`Simple`: a simple AEAD scheme

A submission to the NIST Lightweight Cryptography Standardization Process

Shay Gueron [1,2] and Yehuda Lindell [3,4]

[1]University of Haifa, Israel, [2]Amazon Web Services, USA [3] Bar Ilan University, Israel
[4] Unbound Tech Ltd. , Israel

March 30, 2019 (at 12:09 Noon)

Submitters: Shay Gueron and Yehuda Lindell. There are no auxiliary submitters.

Inventors/Developers: Shay Gueron and Yehuda Lindell.

Implementation Owners: Submitters.

Email Address (preferred): shay@math.haifa.ac.il

Postal Address and Telephone (if absolutely necessary):
Shay Gueron, University of Haifa, Haifa, Israel, +972 (50) 5638806.

Signature: x. See also printed version of "Statement by Each Submitter".

Version: 0.1

Release Date: February 23, 2019

Table of Contents

1 Introduction

This submission describes the AEAD scheme called `Simple`, that receives input triples (N, A, M) where A is a header (Additional Authenticated Data) to be authenticated, M is a message to be encrypted and authenticated, and N is a nonce. It processes the input under the key K, and outputs ciphertext C and an authentication tag Tag. N, A, M, C, Tag are strings of bits. In practice, it may be assumed that their lengths are divisible by 8, so that they could be viewed as strings of bytes. `Simple` is nonce respecting, i.e., security is preserved as long as the nonce N is not repeated during the lifetime of the key.

`Simple128` is designed for the use case of lightweight devices where: a) keys are hard to change and therefore need to remain highly secure for a large number of messages; b) a simple design is sought; c) it is desired to have instantiations with lightweight underlying block ciphers E.

In fact, `Simple` is a mode of operation that can use any underlying block cipher E with block size denoted by n and key size denoted by κ. It has two flavors, namely `Simple128` for the case where $\kappa = n$ and `Simple64` for the case where $\kappa = 2n$. Specific instantiations of `Simple128` use $n = 128$, $\kappa = 128$, nonce size of 120 bits, and the choice of `GIFT` ([5], [6]) and `AES128` ([2]) for the lightweight block cipher E. Specific instantiations of `Simple64` use $n = 64$, $\kappa = 128$, nonce size of 56 bits, and the choice of `GIFT` ([5], [6]) or `PRESENT` ([13]) for the lightweight block cipher E.

The following table summarizes the proposals and some of their characteristics.

Scheme's name	κ	n	δ	τ	$\lvert N\rvert$	A_{max}	M_{max}	D_{max}	Q_{max}	Nonce usage
	128	128	64	8	120	$2^{53}-8$	$2^{53}-8$	$2^{53}-8$	2^{46}	non-repeating
`Simple128-GIFT`										
`Simple128-AES128`										
`Simple128-Speck`										
	128	64	32	8	56	2^{10}	2^{10}	2^{40}	2^{30}	non-repeating
`Simple64-GIFT`										
`Simple64-PRESENT`										
`Simple64-Speck`										

Table 1. Summary of the proposals with their characteristics and parameters. The highlighted row is the **primary** submission. The key length (κ), block length (n), nonce length ($|N|$) and the parameters which determine the effective `IV` size (τ, δ) are given in bits. The maximal lengths of the header (A_{max}), message (M_{max}) and the total amount of data allowed to be processed with the same key (D_{max}) are given in bits. The maximal number of messages that can be sent, using a given key is Q_{max}. The nonce values need to be non-repeating during the lifetime of a key. The labels "`GIFT`"/"`AES128`"/"`PRESENT`" indicate the underlying block ciphers (and matching block/key lengths) that are used for the specific instantiation of the mode of operation. Some additional combinations are discussed below.

2 Preliminaries

2.1 Notation and conventions

Strings of bits. This document deals with strings of bits (strings for short). The length of a string S is denoted by $|S|$, where $S > 0$ for all nonempty strings. The empty string is denoted by the symbol $\perp$, which is also used as a signal for failure. The symbol $\oplus$ denotes the bit-wise XOR operation and the symbol $\|$ denotes concatenation of strings. By convention, strings are written in "Little Endian" orientation. If S is a string of length $|S| = L$ its bits are written as $(S =)\ s_{L-1}s_{L-2}\dots s_0$ such that the least significant bit (s_0) is in the rightmost position and the most significant bit (s_{L-1}) is in the leftmost position. A string of k repeated zero bits is denoted by 0^k. The right-shift of the string $S = s_{L-1}\dots, s_0$ by θ positions ($\theta \le L$) is the string $0^\theta \| s_{L-1}\dots s_\theta$ and is denoted by $S \gg \theta$.

Encoding of integers as strings. For integers p, k such that $0 \le p < 2^k$, $p_{[\#\mathbf{k}]}$ denotes the k-bit (string) binary representation of the integer p. For example $19_{[\#\mathbf{8}]} = 00010011$.

Blocks and block ciphers. Hereafter, E is used for denoting a block cipher with block size of n bits and key size of $\kappa \ge n$ bits. For all the cases discussed here, n and κ are powers of 2, and either $\kappa = n$ or $\kappa = 2n$. A string of n bits is called a block. The encryption of the plaintext block P under the key K is denoted by $E(K, P)$. Let $B = b_{n-1}\dots b_0$ be a block. Then the notation $[B2, B1] = B$ indicates that $B2$ and $B1$ are strings of $n/2$ bits defined by $B2 = B_{n-1}\dots B_{n/2}$, $B1 = B_{n/2-1}\dots B_0$. They are also called "half blocks". For an integer $1 \le r < n$ the truncation of B to r bits is the string of r bits $b_{r-1}\dots b_0$, and is denoted by $Truncate(B, r)$.

Parsing a string of bits as blocks. Let $V = p_{v-1}p_{v-2}\dots p_0$ be a nonempty string of v bits where v is divisible by n, i.e., $v = \xi \times n$ for some positive integer ξ. Then, V can be parsed as a sequence of ξ blocks. It is written as $\bar{V}_{\xi-1} \| \dots \| \bar{V}_0$. Alternatively, V can be represented as a list (sequence of blocks) $V_1, \dots, V_\xi$ (with indexes increasing from left to right), where $\bar{V}_j = V_{j+1} = p_{jn+n-1}p_{jn+n-2}\dots p_{jn}$, $j = 0, \dots, \xi - 1$.

Example 2.1. Consider $v = 6$, $n = 2$, $\xi = 3$ and $V = p_5p_4p_3p_2p_1p_0$. Then $V = \bar{V}_2 \| \bar{V}_1 \| \bar{V}_0$ where $\bar{V}_2 = p_5p_4$, $\bar{V}_1 = p_3p_2$, $\bar{V}_0 = p_1p_0$. V can be written as a list (sequence) of blocks $V = V_1, V_2, V_3$ (increasing indexes) where $V_1 = \bar{V}_0$, $V_2 = \bar{V}_1$, $V_3 = \bar{V}_2$.

The (mandatory) $\mathtt{pad10^*}(\ )$ padding. Let Y be a string of bits. The mandatory padding of Y is the string $\mathtt{pad10^*}(Y)$ generated by first appending the bit 1 and then zero padding the result.

Example 2.2. Take $n = 8$ and $Y = 1010101010$ with $y = 10$ bits. Then, $\mathtt{pad10^*}(Y) = 0000011010101010$; parsed as 2 blocks $\mathtt{pad10^*}(Y) = Y_1, Y_2 = 10101010, 00000110$. For $Y = 10101010$ with $y = 8$ bits we have $\mathtt{pad10^*}(Y) = 0000000110101010$, parsed as $\mathtt{pad10^*}(Y) = Y_1, Y_2 = 10101010, 00000001$. For $Y = 1010$ with $y = 4$ bits we have $\mathtt{pad10^*}(Y) = 00011010$, and it is parsed as $\mathtt{pad10^*}(Y) = Y_1 = 00011010$. For $Y = \perp$ we have $\mathtt{pad10^*}(Y) = 00000001$, and it is parsed as $\mathtt{pad10^*}(Y) = Y_1 = 00000001$.

Remark 2.1. $\mathtt{pad10^*}(S)$ padding always modifies the string S (hence the label "mandatory"). It is always a nonempty string, and in particular, consists of at least one block. If $|S|$ is divisible by n, then one block is added to S in order to generate $\mathtt{pad10^*}(S)$.

2.2 The building blocks CBCMAC-IV, CTRENC, and CENC

CBCMAC-IV. Let X be a nonempty sequence of $x > 0$ blocks, $X = X_1, \ldots, X_x$, let R be a block, and let K be a key. Define the following sequence of blocks: $T_0 = R$ and $T_i = E(K, X_i \oplus T_{i-1})$ for $i = 1, \ldots, x$. The last block T_x is also referred to as an authentication tag (for X). The CBCMAC-IV of X under the key K and the IV R is denoted by $\texttt{CBCMAC-IV}(K, R, X)$ and is defined to be the block T_x. The input X is also referred to as a "message".

Remark 2.2. CBCMAC-IV is defined only for sequences (messages) of full blocks, consisting of at least one block.

Remark 2.3. The standard basic CBC-MAC is a special case of CBCMAC-IV where $R = 0^n$, i.e., $\texttt{CBCMAC-IV}(K, 0^n, X)$. Alternatively, CBCMAC-IV of the message X can be viewed as the standard CBC-MAC applied to $X' = (X_1 \oplus R), X_2, \ldots, X_x$. In other words, $\texttt{CBCMAC-IV}(K, R, X) = \text{CBC-MAC}(K, X')$.

CTRENC. Let $0 < \delta \leq n$ be an integer, let IV be a string of $(n - \delta)$ bits, and let K be a key. Let M be a nonempty string of bits such that $|M| \leq n \cdot 2^\delta$. Denote $|M| \pmod n = r$. Parse $M = M_1, \ldots, M_m$ as a sequence of $m = \lceil |M|/n \rceil$ blocks (possibly appending $(n - r)$ 0 bits to M, when $r > 0$, to complete to M to an integer number of blocks). Observe that $1 \leq m \leq 2^\delta$. The (counter mode) CTR encryption of M under the key K with the IV IV is denoted by $\texttt{CTRENC}(K, \texttt{IV}, M)$ and is defined as the ciphertext C computed as follows.

for $j = 1, \ldots, m$
  $Ctr_j = \texttt{IV} \parallel (j-1)_{[\#\delta]}$
  $C_j = M_j \oplus E(K, Ctr_j)$
if $r > 0$ then $C_m^* = Truncate(C_m, r)$
$C = C_m^* \parallel C_{m-1} \parallel \ \ldots \ \parallel C_1$

By definition, $|C| = |M|$. (as a degenerate case, the ciphertext for an empty string is also an empty string)

Remark 2.4. The blocks Ctr_j are called counter blocks. They are well defined for $j = 1, \ldots, m$ due to the constraint $1 \leq m \leq 2^\delta$ and the running counter $(j-1)_{[\#\delta]}$. The choice of δ implies limits on the longest possible length of the message M, although an implementation can choose to allocate δ bits for counter, but independently restrict message lengths to less than $\sim 2^\delta$ blocks. When $\delta = n$ the IV is an empty string and the counter blocks have the form $(j-1)_{[\#\mathbf{n}]}$.

