2_cd_35_74
N1:cc_d0_e6_fe_d5_24_92_24_1c_22_ee_2e_50_80_4b_bf
S0:4f_50_bd_87_f4_a5_d8_d6_4b_76_1b_0e_20_18_fd_15
S1:8f_d3_bb_cd_1b_8a_32_d2_24_41_2b_5e_7c_86_d9_6d
S2:45_81_43_3d_06_4e_32_ec_7e_b8_27_bc_42_4e_01_e8
S3:2e_5b_4e_d2_a9_a3_6c_25_9e_1b_8e_d6_79_f8_64_c0
t= -1:
H:02_5b_42_7a_b0_1c_21_7c_4b_59_ef_03_aa_78_4d_38_1c_2d_e5_64_54_ee_6a_97_03_2d_e8_00_a2_c1_0f_02
L:88_6e_54_fc_6c_8b_f4_99_71_b9_c5_cc_83_da_41_6f_a1_83_56_7b_da_78_e7_a5_83_43_6a_2d_f0_84_0b_e8
N0:ae_a9_e3_7b_73_77_2d_99_bd_df_5e_1d_2b_a1_33_84
N1:1d_49_0d_68_38_af_0a_98_6c_0d_4d_e3_0c_e7_ac_00
S0:1a_0d_ab_3c_c8_ee_cd_b0_6b_e7_7e_9b_72_6b_5e_1c
S1:bf_3a_f3_98_b0_89_6f_b5_85_ef_00_54_2d_eb_4b_b7
S2:eb_5f_9f_d2_0f_54_56_52_69_36_c6_13_7c_7f_fa_57
S3:89_69_4a_1b_28_27_14_50_46_31_95_59_f0_43_63_e3
End of Initialization (t=0):
H:7c_35_eb_54_ce_ac_a0_4a_ca_b0_4f_d7_ad_37_b1_c7_61_d1_dc_64_e8_5d_ef_0b_73_b3_2f_bd_76_2b_62_c0
L:02_5b_42_7a_b0_1c_21_7c_4b_59_ef_03_aa_78_4d_38_1c_2d_e5_64_54_ee_6a_97_03_2d_e8_00_a2_c1_0f_02
N0:d7_12_cd_3a_a6_a2_dd_9e_34_a8_cb_7e_a8_c3_60_37
N1:07_da_b1_20_81_5e_d8_2a_b5_68_7b_58_28_60_b0_cc
S0:db_c0_8f_f7_25_1f_3a_8e_b1_35_51_ed_52_7e_dc_54
S1:9e_99_e7_c1_61_73_b2_c6_53_33_57_37_16_29_f0_10
S2:0c_45_8d_a1_42_7a_57_6a_36_19_f2_26_6a_18_78_4a
S3:89_61_dc_6e_c8_53_2d_0d_9f_48_cb_b4_40_cc_c2_5b
Output Keystream Z's for t=0...15 (also the ciphertext of the 4096-bit all-zero plaintext):
ab_7b_74_9b_db_a6_ac_04_83_06_b0_f0_2b_05_de_36_ae_4c_d3_1b_f8_11_5c_48_b6_46_6f_d0_f9_b1_48_67
c8_f9_b4_dc_7e_28_00_6a_ae_b9_35_01_d9_84_74_b2_a1_68_98_36_03_d0_e8_20_96_b6_09_90_66_06_7f_fb
c1_5e_b0_2b_2b_a3_99_75_09_d0_36_f2_1a_0a_48_f5_52_28_0b_0e_f3_3e_32_45_6b_04_df_37_db_1f_50_85
db_22_3e_b7_81_87_c9_35_c4_3b_df_ec_1f_af_39_0f_b2_69_10_88_02_aa_52_73_7e_6a_17_d1_9e_95_30_56
24_29_56_dc_bf_34_87_06_c4_ce_8a_0d_65_36_86_52_0c_30_34_d9_7f_88_9f_73_ae_f9_e5_2c_c8_5e_2e_bf
7b_8b_7e_86_f7_64_9f_f0_2d_ae_a0_ca_3d_9c_f0_67_69_b3_5c_cf_6f_00_d1_a9_e9_38_f0_fa_9c_2f_9d_84
30_f1_b5_15_5c_56_4c_8e_29_5b_4e_e9_82_96_5c_af_25_d9_83_d4_fd_2f_c8_11_c0_2b_ab_23_48_e3_aa_11