CENC. Let $0 < \delta < n$ be an integer, let IV be a string of $(n - \delta)$ bits, and let K be a key. Let M be a nonempty string of bits such that $|M| \leq n \cdot (2^{\delta} - 1)$. Denote $|M| \pmod{n} = r$. Parse $M = M_1, \ldots, M_m$ as a sequence of $m = \lceil |M|/n \rceil$ blocks (possibly appending $(n - r)$ 0 bits to M, when $r > 0$, to complete to M to an integer number of blocks). Observe that $1 \leq m \leq 2^{\delta} - 1$. The CENC encryption of M under the key K with the IV IV is denoted by $\texttt{CENC}(K, \texttt{IV}, M)$ and is defined as the ciphertext C that is computed as follows.

$T_0 = E(K, \texttt{IV} \parallel (0)_{[\#\delta]})$
for $j = 1, \ldots, m$
  $Ctr_j = \texttt{IV} \parallel (j)_{[\#\delta]}$
  $C_j = M_j \oplus E(K, Ctr_j) \oplus T_0$
if $r > 0$ then $C_m^* = Truncate(C_m, r)$
$C = C_m^* \parallel C_{m-1} \parallel \ \ldots \ \parallel C_1$

By definition, $|C| = |M|$. The CENC encryption mode is defined in [23] and its improved security bounds are shown in [24] (to be a corollary from a theorem proved in [28]). The variation described here is a special case of CENC, where a single additional block is XORed to all blocks in the message whatever its length (instead of having a fixed number w of blocks after which the additional block is replaced).

Remark 2.5. The blocks Ctr_j are called counter blocks. They are well defined for $j = 1, \ldots, m$ due to the constraint $0 < m \leq 2^{\delta} - 1$. The choice of δ implies limits the longest possible length of the message M, although an implementation can choose to allocate δ bits for counter, but independently restrict message lengths to less than $\sim 2^{\delta}$ blocks.

3 Specification of the AEAD scheme Simple128

$\texttt{Simple128} = \texttt{Simple128}_K(N, A, M)$ is an AEAD scheme that operates over a block cipher E for which $\kappa = n$. The scheme encrypts and authenticates a header A, a message M with a nonce N, under the key K. Note that M and/or A may be the empty strings.

Parameters. The parameters that define Simple128 are A_{max}, M_{max}, D_{max}, τ, δ, n, κ (with $\kappa = n$) as follows. The maximal allowed lengths for the header and for the message in any single encryption are denoted by A_{max} and M_{max}, respectively. To be considered legitimate, the input strings A, M must satisfy $0 \leq |A| \leq A_{max}$, $0 \leq |M| \leq M_{max}$. The nonce N has length $|N| = (n - \tau)$, where $2 \leq \tau < n$. It is assumed that $M_{max} \leq n \cdot 2^{\delta} - 1$, and for simplicity the value $\delta = n/2$ is fixed.

For convenience, D_{max} is used hereafter in order to denote the maximal number of bits that can be processed with a given key.

Structure. Simple128 can be viewed as a three step construction: **(a)** nonce based derivation based on CTR mode that produces an encryption key, an authentication key, and two half nonces; **(b)** encryption (in CTRENC mode) of M; **(c)** authentication of $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$ using CBCMAC-IV.

3.1 Nonce based derivation (for $\kappa = n$)

The following derivation function $\texttt{Derive}(K, N)$ is defined.

$\texttt{Derive}(K, N)$
Input: K, N
Parameter: τ
(a.) $K_E = E(K, N \parallel 0_{[\#\tau]})$
(b.) $K_{MAC} = E(K, N \parallel 1_{[\#\tau]})$
(c.) $[N2, N1] = E(K, N \parallel 2_{[\#\tau]})$
Output: $[N2, N1], K_E, K_{MAC}$

Algorithm `Derive`**.**

3.2 The `Simple128` AEAD scheme

The encryption and decryption flows of `Simple128` are illustrated in Algorithm 1 and Algorithm 2, respectively.

$\texttt{Simple128}_K(N, A, M)$
Input: N, A, M
1. $([N2, N1], K_E, K_{MAC}) = \texttt{Derive}(K, N)$
2. if $M = \perp$ then $C = \perp$
else
$\texttt{IV} = N1$
$C = \texttt{CTRENC}(K_E, \texttt{IV}, M)$
end if
3. $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$
$R = N2 \parallel 0^{n/2}$
$Tag = \texttt{CBCMAC-IV}(K_{MAC}, R, X)$
Output: $Tag \parallel C$

Algorithm 1. `Simple128` **- encryption flow.**

$\texttt{Simple128}_K(N, Tag, A, C)$
Input: N, A, C, Tag
1. $([N2, N1], K_E, K_{MAC}) = \texttt{Derive}(K, N)$
2. if $C = \perp$ then $M = \perp$
else
$\texttt{IV} = N1$
$M = \texttt{CTRENC}(K_E, \texttt{IV}, C)$
end if
3. $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$
$R = N2 \parallel 0^{n/2}$
$Tag' = \texttt{CBCMAC-IV}(K_{MAC}, R, X)$
if $Tag' = Tag$ then
$S = M$
else
$S = \perp$
end if
Output: S

Algorithm 2. `Simple128` **- decryption flow.**

Remark 3.1. Note that the decryption flow is almost identical to the encryption flow. This is advantageous in the lightweight setting, as it reduces the size of the code.

3.3 Concrete instantiations

The specific instantiations proposed here are called `Simple128-GIFT` and `Simple128-AES128`. They correspond to the choices `GIFT` (the version with block size of 128 bits and a 128 bits key; [5], [6]) and of `AES128` ([2]) for the block cipher E.

Parameters choice for concrete instantiations. The selected parameters are $\kappa = n = 128$, $\delta = n/2 = 64$, $\tau = 8$, $M_{max} = A_{max} = 2^{53} - 1$, $D_{max} = 2^{53} - 1$. The lengths of all the inputs and outputs are required to be divisible by 8, so that they can be viewed as strings of bytes.

Remark 3.2. Although $\tau = 2$ suffices for the derivation, the value $\tau = 8$ is chosen so that all lengths are divisible by 8 (and considered as bytes).

Random nonces. The nonce length is 120 bits. A uniform random selection of nonces, used across q encryptions has nonce collision probability of (at most) $q^2/2^{121}$. For a limit of $q \leq 2^{46}$ encryptions, this probability is at most 2^{-29} which seems a sufficient margin for practical usage.

4 Specification of the AEAD scheme `Simple64`

$\texttt{Simple64} = \texttt{Simple64}_K(N, A, M)$ is an AEAD scheme that operates with a block cipher E for which $\kappa = 2n$. The scheme encrypts and authenticates a header A, a message M with a nonce N, under the key K. As above, M and/or A may be the empty strings.

Parameters. The parameters that define `Simple64` are A_{max}, M_{max}, D_{max}, τ, δ, n, κ (with $\kappa = n$) as follows. The maximal allowed lengths for the header and for the message are denoted by A_{max} and M_{max}, respectively. To be considered legitimate, the input strings A, M must satisfy $0 \leq |A| \leq A_{max}$, $0 \leq |M| \leq M_{max}$. The nonce N has length $|N| = (n - \tau)$, where $2 \leq \tau < n$. The maximal number of bits that can be processed with a given key is denoted by D_{max}. It is assumed that $M_{max} \leq n \cdot 2^{\delta} - 1$, and for simplicity the value $\delta = n/2$ is fixed.

Structure. `Simple64` can be viewed as a three steps construction: **(a)** nonce based derivation based on XORP/CENC that produces an encryption key, an authentication key, and two half nonces; **(b)** encryption (in `CENC` mode) of M; **(c)** authentication of $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$ using `CBCMAC-IV`.

4.1 Nonce based derivation (for $\kappa = 2n$)

The following derivation function $\texttt{DeriveDouble}(K, N)$, is defined.

$\texttt{DeriveDouble}(K, N)$
Input: K, N
Parameter: τ
$T0 = E(K, N \parallel 0_{[\#\tau]})$
$T1 = E(K, N \parallel 1_{[\#\tau]}) \oplus T0$
$T2 = E(K, N \parallel 2_{[\#\tau]}) \oplus T0$
$T3 = E(K, N \parallel 3_{[\#\tau]}) \oplus T0$
$T4 = E(K, N \parallel 4_{[\#\tau]}) \oplus T0$
$T5 = E(K, N \parallel 5_{[\#\tau]}) \oplus T0$
(a.) $K_E = T2 \parallel T1$
(b.) $K_{MAC} = T4 \parallel T3$
(c.) $[N2, N1] = T5$
Output: $[N2, N1], K_E, K_{MAC}$

Algorithm `DeriveDouble`.

4.2 The Simple64 AEAD scheme

The encryption and decryption for Simple64 are illustrated in Algorithm 3 and Algorithm 4, respectively.

$\texttt{Simple64}_K(N, A, M)$
Input: N, A, M
1. $([N2, N1], K_E, K_{MAC}) = \texttt{DeriveDouble}(K, N)$
2. if $M = \perp$ then $C = \perp$
   else
      $\texttt{IV} = N1$
      $C = \texttt{CENC}(K_E, \texttt{IV}, M)$
   end if
3. $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$
   $R = N2 \parallel 0^{n/2}$
   $Tag = \texttt{CBCMAC-IV}(K_{MAC}, R, X)$
Output: $Tag \parallel C$

Algorithm 3. Simple64 **- encryption flow.**

$\texttt{Simple64}_K(N, Tag, A, C)$
Input: N, A, C, Tag
1. $([N2, N1], K_E, K_{MAC}) = \texttt{DeriveDouble}(K, N)$
2. if $C = \perp$ then $M = \perp$
   else
      $\texttt{IV} = N1$
      $M = \texttt{CENC}(K_E, \texttt{IV}, C)$
   end if
3. $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$
   $R = N2 \parallel 0^{n/2}$
   $Tag' = \texttt{CBCMAC-IV}(K_{MAC}, R, X)$
    if $Tag' = Tag$ then
      $S = M$
   else
      $S = \perp$
   end if
Output: S

Algorithm 4. Simple64 **- decryption flow.**

Remark 4.1. Note that the decryption flow is almost identical to the encryption flow. This is advantageous in the lightweight setting, as it reduces the size of the code.

4.3 Concrete instantiations

The specific instantiations are called Simple64-GIFT and Simple64-PRESENT. They correspond to the choice of GIFT (with block size of 64 bits and a 128 bits key) [5], and [6], and PRESENT (with block size of 64 bits and a 128 bits key) [13] as the block cipher E.

Parameters choice for concrete instantiations. The selected parameters are $n = 64$, $\kappa = 128$, $\delta = n/2 = 32$, $\tau = 8$, $M_{max} = 2^{10}$, $A_{max} = 2^{10}$, $D_{max} = 2^{40}$. This corresponds to messages of up to 128 bytes, and AAD (Additional Authenticated Data) of length up to 128 bytes. The lengths of all the inputs and outputs are required to be divisible by 8, so that they can be viewed as strings of bytes.

Remark 4.2. Although $\tau = 3$ suffices for the derivation, the value $\tau = 8$ is chosen so that all lengths are divisible by 8 (and considered as bytes).

Random nonces. The nonce length is 56 bits. A uniform random selection of nonces, used across q encryptions has nonce collision probability of (at most) $q^2/2^{57}$. For a limit of $q \leq 2^{18}$ encryptions, this probability is at most 2^{-21} which seems a sufficient margin for practical usage. Thus, although Q_{max} equals 2^{30} for unique nonces, it should be set to 2^{18} for random nonces.

5 The concrete proposals of the submission

This section defines the concrete proposals that combine `Simple128` and `Simple64` and specific block ciphers, to define a concrete instantiation. The proposals use the block ciphers `GIFT`, `AES128`, `PRESENT`, and `Speck` as follows.
For $n = 128$, $\kappa = 128$, the proposed instantiations are:

- `Simple128-GIFT`; this is the **primary** submission.
- `Simple128-AES128`
- `Simple128-Speck`

For $n = 64$, $\kappa = 128$, the proposed instantiations are:

- `Simple64-GIFT`
- `Simple64-PRESENT`
- `Simple128-Speck`

These are summarized in Table 1 (page 3).

A brief description of the block ciphers used in the submission are given in the Appendix, and detailed descriptions are given in the respective specifications:
[2] for `AES128`; [5] and [6] for `GIFT`; [13] for `PRESENT`; [7] for Speck-64/128.

5.1 Known answer tests (KAT's)

- The KAT files of `Simple128-GIFT` are available in:

  `crypto_aead/simple128gift/LWC_AEAD_KAT_128_120.txt`

- The KAT files of `Simple128-AES128` are available in:

  `crypto_aead/simple128aes10/LWC_AEAD_KAT_128_120.txt`

- The KAT files of `Simple128-Speck` are available in:

  `crypto_aead/simple128Speck/LWC_AEAD_KAT_128_120.txt`

– The KAT files of `Simple64-GIFT` are available in:

  `crypto_aead/simple64gift/LWC_AEAD_KAT_128_56.txt`

– The KAT files of `Simple64-PRESENT` are available in:

  `crypto_aead/simple64present/LWC_AEAD_KAT_128_56.txt`

– The KAT files of `Simple64-Speck` are available in:

  `crypto_aead/simple64Speck/LWC_AEAD_KAT_128_56.txt`

6 Design rationale, features and advantages

6.1 Design goals and desired properties

The proposed mode of operation (formulating an AEAD scheme) can be instantiated with any cipher (E) with block size of 128 (and 64) bits. Specific instantiation with a suitable choice of a lightweight cipher would be useful in lightweight setting. The desired properties that are targeted in this proposal are as follows.

1. **Security.** The top priority design goal is the security achieved by the scheme. The mode is intended to rely only on the standard (and most basic) assumption on E, that modern block ciphers (in particularly the ones selected for the instantiation) should have as a design goal, namely PRP security. E should be indistinguishable from a random permutation (over $\{0,1\}^{128}$ or $\{0,1\}^{64}$) when the keys are selected uniformly at random from the key space. In particular, additional properties (e.g., related key security) are not required.
2. **Flexibility.** The mode[1] should allow for encrypting a large amount of blocks while preserving a high security margin. To be concrete, considering a 128-bit block, the goal is to be able to encrypt up to 2^{50} bytes (equivalently, 2^{46} blocks) in *any* configuration of number of messages and message length. Specifically (and per the requirements of the NIST call [3, Sec. 3.1]), security should be maintained when 2^{50} single-byte encryptions are carried out, or when encrypting a message of size 2^{50} bytes, and everything in between.
3. **Long lifetime for keys.** The maximal number of messages that can be encrypted with a single key is a significant concern in lightweight scenarios, where devices (communicating with a server) are deployed "in the field" and it could be extremely difficult to rotate keys. It is likely that messages emitted from the device are (very) short, but over time, a large number of such message need to sent.
4. **Random nonces.** The mode that requires a uniquely-chosen nonce should be able to support usage under a randomly-chosen nonce setting. To satisfy the above requirements, a mode that uses a long nonce (as long as possible) is desired.
5. **Simplicity and frugality.** The mode should be simple. Simple to describe and also to implement, e.g., not involving a length block, or not requiring complex padding or multiple conditional branches. The mode should use only a small number of cryptographic primitives, preferably one.
6. **Online (streaming) mode.** The mode should be "online": the lengths of A and of M should not be needed at the onset.

[1] At least the primary instantiation per [3].

Challenges to overcome. CBC-MAC and Counter Mode encryption are obvious choices for *simple* authentication and an encryption primitive. Indeed, the known AEAD scheme CCM (([14], [30], [31]) leverages exactly these primitives to gain its simplicity. Unfortunately, raw CBC-MAC is known to be insecure for messages of arbitrary length, unless it operates over a prefix free set of inputs (or otherwise enhance by e.g., encrypting the tag with an additional key). Due to this, CCM mode encrypts the CBC-MAC tag and also uses a relatively complex padding to achieve the prefix free property. For this reason, CCM is not an online mode and the lengths of A and M are required in advance.

Supporting the use of random nonces poses another design challenge due to conflicting requirements. On the one hand, a nonce respecting scheme cannot safely use short randomized nonces because of the high probability that nonces would repeat. For example, using a 96 bits (or shorter) randomized nonce cannot support the encryption of 2^{50} messages. On the other hand, using Counter Mode with a long nonce/IV limits the maximal allowed length of the message. For example, in order to leave enough bits (in a block of size 128) to account for a message of length 2^{50} bytes (2^{46} blocks), the nonce can have at most 82 bits. The CCM mode (with a nonce length between 56 and 112 bits) is not design to address these conflicting desired. As described below, we overcome this by having a long nonce as input, but deriving a shorter nonce for the actual encryption (and since we derive both a nonce and encryption key, security is maintained as long as there is no simultaneous collision of the derived key and derived nonce).

6.2 A brief intuitive description of `Simple128` and `Simple64`

The mode uses a block cipher E with key size κ and block size n, where $\kappa = n$ or $\kappa = 2n$. The nonce size is $(n - \tau)$ for some $\tau \geq 3$, such that the strings $N \parallel i_{[\#\tau]}$, $i = 0, \ldots 5$ fit in one block. Upon input (N, A, M) and key K, the output C and tag are produced in three (logical) steps, namely Derive, Encrypt, Authenticate as follows.

1. **Derive.** Derive $2\kappa + n$ random bits to define K_E and K_M of length κ each, and two half blocks N_1, N_2 of length $n/2$ each. The derivation invokes E over $N \parallel i_{[\#\tau]}$ for 3 times (if $\kappa = n$) or 6 times (if $\kappa = 2n$).
2. **Encrypt.** Encrypt M in `CTRENC` (if $\kappa = n$) mode or in `CENC` (if $\kappa = 2n$) mode with the key K_E and initial counter N_1. Let C be the result.
3. **Authenticate.** Authenticate $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$ using `CBCMAC-IV` with the key K_M and the IV $N2$. Let Tag be the result.

   Output (C, Tag).

Remark 6.1. The Derive-Encrypt-Authenticate description is a logical description of *Simple*, which is useful for the simplicity of description and for analysis. An implementation can (and should) interleave the encryption and the authentication steps in order to, among other things, avoid reading the input from memory twice.

6.3 Design rationale

- The chosen value for τ is $\tau = 8$, which makes it a byte (technically, $\tau = 3$ suffices).
- The role of per-nonce derivation is to extend the lifetime of the key, to isolate encryption from authentication, and to randomize the IV's for the `CBCMAC-IV` and for the `CTRENC`/`CENC` encryption. Explanations follow.

1. The approach for extending the lifetime of a key through a nonce based derivation method has been studied in [17] and applied to AES-GCM-SIV ([18], [19]). This derivation precedes encryption and authentication, and is an overhead paid toward key lifetime enhancement. Altogether, `Simple128` requires $2\kappa + n$ random bits and these are derived by means of a few invocations of E, whose number depends on the relation between κ and n and on the actual value of n.
   - When $\kappa = n$ and n is sufficiently large (here, 128) 3 calls suffice, as executed by `Derive`. Here, the permutation (E) is viewed as a pseudorandom function. With the targeted number of allowed encryption operations with the same key, the PRP-PRF advantage can be tolerated.
   - When $\kappa = 2n$ and n is small (here, 64) 6 calls are required, as executed in `DeriveDouble`. Here, the PRP-PRF advantage cannot tolerate a large number of messages with the same key, so the permutation is not viewed directly as a pseudorandom function. Rather, a derivation that is based on `CENC` ([23], [23]) is used. This helps the design achieve strong security bounds (especially for $n = 64$).
2. Authentication and encryption are isolated from each other through using separate (independent) keys K_E, K_M.
3. n bits of the overall derived random values are split into two half nonces ($N2$, $N1$). These are used for randomizing the IV for the encryption, and the IV of the `CBCMAC-IV`. This method secures prefix-free inputs to `CBCMAC-IV` (with high probability), and distinct IV's for the `CTRENC`/`CENC` encryption for the case $\kappa = 2n$ where derived keys are not guaranteed to be distinct.