fa_7b_fb_74_6c_54_52_b9_aa_b7_28_6d_9d_5e_1d_82_1f_cc_15_77_28_47_86_ef_78_3f_b5_07_21_d5_59_d7
f6_6d_27_d3_ba_2e_fb_fb_c1_69_5e_1d_2e_b2_84_3c_e7_a6_71_fc_90_6f_3b_a9_4f_6f_d9_c6_44_0e_2d_6a
30_ce_9e_1c_a8_b4_06_b2_f8_fd_b9_22_ce_86_c0_70_df_83_3f_b6_36_bb_87_c6_eb_cd_2a_67_22_1a_98_b2
07_78_e9_d5_cd_5c_1d_05_55_c0_8c_7c_02_1b_21_bc_52_c5_a8_09_47_e0_57_72_71_71_68_38_ed_a5_03_26
ab_77_d9_40_95_c2_84_06_86_f9_e9_96_b7_e2_6e_be_95_be_3a_e0_14_e4_47_f4_29_5d_fe_b6_65_f5_9e_c0
79_76_f6_29_30_be_ce_78_d1_c9_68_7f_f4_21_1d_7e_de_4b_99_b9_5a_ea_73_06_3b_63_74_80_4e_9e_16_7b
b7_7e_00_47_95_33_41_ff_fa_01_67_1d_7f_a3_97_7a_81_bc_88_a8_5b_76_16_43_a0_ed_18_59_2c_08_80_ef
fe_0a_a4_82_58_52_de_bf_2f_f2_00_59_77_da_c4_03_b3_1d_3c_2f_42_c0_87_83_c6_aa_c0_35_3f_4d_ff_0e
```

```
9e_12_55_8f_35_99_23_95_26_fc_b9_cb_cf_a3_6c_10_27_b3_b2_f4_85_fd_b7_d2_81_96_dc_a4_cf_59_38_83
The 128-bit tag T:
1f_2c_fc_21_4e_94_b4_ec_8e_29_4c_36_21_62_3a_44
```

Appendix D
LOL2.0-Mini and LOL2.0-Double with Standard AEAD Scheme GCM

We consider the AEAD mode based on universal hash functions GHASH based on which we can transform the stream ciphers LOL2.0-Mini and LOL2.0-Double to AEAD schemes. GHASH involves the arithmetic operations addition ($\oplus$) and multiplication ($\cdot$) over the finite field $\mathbb{F}_{2^{128}} = \mathbb{F}_2[x]/g(x)$ where $g(x)$ is the 128-degree low-hamming-weight irreducible polynomial

$$g(x) = x^{128} + x^7 + x^2 + x + 1.$$

In the GHASH based AEAD modes, the keystream bits of the stream cipher are divided into 128-bit blocks denoted as $Z^0, Z^1, \ldots$. The general process of the GHASH based AEAD mode is defined as Algorithm 7. As can be seen, the 1st keystream block Z^0 is used as the key H of GHASH and the 2-nd keystream block Z^1 is used as as the final masking Γ_T for the tag. The subsequent keystream blocks $Z^2, \ldots, Z^{\ell+1}$ are used to encrypt the plaintext blocks $P^1, \ldots, P^\ell$ and generate the ciphertext blocks $C^1, \ldots, C^\ell$, feeding into GHASH for computing the final tag T. More details on message/associate data padding rules as well as other restrictions can be found in [32].