– When $\kappa = n = 128$, encryption of at most 2^{50} messages can be executed with `CTRENC`, because the PRP-PRF security bounds suffice. However, when $\kappa = 2n = 128$, the birthday bound on a 64-bit block cipher (2^{32} without any margins) and the desire to keep sufficient security margins do now allow to encrypt a large amount of data. In this case, `Simple64` uses `CENC` for encryption at a (relatively low) cost of one additional invocation of E per message.

Remark 6.2 (The use of `CBCMAC-IV`). Observe that `Simple128`/`Simple64` use CBC-MAC in a special way: **(a)** the MAC is computed on the *plaintext message* (and not on the ciphertext); and **(b)** an IV is used. It is important to note that the usage is secure in these conditions, due to the key and half nonce derivation.

Here, security is maintained due to the low probability of *simultaneous* collision between: **(a)** an encryption key and an $n/2$-bit nonce (counter mode is secure as long as a key is never used on equal counter blocks); **(b)** a MAC key and $n/2$-bit nonce (making the set of MAC-ed messages prefix free).

The plain CBC-MAC is very simple to implement, and avoiding length encoding allows the AEAD to be an online (streaming) mode.

The choice of the block ciphers to be used with `Simple128`/`Simple64`. `Simple128` is instantiated with `GIFT` (128-bit block size) as the primary variant, and with `AES128`, and `Speck` (128-bit block size). `Simple64`, is instantiated with `GIFT` (64-bit block size), `PRESENT`, and `Speck` (64-bit block size). The choice of AES is obvious, as this is a well established cipher used in multiple standards.

For lightweight-dedicated designs, `PRESENT`, which is already a well studied cipher that is included in ISO standards (ISO/IEC 29192-2:2012 and ISO/IEC 29192-5:2016), is a solid and efficient choice. `GIFT` is a relatively new development (proposed as an

improvement over PRESENT) and is a very competitive design accompanied with a solid security analysis.

Speck is a competitive lightweight design, especially efficient in sofwtare. To the best of our knowledge there is no single-key attack on full-round Speck (64/128) with complexity that is significantly better than brute-force approach.

All of these ciphers are public and their design rationale, properties and security are publicly available.

6.4 Block cipher calls count for Simple128/Simple64

The number of processed blocks for input (N, A, M)**.** The input (for encryption) to the AEAD schemes (Simple128 and Simple64) is (N, A, M). The following computation counts the number of blocks in the padded A and M combination.

- *For the encryption of M:* M (possibly padded with 0 bits to the next boundary of a multiple of n) is parsed as m blocks, where $m = \lceil |M|/n \rceil$.
- *For the authentication of $X = \mathit{pad10}^*(A) \parallel \mathit{pad10}^*(M)$:* X consists of x blocks where $x = a + m'$, and a is the number of blocks in $\mathtt{pad10}^*(A)$ and m' is the number of blocks in $\mathtt{pad10}^*(A)$. The value of a is $a = 1 + \lfloor |A|/n \rfloor$ (e.g., $a = 1$ if $A = \perp$) and the value of m' is $m' = 1 + \lfloor |M|/n \rfloor$.

The performance of Simple128. The performance of Simple128 is measured in terms of the number of invocations of the block cipher E for processing A and M (lower is better). The Derive step requires 3 invocations of E. The CTRENC encryption requires m invocations of E. The CBCMAC-IV authentication requires $x = a + m'$ invocations of E. Thus, the total number of invocations of E is

$$\mathtt{TotalECalls}(\mathtt{Simple128}) = 3 + a + m' + m = 5 + \lfloor |A|/n \rfloor + \lfloor |M|/n \rfloor + \lceil |M|/n \rceil \quad (1)$$

The performance of Simple64. The DeriveDouble step requires 6 invocations of E. The CENC encryption requires $1 + m$ invocations of E. The CBCMAC-IV authentication requires $x = a + m'$ invocations of E. Thus, the total number of invocations of E is

$$\mathtt{TotalECalls}(\mathtt{Simple64}) = 6 + a + m' + m = 8 + \lfloor |A|/n \rfloor + \lfloor |M|/n \rfloor + \lceil |M|/n \rceil \quad (2)$$

6.5 Features

- The design of the mode is very simple, which is a very advantageous property for implementation in the lightweight setting. In addition, the code for decryption is almost identical to encryption, which reduces the size of the code base.
- The mode uses well known constructions CBC/CTR/CENC and nonce-based derivation [17,18,19] (reference the CCS paper and the AES-GCM-SIV) which are proven and have been studied and used.
- Provable security: analysis is for a single key. Some properties hold for multi users too.
- A small number of primitives. The mode uses E only in the encryption direction (no decryption)

- The mode can work with any good cipher. The security relies only on the PRP security of E (specifically, not requiring related key security for the mode, which would be problematic for `GIFT`) For example, see remark on page 10 of [6]): "Remark: GIFT aims at single-key security, so we do not claim any related-key security (even though no attack is known in this model as of today). In case one wants to protect against related-key attacks as well, we advise to double the number of rounds."
- The modes allow for online encryption (streaming messages - no need to know the lengths in advance)
- For `Simple128`, the long nonce (120 bits) allows for using a random nonce with low collision probabilities.
- The mode is simple and easy to analyze.
- `Simple64` allows for crossing the birthday bound w.r.t. the number of encrypted messages, i.e., it offers the option to use a 64-bit block cipher with a respectable number of messages for a single key. The limitation when encryption a very large number of messages is that the messages need to be short, which is a reasonable assumption in the lightweight setting.

6.6 Limitations

While the overhead of the derivation (`Derive`/`DeriveDouble`) step in `Simple128` and `Simple64` is not significant for messages that are not very short, it *is* significant for short messages. However, this seems to be a proper tradeoff between the performance of the schemes and the desired to meet the goals specified in Section 6.3.

Note also that the authentication algorithm used for `Simple128` and `Simple64` is `CBCMAC-IV`, which is an inherently serial process. Here, `CBCMAC-IV` is chosen for its utmost simplicity. This seems to be an adequate tradeoff, because encryption on small devices is likely to be done serially, with an on-the-fly key scheduling for the underlying block cipher.

For the 64-bit block cipher, `Simple64` allows to encrypt a large number of messages (2^{30}) with the same key, provided that they are short (up to 128 bytes here). In the lightweight setting, this seems to be an reasonable tradeoff for allowing the use of a 64-bit block cipher.

It is worth mentioning that `Simple128` and `Simple64` are insecure in nonce-misusing scenarios.

6.7 Possible optimization for specific use cases

It is worth mentioning that for both `Simple128` and `Simple64`, the derivation can be shortened when the mode is used only for authentication and not for confidentiality (i.e., $M = \perp$) since no encryption key (K_E) needs to be derived. This is useful for designs of protocols that use `Simple128`/`Simple64` over messages that are comprised only of AAD.

Note that if A is shorter than one block, then $|X| = |\texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)| = 1 + |\texttt{pad10}^*(M)|$. On the other hand, when A has exactly one block then $|X| = 1 + |\texttt{pad10}^*(M)|$. This suggests that (at least for short messages) designs of protocols that use `Simple128`/`Simple64` should consider encoding A to be shorter than one block. A similar observation is appropriate for the case where M is shorter than one block.

7 Security analysis for `Simple128` and `Simple64`

The precise theorems and proofs will be provided in a separate paper. This document provides an informal outline of the statements and some explanations on how the bounds can be derived.

Hereafter, for simplicity, for every (encryption) query (N, A, M) consider the number of blocks in $\texttt{pad10}^*(M)$ as the "length in blocks of the message", and consider the number of blocks in $X = \texttt{pad10}^*(A) \parallel \texttt{pad10}^*(M)$ as "length in blocks of (the tagged message) X".

The security of the AEAD schemes proposed here is expressed in terms of upper bounds on the distinguishing advantage of an adversary with a given budget of encryption and decryption queries. An adversary $\mathcal{A}$ is said to be a $(q_E, q_D, n, \vec{\ell}_E, \vec{x}_E)$-adversary against the AEAD scheme if it makes at most q_E encryption queries and at most q_D decryption queries in the following setting: **(a)** The respective message lengths (in blocks) of the encryption queries are $\vec{\ell}_E = (\ell_1, \ldots, \ell_{q_E})$; **(b)** The respective lengths (in blocks) of the decryption queries (X) are $\vec{x}_E = (x_1, \ldots, x_{q_E})$.

Denote $x_{max} = \max(x_1, \ldots, x_{q_E})$ and $\ell_{max} = \max(\ell_1, \ldots, \ell_{q_E})$. $\mathcal{A}$ is assumed to be a nonce respecting adversary (i.e., it does not repeat a nonce in encryption queries). Let the block cipher E be modeled as a pseudorandom permutation with PRP advantage at most $\mathsf{Adv}_E^{prp}(q')$ after q' samples.

Let $\Pi = \texttt{Simple128}$. Then,

$$\begin{aligned}
&\mathsf{Adv}_{\Pi}^{\mathsf{nAE}}(\mathcal{A}) \\
&\leq \frac{9 \cdot {q_E}^2}{2^{n+1}} + \sum_{i=1}^{q_E} \frac{{\ell_i}^2}{2^{n+1}} + q_D \cdot \left(\frac{12 \cdot x_{\max}}{2^n} + \frac{64 \cdot {x_{\max}}^4}{2^{2n}} \right) + \mathsf{Adv}_E^{prp}(\texttt{TotalECalls}) \\
&\leq \frac{9 \cdot {q_E}^2}{2^{n+1}} + q_E \cdot \frac{{\ell_{max}}^2}{2^{n+1}} + q_D \cdot \left(\frac{12 \cdot x_{\max}}{2^n} + \frac{64 \cdot {x_{\max}}^4}{2^{2n}} \right) + \mathsf{Adv}_E^{prp}(\texttt{TotalECalls}).
\end{aligned} \tag{3}$$

Explanation. The first term in the bound (the RHS of the above inequality) corresponds to the PRP-PRF advantage from q_E derivations (each one producing 3 keys/values) using `Derive`. The second term corresponds to the PRP-PRF advantage of the `CTRENC` encryption. The third term corresponds to the successful forgery probability in q_D attempts, against `CBCMAC-IV` (see [9]). Note that due to the permutation-based derivation, all MAC keys are distinct, so a MAC key is used for authenticating (at most) one message. The fourth term corresponds to the PRP advantage of the block cipher itself, with `TotalECalls` samples, which is a measurement of its quality as the (underlying) block cipher. The second inequality is brought for simplicity and assumes that each message encrypted is the maximum allowed length.

These expressions can be used in order to set the limits on the input lengths and on the total amount of data to be processed under a single key by substituting $n = 128$, assuming $\mathsf{Adv}_E^{prp}(\texttt{TotalECalls})$ negligible relatively to the other terms. See Table 7 for a demonstration of our bounds for a number of different cases. As can be seen, our mode enables encrypting *well beyond the birthday bound* (overall number of blocks encrypted) while still providing security of 2^{-32}, and is highly flexible providing security for a very large number of medium-size encryptions and few massive encryptions.

n	q_E	ℓ_{max}	q_D	x_{max}	total bytes encrypted	security bound
128	2^{46}	2^{24}	2^{60}	2^{30}	2^{74}	2^{-32}
128	2^{32}	2^{32}	2^{48}	2^{40}	2^{68}	2^{-32}
128	2^{16}	2^{40}	2^{32}	2^{46}	2^{60}	2^{-32}
128	2^{8}	2^{44}	2^{48}	2^{40}	2^{56}	2^{-32}
128	2^{1}	2^{48}	2^{48}	2^{40}	2^{53}	2^{-32}

Table 2. *Security bounds for the* `Simple128` *mode of operation. The impact of* q_E, ℓ_{max} *and* q_D, x_{max} *on the bounds are independent of each other, and so all combinations of these pairs in the table yield the same bounds.*

Remark 7.1. When considering a block cipher (not an ideal random permutation) the term $(T_E \cdot \mu)/2^n$ needs to be added, where μ is the multiplicity of a block in the scheme. The value of μ is small for the single key `Simple128`.

Remark 7.2. If the mode is used with a random nonce (or 120 bits) then the probability for at least one pair of colliding nonce values in q_E attempts is upper bounded by ${q_E}^2/2^{121}$. This term needs to be added to the upper bound on the advantage. It is larger than the first term ${q_E}^2/2^{129}$.

Bounds for `Simple64`.

Let $\Pi =$ `Simple64`. Suppose that $q_E \leq \min\left\{\frac{2^n}{67\cdot 3}, \frac{2^n}{67\cdot \ell_{max}}\right\}$. Then,

$$\begin{aligned}
&\mathsf{Adv}^{\mathsf{nAE}}_{\Pi}(\mathcal{A}) \\
&\leq \frac{25 \cdot q_E}{2^n} + \frac{q_E \cdot (\ell_{max})^2}{2^n} + q_D \cdot \left(\frac{48 \cdot x_{\max}}{2^n} + \frac{256 \cdot {x_{\max}}^4}{2^{2n}}\right) \\
&\quad + \frac{{q_E}^3}{6 \cdot 2^{2n}} + \frac{{q_E}^2}{2^{3n/2+1}} + \mathsf{Adv}^{prp}_{E}(\texttt{TotalECalls})
\end{aligned} \tag{4}$$

Explanation. The first term in the bound (the RHS of the above inequality) corresponds to PRP-PRF advantage from q_E derivations (each one producing 5 keys/values) using `DeriveDouble` (see [23], [24]) The second term corresponds to the PRP-PRF advantage of the `CENC` encryption (see [23], [24]). The third term corresponds to the successful forgery probability in q_D attempts, against `CBCMAC-IV` (see [9]). Here, our analysis takes into account that MAC keys can repeat (at most) twice, so a MAC key can be used for authenticating (at most) two messages. The fourth term corresponds to the probability that the derivation repeats a value 3 or more times. The fifth term corresponds to the probability of the event where two keys are repeated *and* the half nonce ($N1/N2$) is also repeated. The sixth term corresponds to the PRP advantage of the block cipher itself, with `TotalECalls` samples, which is a measurement of its quality as the (underlying) block cipher.

See Table 7 for a demonstration of our bounds for a number of different cases. As can be seen, our mode enables encrypting securely for very good parameters even using a 64-bit block cipher. Note that standard modes of operation break at 2^{30} blocks except with probability 2^{-5}, whereas here it is possible to encrypt this many blocks with security 2^{-32}. Even more blocks can be encrypted if 2^{-24} security is acceptable. This is therefore a unique mode in that it enables the secure use of a 64-bit block cipher (albeit, at the cost of more block cipher invocations for the key derivation).

n	q_E	ℓ_{max}	q_D	x_{max}	total bytes encrypted	security bound
64	2^{26}	2^{2}	2^{22}	2^{2}	2^{31}	2^{-32}
64	2^{23}	2^{4}	2^{19}	2^{6}	2^{30}	2^{-32}
64	2^{15}	2^{8}	2^{15}	2^{9}	2^{26}	2^{-32}
64	2^{27}	2^{6}	2^{27}	2^{6}	2^{36}	2^{-24}
64	2^{20}	2^{9}	2^{20}	2^{13}	2^{32}	2^{-24}
64	2^{13}	2^{13}	2^{14}	2^{18}	2^{29}	2^{-24}

Table 3. *Security bounds for the Simple64 mode of operation. The impact of q_E, ℓ_{max} and q_D, x_{max} on the bounds are independent of each other, and so all combinations of these pairs in the table yield the same bounds.*

7.1 The security of the underlying block ciphers

The security of Simple128 (as a mode of operation) depends on the of the underlying block cipher. Specifically, the (only) property affects the security is the indistinguishability of the block cipher from a random permutation, when the cipher key is selected uniformly at random with the appropriate length (and not e.g., on the related key security of the cipher). This property is a fundamental design goal of block ciphers.

The concrete Simple128 instantiations that are given in this proposal are based on the well known and fairly well studied block ciphers, namely GIFT, AES128, Speck, PRESENT. Some brief comments on te security of these block ciphers is given below.

Security of AES128. The security of AES128is well-established. The best attack on AES128 is the biclique attack in [12] (key recovery with complexity of 2^{126} computations). Some related key attacks on AES with 192-bit and 256-bit keys are known, but not for AES128 (see also [25] that shows that AES128 in related key settings is almost as secure as in single key settings). Recent distinguishers on AES128 [20,21,29,22,8] were shown, but only for reduced rounds AES128. To the best of our knowledge, there is no attack on AES128 that has significantly better complexity than the brute force approach.

Security of GIFT. Analysis of GIFT is provided by the designers in [5,6]. The recent paper [15] shows 9-round (out of 28 rounds) distinguishers for GIFT-64. For GIFT-128 this paper finds only a distinguisher with high data complexity (similar to the original one in [5,6]). To the best of our knowledge, there is no attack on GIFT (with block-size /key-length of 128/128 and 64/128) that has significantly better complexity than the brute force approach.

Security of PRESENT. PRESENT is considered as an acceptable lightweight cipher. It is included in the international standard for lightweight cryptography by ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission). An attack (truncated differential attack) on reduced round PRESENT (26 rounds out of 31) is presented in [11]. An attack on PRESENT (using biclique cryptanalysis) is presented in [1], but its complexity is very close to that of brute force analysis.

Security of Speck. Speck is defined and analyzed in [7]. Additional analysis of reduced-rounds or related-key settings of Speck appears in [10,26,27,4]. The best attack (differential cryptanalysis) on Speck breaks 20 out of the 27 rounds with time complexity $2^{125.56}$ (and data complexity $2^{61.56}$). To the best of our knowledge. there

is no single key attack on `Speck` (with block-size /key-length of 128/128 and 64/128) that has significantly better complexity than the brute force approach.

7.2 Statement

We declare that there are no hidden weaknesses in the `Simple128` and `Simple64` modes of operation. To the best of our knowledge, public third-party analysis do not raise any security threat to the submission's specific proposals, within the limits prescribed in Table 1 (page 3).

8 Acknowledgments

Shay Gueron is a professor at the Department of Mathematics at the University of Haifa and a Senior Principal Engineer at AWS. Yehuda Lindell is a professor at the Department of Computer Science at Bar Ilan University and CEO at Unbound Tech Ltd.

Shay Gueron is a member of the Center for Cyber Law & Policy at the University of Haifa. Shay Gueron and Yehuda Lindell are members of the BIU Center for Research in Applied Cryptography and Cyber Security.

This research was supported by: the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office; the Israel Science Foundation (grant No. 1018/16); a grant from the Ministry of Science and Technology, Israel, and the Department of Science and Technology, Government of India; and the Center for Cyber Law & Policy at the University of Haifa, in conjunction with the Israel National Cyber Directorate in the Prime Ministers Office.

References

1. F. Abed, C. Forler E. List S. Lucks. J. Wenzel: Biclique Cryptanalysis Of PRESENT, LED, And KLEIN, IACR Cryptology ePrint Archive 2012/591 https://eprint.iacr.org/2012/591.pdf (2012).
2. -, FIPS PUB 197: Advanced Encryption Standard (AES) https://web.archive.org/web/20170312045558/http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf (2001).
3. -, Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf
4. Ashur, T., Bodden, D., Dunkelman, O.: Linear cryptanalysis using low-bias linear approximations. IACR Cryptology ePrint Archive 2017/204 https://eprint.iacr.org/2017/204.pdf (2017).
5. S. Banik, S.K Pandey, T. Peyrin, Y. Sasaki, S.M. Sim, Y. Todo. GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption. In: Cryptographic Hardware and Embedded Systems - CHES 2017 Proceedings 321-345 (2017).