Algorithm 7: The Ghash based AEAD scheme

```
procedure GCMEncrypt($(K, IV, AD, M)$)
    Pad $AD$ as $\bar{AD} = \bar{AD}_0 \| \dots \| \bar{AD}_{\ell-1}$
    Pad $M$ as $\bar{M} = \bar{M}_0 \| \dots \| \bar{M}_{m-1}$ and define the values $\mu \leftarrow (|M| \mod 128)$ and $\nu \leftarrow \lceil \frac{|M|}{128} \rceil$
    $\boldsymbol{S} \leftarrow$ Load$(K, IV)$
    $\boldsymbol{S} \leftarrow$ scInit$(\boldsymbol{S})$
    $(H, \boldsymbol{S}) \leftarrow$ scOut$(\boldsymbol{S})$
    $(\Gamma_T, \boldsymbol{S}) \leftarrow$ scOut$(\boldsymbol{S})$
    Initialize $S_a^* \leftarrow 0^{128}$
    Define the 128-bit $\Theta \leftarrow |AD| \| |M|$
    if $|AD| > 0$ then
        for $i = 0, \dots, \ell - 1$ do
            $S_a^* \leftarrow H \cdot \bar{AD}_i \oplus S_a^*$
    if $|M| > 0$ then
        for $i = 0, \dots, m - 1$ do
            $(Z_i, \boldsymbol{S}) \leftarrow$ scOut$(\boldsymbol{S})$
            $C_i \leftarrow \bar{M}_i \oplus Z_i$
            $S_a^* \leftarrow H \cdot C_i \oplus S_a^*$
        $C_{\nu-1} \leftarrow$ trunc$(C_{\nu-1}, \mu)$
    $S_a^* \leftarrow H \cdot \Theta \oplus S_a^*$
    Compute $T \leftarrow S_a^* \oplus \Gamma_T$
    return $(C, T)$ where $C = C_0 \| \dots \| C_{\nu-1}$
procedure GCMDecrypt($(K, IV, AD, C)$)
    Pad $AD$ as $\bar{AD} = \bar{AD}_0 \| \dots \| \bar{AD}_{\ell-1}$
    Pad $C$ as $\bar{C} = \bar{C}_0 \| \dots \| \bar{C}_{m-1}$ and the values $\mu \leftarrow (|C| \mod 128)$ and $\nu \leftarrow \lceil \frac{|C|}{128} \rceil$
    $\boldsymbol{S} \leftarrow$ Load$(K, IV)$
    $\boldsymbol{S} \leftarrow$ scInit$(\boldsymbol{S})$
    $(H, \boldsymbol{S}) \leftarrow$ scOut$(\boldsymbol{S})$
    $(\Gamma_T, \boldsymbol{S}) \leftarrow$ scOut$(\boldsymbol{S})$
    Initialize $S_a^* \leftarrow 0^{128}$
    Define the 128-bit $\Theta \leftarrow |AD| \| |C|$
    if $|AD| > 0$ then
        for $i = 0, \dots, \ell - 1$ do
            $S_a^* \leftarrow H \cdot \bar{AD}_i \oplus S_a^*$
    if $|C| > 0$ then
        for $i = 0, \dots, m - 1$ do
            $(Z_i, \boldsymbol{S}) \leftarrow$ scOut$(\boldsymbol{S})$
            $M_i \leftarrow \bar{C}_i \oplus Z_i$
            $S_a^* \leftarrow H \cdot C_i \oplus S_a^*$
        $M_{\nu-1} \leftarrow$ trunc$(M_{\nu-1}, \mu)$
    $S_a^* \leftarrow H \cdot \Theta \oplus S_a^*$
    Compute $T' \leftarrow S_a^* \oplus \Gamma_T$
    return $M = M_0 \| \dots \| M_{\nu-1}$ when $T = T'$ and $\perp$ otherwise
```