6. S. Banik, S.K Pandey, T. Peyrin, S.M. Sim, Y. Todo, Y. Sasaki: GIFT: A Small Present. IACR Cryptology ePrint Archive 2017/622 https://eprint.iacr.org/2017/622.pdf (2017).
7. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, L. Wingers. SIMON and SPECK: Block Ciphers for the Internet of Things. Report #2015/585, 2015. https://eprint.iacr.org/2015/585
8. Bar-On, A., Dunkelman, O., Keller, N., Ronen, E., Shamir, A.: Improved key recovery attacks on reduced-round AES with practical data and memory complexities. In: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II. (2018) 185–212
9. M. Bellare, K. Pietrzak and P. Rogaway. Improved Security Analyses for CBC MACs. In *CRYPTO 2005*, Springer (LNCS 3621), pages 527–545, 2005.
10. Biryukov, A., Roy, A., Velichkov, V.: Differential analysis of block ciphers SIMON and SPECK. In: Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers. (2014) 546–570
11. C. Blondeau, K. Nyberg: Links Between Truncated Differential and Multidimensional Linear Properties of Block Ciphers and Underlying Attack Complexities IACR Cryptology ePrint Archive 2015/184 https://eprint.iacr.org/2015/184.pdf (2015).
12. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique cryptanalysis of the full AES. In: Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings. (2011) 344–371
13. A. Bogdanov, L.R Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B Robshaw, Y. Seurin, C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In Paillier, P., Verbauwhede, I., eds.: CHES 2007. Volume 4727 of LNCS., Springer, Heidelberg 450-466 (2007)
14. M. Dworking. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38c.pdf
15. Eskandari, Z., Kidmose, A.B., Kölbl, S., Tiessen, T.: Finding integral distinguishers with ease. IACR Cryptology ePrint Archive **2018** (2018) 688
16. P. Fouque, G. Martinet, F. Valette and S. Zimmer. On the Security of the CCM Encryption Mode and of a Slight Variant. In *ACNS 2008*, Springer (LNCS 5037), pages 411–428, 2008.
17. S. Gueron, Y. Lindell. Better bounds for block cipher modes of operation via nonce-based key derivation. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017. 1019-1036 (2017).
18. S. Gueron, A. Langley, Y. Lindell. AES-GCM-SIV:specification and analysis. ePrint report 2017/168, https://eprint.iacr.org/2017/168 (2017).
19. S. Gueron, A. Langley, Y. Lindell. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. IETF Internet Draft https://datatracker.ietf.org/doc/draft-irtf-cfrg-gcmsiv/ (Last updated on 2019-01-18)
20. Grassi, L., Rechberger, C., Rønjom, S.: Subspace trail cryptanalysis and its applications to AES. IACR Trans. Symmetric Cryptol. **2016**(2) (2016) 192–225
21. Grassi, L., Rechberger, C., Rønjom, S.: A new structural-differential property of 5-round AES. In: Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic

Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II. (2017) 289–317
22. Grassi, L.: Mixture differential cryptanalysis: a new approach to distinguishers and attacks on round-reduced AES. IACR Trans. Symmetric Cryptol. **2018**(2) (2018) 133–160
23. T. Iwata. New Blockcipher Modes of Operation with Beyond the Birthday Bound Security. In: Proceeding of Fast Software Encryption 2006, Lecture Notes in Computer Science, vol. 4047, pp. 310-327. Springer (2006).
24. T. Iwata, B. Mennink, D. Vizár. CENC is Optimally Secure. Report #2016/1087, 2016. https://eprint.iacr.org/2016/1087
25. Khoo, K., Lee, E., Peyrin, T., Sim, S.M.: Human-readable proof of the related-key security of AES-128. IACR Trans. Symmetric Cryptol. **2017**(2) (2017) 59–83
26. Liu, Y., Fu, K., Wang, W., Sun, L., Wang, M.: Linear cryptanalysis of reduced-round SPECK. Inf. Process. Lett. **116**(3) (2016) 259–266
27. Liu, Y., Witte, G.D., Ranea, A., Ashur, T.: Rotational-xor cryptanalysis of reduced-round SPECK. IACR Trans. Symmetric Cryptol. **2017**(3) (2017) 24–36
28. J. Patarin. On linear systems of equations with distinct variables and small block size. In: Proceedings of Information Security and Cryptology 2005, Lecture Notes in Computer Science, vol. 3935, pp. 299-321. Springer (2006).
29. Rønjom, S., Bardeh, N.G., Helleseth, T.: Yoyo tricks with AES. In: Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I. (2017) 217–243
30. D. Whiting, R. Housley and N. Ferguson. IEEE 802.11-02/001r2: AES Encryption & Authentication Using CTR Mode & CBC-MAC. March 2002.
31. D. Whiting, R. Housley and N. Ferguson. Counter with CBC-MAC (CCM), AES Mode of Operation. Contribution to NIST, May 2002. Available from http://csrc.nist.gov/encryption/modes/proposedmodes/

A Block Cipher Specifications

This appendix provides a brief description of the block ciphers used in the submission. Detailed descriptions are given in the specifications given in:
[13] for `PRESENT`;
[5] and [6] for `GIFT`;
[7] for Speck-64/128;
[2] for `AES128`.

A.1 `PRESENT` (64 bits block and 128 bits key)

PRESENT (with block length of 64 bits and key length of 128 bits) is an Substitution-Permutation network (SP) design with 31 rounds. The key scheduling expands an input key to 32 round keys (Ki, i=1, 2, ..., 32) where K32 is used for "post-whitening" (XOR-ed to the state after the 31 rounds are executed).
Each round consists of three steps: addRoundKey, sBoxLayer, pLayer.

- addRoundKey: add (XOR) the i-th round key (Ki) to the state
- sBoxLayer: apply a single 4-bit S-box 16 times (in parallel) to the 64-bit state.
- pLayer: apply a (fixed) permutation of the bits of the state.

Key scheduling:

- Denote the (current) key bits by K = k127 k126 ... k1 k0.
  Extract the 64 bits k127 k126 ... k64 as the round key.
  Update the key state after the extraction.
- Key state is updated (after the round key extraction):
  Rotate the key register by 61 bit positions to the left; Pass the leftmost eight bits (two leftmost nibbles) through two S-boxes; Add (XOR) the round counter (value i) to k66 k65 k64 k63 k62 (of the current key).

The encryption routine of the cipher is described in the (pseudo-)code below (C syntax).

The constants of PRESENT

```
#define ROUNDS 31
//State-size in Half-Bytes
#define SSZ 16
typedef unsigned char u8;
typedef unsigned int u32;
//SBOX
const u8 PRESENT_SBOX[16] = {0xC, 5, 6, 0xB, 9, 0, 0xA, 0xD, 3, 0xE, 0xF, 8, 4, 7, 1, 2};
//bit permutation
const u8 PRESENT_PERM[64] = {
   0, 16, 32, 48,  1, 17, 33, 49, 2, 18, 34, 50,  3, 19, 35, 51,
   4, 20, 36, 52,  5, 21, 37, 53, 6, 22, 38, 54,  7, 23, 39, 55,
   8, 24, 40, 56,  9, 25, 41, 57, 10, 26, 42, 58, 11, 27, 43, 59,
  12, 28, 44, 60, 13, 29, 45, 61, 14, 30, 46, 62, 15, 31, 47, 63
};
```

Encryption: PRESENT

```
#define ROUNDS 31
//State-size in Half-Bytes
#define SSZ 16
void PRESENT_encrypt(u8 *ct, const u8 *pt,
                     const u8 *masterkey){
  //convert input data from bytes to Half-Bytes
  u8 k_register[32];
  for(u8 i=0; i<16; i++){
    k_register[2*i] = masterkey[i]&0xF;
    k_register[2*i+1] = (masterkey[i]&0xF0)>>4;
  }
  u8 state[SSZ];
  for(u8 i=0; i<SSZ/2; i++){
    state[2*i] = pt[i]&0xF;
    state[2*i+1] = (pt[i]&0xF0)>>4;
  }
  //state = MSB [15][14]...[1][0] LSB
  //key = MSB [31][30]...[1][0] LSB
  u8 round_key[ROUNDS+1][SSZ] = { 0 };
  //generateRoundKeys
  u8 r_counter = 1;
  u8 temp_k_reg[32] = { 0 };
  for(u8 r=0; r<ROUNDS+1; r++){
    //Extract roundkey k127...k64
    for(u8 i=0; i<SSZ; i++){
      round_key[r][i] = k_register[16+i];
    }
    //ROL 61
    for(u8 i=0; i<32; i++){
      temp_k_reg[i] = (k_register[(i+32-15)%32]<<1 |
         k_register[(i+32-16)%32]>>3)&0xF;
    }
    //SBOX-Substitution
    temp_k_reg[31] = PRESENT_SBOX[temp_k_reg[31]];
    temp_k_reg[30] = PRESENT_SBOX[temp_k_reg[30]];
    //~copy temp register
    for(u8 i=0; i<32; i++){
      k_register[i] = temp_k_reg[i];
    }
    //Round Counter Addition
    k_register[15] ^= (r_counter<<2)&0xF;
    k_register[16] ^= r_counter>>2;
    r_counter++;
  }
  u8 bits [64] = { 0 };
  u8 perm_bits [64] = { 0 };
  for(u8 r=0; r<ROUNDS; r++){
    //addRoundKey
    for(u8 i=0; i<SSZ; i++){
      state[i] = state[i] ^ round_key[r][i];
    }
    //sBoxLayer
    for(u8 i=0; i<SSZ; i++){
      state[i] = PRESENT_SBOX[state[i]];
    }
    //pLayer
    //convert state to bits
    for(u8 i=0; i<SSZ; i++){
      for(u8 j=0; j<4; j++){
        bits[4*i+j] = (state[i] >> j) & 0x1;
      }
    }
    //permute the bits
    for(u8 i=0; i<64; i++){
      perm_bits[PRESENT_PERM[i]] = bits[i];
    }
    //convert permutated bits to state
    for(u8 i=0; i<SSZ; i++){
      state[i]=0;
      for(u8 j=0; j<4; j++){
        state[i] ^= perm_bits[4*i+j] << j;
      }
    }
  }
  //last addRoundKey
  for(u8 i=0; i<SSZ; i++){
    state[i] = state[i] ^ round_key[ROUNDS][i];
  }
  //convert back from half-bytes
  for(u8 i=0; i<SSZ/2; i++){
    ct[i] = state[2*i+1]<<4 | state[2*i];
  }
  return;
}
```

A.2 GIFT (128 / 64 bits block and 128 bits key)

GIFT (with block length 64 bits or 128 bits is an SP network with 28 rounds for block size of 64 bits, and 40 round for block size of 128 bit. The key size is 128 bits.
Each round of GIFT consists of three steps: SubCells, PermBits and AddRoundKey.

- SubCells: apply 16 4-bit Sboxes (GS), in parallel, to every nibble of the state.
- PermBits: apply a permutation of the bits of the state.
- AddRoundKey: add (XOR) a (32 bits) round key to bit 0 and bit 1 of eacn nibble of the state; add (XOR) the bit 1 is to the most significant bit of each nibble; add (XOR) a (6-bit) round constant to bit 3 of the first 6 nibbles of the state.

Key scheduling:

- Split the (current) 128-bit key into 8 16-bit words K = k7 k6 k5 k4 k3 k2 k1 k0.
  Extract k1 and k0 as the round key.
  Update the key state after the extraction.
- Key state is updated (after the round key extraction):
  K ← right-rotate-16 (k1, 2) right-rotate-16 (k0, 12) k7 k6 k5 k4 k3 k2
  (here, right-rotate-16 (e, f) is the right rotation of the 16-bit value e by f positions)

The encryption routines of the cipher (with block sizes of 64 and of 128 bits) are described in the (pseudo-)code below (C syntax).

GIFT (128 bits block)

The constants of GIFT (128 bits block)

```
#define ROUNDS 40
typedef unsigned char u8;
typedef unsigned int u32;
//Sbox
const u8 GIFT_S[16] = { 1,10, 4,12, 6,15, 3, 9, 2,13,11, 7, 5, 0, 8,14};
//bit permutation
const u8 GIFT_P[]={
/* Block size = 128 */
  0, 33, 66, 99, 96,  1, 34, 67, 64, 97,  2, 35, 32, 65, 98,  3,
  4, 37, 70,103,100,  5, 38, 71, 68,101,  6, 39, 36, 69,102,  7,
  8, 41, 74,107,104,  9, 42, 75, 72,105, 10, 43, 40, 73,106, 11,
 12, 45, 78,111,108, 13, 46, 79, 76,109, 14, 47, 44, 77,110, 15,
 16, 49, 82,115,112, 17, 50, 83, 80,113, 18, 51, 48, 81,114, 19,
 20, 53, 86,119,116, 21, 54, 87, 84,117, 22, 55, 52, 85,118, 23,
 24, 57, 90,123,120, 25, 58, 91, 88,121, 26, 59, 56, 89,122, 27,
 28, 61, 94,127,124, 29, 62, 95, 92,125, 30, 63, 60, 93,126, 31
};
// round constants
const u8 GIFT_RC[62] = {
    0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F,
    0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B,
    0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E,
    0x1C, 0x38, 0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A,
    0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04, 0x09, 0x13,
    0x26, 0x0c, 0x19, 0x32, 0x25, 0x0a, 0x15, 0x2a, 0x14, 0x28,
    0x10, 0x20
};
```

Encryption: GIFT (128 bits block)

```
#define ROUNDS 40
typedef unsigned char u8;
typedef unsigned int u32;

void blockcipher_encrypt(u8 *ct, const u8 *pt, const u8 *masterkey){

//convert input data from bytes to halfbytes
    u8 key[32];
    for(u8 i=0; i<16; i++){
        key[2*i] = masterkey[i]&0xF;
        key[2*i+1] = (masterkey[i]&0xF0)>>4;
    }

u8 input[32];
    for(u8 i=0; i<16; i++){
        input[2*i] = pt[i]&0xF;
        input[2*i+1] = (pt[i]&0xF0)>>4;
    }



    //input = MSB [15][14]...[1][0] LSB
    //key = MSB [31][30]...[1][0] LSB

u8 bits[128];
u8 perm_bits[128];
u8 key_bits[128];
u8 temp_key[32];

for(u8 r=0; r<ROUNDS; r++){

//SubCells
for(u8 i=0; i<32; i++){
input[i] = GIFT_S[input[i]];
}



//PermBits
//input to bits
for(u8 i=0; i<32; i++){
for(u8 j=0; j<4; j++){
bits[4*i+j] = (input[i] >> j) & 0x1;
}
}

//permute the bits
for(u8 i=0; i<128; i++){
perm_bits[GIFT_P[i]] = bits[i];
}

//perm_bits to input
for(u8 i=0; i<32; i++){
input[i]=0;
for(u8 j=0; j<4; j++){
 input[i] ^= perm_bits[4*i+j] << j;
}
}



//AddRoundKey
//input to bits
for(u8 i=0; i<32; i++){
for(u8 j=0; j<4; j++){
bits[4*i+j] = (input[i] >> j) & 0x1;
}
}

//key to key_bits
for(u8 i=0; i<32; i++){
for(u8 j=0; j<4; j++){
key_bits[4*i+j] = (key[i] >> j) & 0x1;
}
}

//add round key
u8 kbc=0;  //key_bit_counter
for(u8 i=0; i<32; i++){
bits[4*i+1] ^= key_bits[kbc];
bits[4*i+2] ^= key_bits[kbc+64];
kbc++;
}

//add constant
bits[3] ^= GIFT_RC[r] & 0x1;
bits[7] ^= (GIFT_RC[r]>>1) & 0x1;
bits[11] ^= (GIFT_RC[r]>>2) & 0x1;
bits[15] ^= (GIFT_RC[r]>>3) & 0x1;
bits[19] ^= (GIFT_RC[r]>>4) & 0x1;
bits[23] ^= (GIFT_RC[r]>>5) & 0x1;
bits[127] ^= 1;

//bits to input
for(u8 i=0; i<32; i++){
input[i]=0;
for(u8 j=0; j<4; j++){
 input[i] ^= bits[4*i+j] << j;
}
}



//key update
//entire key>>32
for(u8 i=0; i<32; i++){
temp_key[i] = key[(i+8)%32];
}
for(u8 i=0; i<24; i++){
key[i] = temp_key[i];
}

//k0>>12
key[24] = temp_key[27];
key[25] = temp_key[24];
key[26] = temp_key[25];
key[27] = temp_key[26];

//k1>>2
key[28] = ((temp_key[28]&0xc)>>2) ^ ((temp_key[29]&0x3)<<2);
key[29] = ((temp_key[29]&0xc)>>2) ^ ((temp_key[30]&0x3)<<2);
key[30] = ((temp_key[30]&0xc)>>2) ^ ((temp_key[31]&0x3)<<2);
key[31] = ((temp_key[31]&0xc)>>2) ^ ((temp_key[28]&0x3)<<2);
}

//convert back from half-bytes
for(u8 i=0; i<16; i++){
ct[i] = input[2*i+1]<<4 | input[2*i];
}

return;
}
```

GIFT (64 bits block)

The constants of GIFT (64 bits block)

```
#define ROUNDS 28

typedef unsigned char u8;
typedef unsigned int u32;

//Sbox
const u8 GIFT_S[16] = { 1,10, 4,12, 6,15, 3, 9, 2,13,11, 7, 5, 0, 8,14};
const u8 GIFT_S_inv[16] = {13, 0, 8, 6, 2,12, 4,11,14, 7, 1,10, 3, 9,15, 5};
//bit permutation
const u8 GIFT_P[] = {
/* Block size = 64 */
  0, 17, 34, 51, 48,  1, 18, 35, 32, 49,  2, 19, 16, 33, 50,  3,
  4, 21, 38, 55, 52,  5, 22, 39, 36, 53,  6, 23, 20, 37, 54,  7,
  8, 25, 42, 59, 56,  9, 26, 43, 40, 57, 10, 27, 24, 41, 58, 11,
 12, 29, 46, 63, 60, 13, 30, 47, 44, 61, 14, 31, 28, 45, 62, 15
};
// round constants
const u8 GIFT_RC[62] = {
    0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F,
    0x1E, 0x3C, 0x39, 0x33, 0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B,
    0x16, 0x2C, 0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E,
    0x1C, 0x38, 0x31, 0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A,
    0x34, 0x29, 0x12, 0x24, 0x08, 0x11, 0x22, 0x04, 0x09, 0x13,
    0x26, 0x0c, 0x19, 0x32, 0x25, 0x0a, 0x15, 0x2a, 0x14, 0x28,
    0x10, 0x20
};
```

Encryption: GIFT (64 bits block)

```
#define ROUNDS 28

typedef unsigned char u8;
typedef unsigned int u32;

void blockcipher_encrypt(u8 *ct, const u8 *pt,
    const u8 *masterkey){

//convert input data from bytes to halfbytes
    u8 key[32];
    for(u8 i=0; i<16; i++){
        key[2*i] = masterkey[i]&0xF;
        key[2*i+1] = (masterkey[i]&0xF0)>>4;
    }

u8 input[16];
    for(u8 i=0; i<8; i++){
        input[2*i] = pt[i]&0xF;
        input[2*i+1] = (pt[i]&0xF0)>>4;
    }

    //input = MSB [15][14]...[1][0] LSB
    //key = MSB [31][30]...[1][0] LSB


u8 bits[64];
u8 perm_bits[64];
u8 key_bits[128];
u8 temp_key[32];

for(u8 r=0; r<ROUNDS; r++){

//SubCells
for(u8 i=0; i<16; i++){
input[i] = GIFT_S[input[i]];
}


//PermBits
//input to bits
for(u8 i=0; i<16; i++){
for(u8 j=0; j<4; j++){
bits[4*i+j] = (input[i] >> j) & 0x1;
}
}

//permute the bits
for(u8 i=0; i<64; i++){
perm_bits[GIFT_P[i]] = bits[i];
}

//perm_bits to input
for(u8 i=0; i<16; i++){
input[i]=0;
for(u8 j=0; j<4; j++){
 input[i] ^= perm_bits[4*i+j] << j;
}
}


//AddRoundKey
//input to bits
for(u8 i=0; i<16; i++){
for(u8 j=0; j<4; j++){
bits[4*i+j] = (input[i] >> j) & 0x1;
}
}

//key to key_bits
for(u8 i=0; i<32; i++){
for(u8 j=0; j<4; j++){
key_bits[4*i+j] = (key[i] >> j) & 0x1;
}
}

//add round key
u8 kbc=0;  //key_bit_counter
for(u8 i=0; i<16; i++){
bits[4*i] ^= key_bits[kbc];
bits[4*i+1] ^= key_bits[kbc+16];
kbc++;
}

//add constant
bits[3] ^= GIFT_RC[r] & 0x1;
bits[7] ^= (GIFT_RC[r]>>1) & 0x1;
bits[11] ^= (GIFT_RC[r]>>2) & 0x1;
bits[15] ^= (GIFT_RC[r]>>3) & 0x1;
bits[19] ^= (GIFT_RC[r]>>4) & 0x1;
bits[23] ^= (GIFT_RC[r]>>5) & 0x1;
bits[63] ^= 1;

//bits to input
for(u8 i=0; i<16; i++){
input[i]=0;
for(u8 j=0; j<4; j++){
 input[i] ^= bits[4*i+j] << j;
}
}


//key update
//entire key>>32
for(u8 i=0; i<32; i++){
temp_key[i] = key[(i+8)%32];
}
for(u8 i=0; i<24; i++){
key[i] = temp_key[i];
}

//k0>>12
key[24] = temp_key[27];
key[25] = temp_key[24];
key[26] = temp_key[25];
key[27] = temp_key[26];

//k1>>2
key[28] = ((temp_key[28]&0xc)>>2) ^ ((temp_key[29]&0x3)<<2);
key[29] = ((temp_key[29]&0xc)>>2) ^ ((temp_key[30]&0x3)<<2);
key[30] = ((temp_key[30]&0xc)>>2) ^ ((temp_key[31]&0x3)<<2);
key[31] = ((temp_key[31]&0xc)>>2) ^ ((temp_key[28]&0x3)<<2);
}

//convert back from half-bytes
for(u8 i=0; i<8; i++){
ct[i] = input[2*i+1]<<4 | input[2*i];
}
return;
}
```

A.3 `Speck` (128 / 64 bits block and 128 bits key)

`Speck` (with block lengths 64 or 128 bits) is an AddRotateXor (ARX) design. It supports a number of block/key size combinations. Here, the key size is 128 bits. For the 64 bits block size, `Speck` has 27 rounds, and for the 128 bits block size, `Speck` has 32 rounds.

– Each round applies the function $F_k : \mathbb{F}_{2^{32}} \times \mathbb{F}_{2^{32}} \to \mathbb{F}_{2^{32}} \times \mathbb{F}_{2^{32}}$ defined by

$$F_R(X_1, X_0) := (((X_1 \ggg 8) \boxplus X_0) \oplus R, ((X_0 \lll 3) \oplus ((X_1 \ggg 8) \boxplus X_0) \oplus R))$$

where R denotes the 32-bit round key, (X_1, X_0) denotes the 64-bit state of the cipher, $\ggg$ / $\lll$ denote right/left rotation in a 32-bit word by the specified number of positions, and $\boxplus$ denotes addition modulo 2^{32}.

Key scheduling:

– Write K as 4 32-bit words (K_3, K_2, K_1, K_0). Let $L_2 = K_3$, $L_1 = K_2$, $L_0 = K_0$, and $R_0 = K_0$. Then, for $0 \leq i < 27$ (or $0 \leq i < 32$) the round keys are defined by:

$$L_{i+3} = (R_i \boxplus (L_i \ggg 8)) \oplus i,$$
$$R_{i+1} = (R_i \lll 3) \oplus L_{i+3},$$

The round keys are $(R_0, \ldots, R_{26})$ for the block size of 64 bits, and $(R_0, \ldots, R_{31})$ for the block size of 128 bits.

Encryption: Speck (64 bits and 128 bits block)

```
#if BLOCKSIZE == 64
# define ROUNDS 27
//size of a word in bytes
//(pt and ct size = 2 words = 1 block)
# define WSZ 4
//number of words for key
# define M 4
#elif BLOCKSIZE == 128
# define ROUNDS 32
//size of a word in bytes
//(pt and ct size = 2 words = 1 block)
# define WSZ 8
//number of words for key
# define M 2
#endif

#define CARRY(r, a, b)
    ( ((a>>7)&(b>>7)) | ((a>>7)&(!(r>>7))) |
    ((!(r>>7))&(b>>7)) )

typedef unsigned char u8;
typedef unsigned int u32;

void blockcipher_encrypt
(u8 *ct, const u8 *pt, const u8 *K)
{
    u8 L[(ROUNDS+M-2)*WSZ] = { 0 };
    u8 RK[ROUNDS*WSZ] = { 0 };
    u8 carry;
    u8 ct_temp[2*WSZ] = { 0 };

//RK0 = K0
for(u8 j=0; j<WSZ; j++){
RK[j] = K[j];
}

//initial Ls
for(u8 i=0; i<M-1; i++){
for(u8 j=0; j<WSZ; j++){
L[i*WSZ+j] = K[(i+1)*WSZ+j];
}
}

//Key Schedule
    for (u8 i=0; i<ROUNDS-1; i++){
carry = 0;

//L[i+m-1] = (ROR(L[i], 8) + RK[i]) ^ i
for(u8 j=0; j<WSZ; j++){
L[(i+M-1)*WSZ+j] = L[i*WSZ+((j+1)%WSZ)] + RK[i*WSZ+j];

//add carry
L[(i+M-1)*WSZ+j] += carry;

//set next carry
carry = CARRY(L[(i+M-1)*WSZ+j], L[i*WSZ+((j+1)%WSZ)], RK[i*WSZ+j]);

if(j==0){
L[(i+M-1)*WSZ+j] ^= i;
}
}

        //RK[i+1] = ROL(RK[i], 3) ^ L[i+m-1]
for(u8 j=0; j<WSZ; j++){
RK[(i+1)*WSZ+j] = (RK[i*WSZ+j]<<3 |
                        RK[i*WSZ+((j+WSZ-1)%WSZ)]>>5) ^
                        L[(i+M-1)*WSZ+j];
}
    }

//Encryption
for(u8 j=0; j<2*WSZ; j++){
ct[j] = pt[j]; //copy pt to ct
}

    for(u8 i=0; i<ROUNDS; i++){
carry = 0;

//ct[1] = (ROR(ct[1], 8) + ct[0]) ^ RK[i]
for(u8 j=0; j<WSZ; j++){
ct_temp[WSZ+j] = (ct[WSZ+((j+1)%WSZ)] + ct[j]);

//add carry
ct_temp[WSZ+j] += carry;

//set next carry
carry = (ct_temp[WSZ+j] < ct[WSZ+((j+1)%WSZ)]) ||
                        (ct_temp[WSZ+j] < ct[j]);


ct_temp[WSZ+j] ^= RK[i*WSZ+j];
}

//ct[0] = ROL(ct[0], 3) ^ ct[1]
for(u8 j=0; j<WSZ; j++){
ct_temp[j] = ( ct[j]<<3 | ct[(j+WSZ-1)%WSZ]>>5) ^
                        ct_temp[WSZ+j];
}

//copy ct from temp
for(u8 j=0; j<2*WSZ; j++){
ct[j] = ct_temp[j];
}
    }
}
```

A.4 AES128 (128 bits block and 128 bits key)

This document relates only to the **AES128** version with a 128-bit key. **AES128** is a substitution-permutation network with block size of 128 bits, and key size of 128 bits

The 128-bit state of **AES128**is viewed as a sequence of 16 bytes and also as a 4×4 matrix over $\mathbb{F}_{2^8}$ in column-major order, i.e., the state $S_{15}, \ldots, S_1, S_0$ is viewed as:

$$\begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \\ S_1 & S_5 & S_9 & S_{13} \\ S_2 & S_6 & S_{10} & S_{14} \\ S_3 & S_7 & S_{11} & S_{15} \end{bmatrix}$$

The encryption flow consists of a whitening step followd by 9 rounds that execute four transformations **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey** on the state (viewed as a sequence of 16 bytes), followed by the last (10-th) round that executes only the transformations **SubBytes**, **ShiftRows**, and **AddRoundKey** (i.e., skipping **MixColumns**). The encryption flow is the sequence:

1. Initial round key whitening: generate the initial state S^0 using **AddRoundKey** (described below) with the initial whitening key K^0 and the plaintext P.
2. Repeat 9 times:
   (a) **SubBytes**: Let S denote the internal state. Then, map $S_i \mapsto \mathsf{SB}[S_i]$, for $i \in \{0, \ldots, 15\}$, where SB denotes the AES S-box as shown in Table 3 below.
   (b) **ShiftRows**: apply this bytes shuffle of the internal state S:

$$\begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \\ S_1 & S_5 & S_9 & S_{13} \\ S_2 & S_6 & S_{10} & S_{14} \\ S_3 & S_7 & S_{11} & S_{15} \end{bmatrix} \longmapsto \begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \\ S_5 & S_9 & S_{13} & S_1 \\ S_{10} & S_{14} & S_2 & S_6 \\ S_{15} & S_3 & S_7 & S_{11} \end{bmatrix}$$

   (c) **MixColumns**: multiply the state (matrix) by the following matrix:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

   over the field $\mathbb{F}_{2^8}$ represented with the irreducible polynomial $x^8+x^4+x^3+x+1$.
   (d) **AddRoundKey**: apply the mapping (XOR with the i-th round key) $S_j \mapsto S_j \oplus K_j^i$, for all $j \in \{0, \ldots, 15\}$.
3. The last round: execute **SubBytes**, **ShiftRows**, and **AddRoundKey** on the state.

Key scheduling:

1. The cipher key is expanded into 11 round keys, one for each round plus one for the key whitening steo. The round keys are obtained as follows. Write the K as 4 32-bit words $K = (K_3, K_2, K_1, K_0)$. For $S = (S_3, S_2, S_1, S_0) \in (\mathbb{F}_{2^8})^4$, let $\mathsf{SubWord}(S) := (\mathsf{SB}[S_3], \mathsf{SB}[S_2], \mathsf{SB}[S_1], \mathsf{SB}[S_0])$, where SB denotes the **AES128** S-box given in Table 3 below. Then, for $i \in \{0, \ldots, 43\}$:

$$W_i := \begin{cases} K_i & \text{if } i < 4 \\ W_{i-4} \oplus (\mathsf{SubWord}(W_{i-1}) \ggg 8) \oplus \mathrm{R}_{i/4} & \text{if } i \geq 4 \text{ and } i \equiv 0 \pmod 4 \\ W_{i-4} \oplus W_{i-1} & \text{otherwise,} \end{cases}$$

Table 4. AES Round Constants Table.

Round:	1	2	3	4	5	6	7	8	9	10
R	01	02	04	08	10	20	40	80	1b	36

where for $i \in \{1, \ldots, 10\}$, write K^i to denote the i-th round key $W_{4i+3}, W_{4i+2}, W_{4i+1}, W_{4i}$, and K^0 to denote the initial whitening key W_3, W_2, W_1, W_0. The notation R denotes the round constant array given in Table 2

Table 5. The `AES128` S-box (`SubBytes` transformation). The table shows the 256 values of `SubBytes`. In this layout, the column is determined by the least significant nibble, and the row by the most significant nibble. For example, the `SubBytes` value of $0x9c$ is $0xde$.

```
   00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00 63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
10 ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
20 b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
30 04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
40 09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
50 53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
60 d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
70 51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
80 cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
90 60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
a0 e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
b0 e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
c0 ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
d0 70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
e0 e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
f0 8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16
```