SNEIKEN and SNEIKHA

Authenticated Encryption and Cryptographic Hashing

Markku-Juhani O. Saarinen

PQShield Ltd.
Prama House, 267 Banbury Road
Oxford OX2 7HT, United Kingdom
mjos@pqshield.com
Tel. +44 (0)7548 620723

Abstract. The SNEIKEN authenticated encryption algorithm and SNEIKHA hashe are lightweight constructions derived from the SNEIK permutation, a simple yet flexible ARX design with very efficient feedback diffusion. The SNEIK design strategy emphasizes simplicity and ease of building full-featured, yet extremely lightweight cryptographic protocols. We present implementations for Atmel AVR and ARM Cortex M3/M4 targets, where SNEIK performs clearly better than comparable NIST SHA3 and AES instances. We also discuss SNEIGEN, a group of fast diffusers that have limited cryptographic security by themselves, but can be used to efficiently support internal operations of (post-quantum) cryptographic algorithms and protocols.

Keywords: Lightweight Cryptography · Sponge · SNEIKEN · SNEIKHA · SNEIGEN

Contents

Version 20190329023300

1 Introduction

This document describes the SNEIK family of primitives for lightweight cryptography. The primary members of the family are the **SNEIKEN128** AEAD (Authenticated Encryption with Associated Data) algorithm and the **SNEIKHA256** cryptographic hash. SNEIKEN256 and SNEIKHA384 can be paired for higher-security applications.

Name	**Type**	**Security**	**Specification**
SNEIKEN128	AEAD	2^{128} (NIST1)	Section 4.
SNEIKEN192	AEAD	2^{192} (NIST3)	Section 4.
SNEIKEN256	AEAD	2^{256} (NIST5)	Section 4.
SNEIKHA256	Hash	2^{128}	Section 5.
SNEIKHA384	Hash	2^{192}	Section 5.
SNEIGEN	Informational		Appendix A.

The classical security for SNEIKEN AEADs indicates the effort required to breach the confidentiality of a given plaintext with a classical computer, and is equivalent to key size. The effort required to breach integrity of ciphertext (i.e. to create a forgery) is claimed to be equivalent to size of the ciphertext expansion (authentication tag). Any valid attack must ensure that a nonce does not repeat under the same secret key.

For SNEIKHA hash functions we primarily indicate the effort required to produce collisions on a classical computer. (Second) pre-image attacks may require more effort, especially for short, fixed-format messages, as used in some hash-based signature schemes.

We set no explicit limits on the input sizes (hashed message, plaintext, associated data, and the amount of data that can be processed under one key), but we assume it to be under 2^{64} bits for security analysis.

The SNEIGEN Entropy Distribution Function XOFs are included as "informational" as they relate to the parallel NIST development of post-quantum asymmetric cryptography. Even though they have clear use cases in lightweight cryptography, they do not meet the normal security criteria set for stand-alone primitives. They are intended as "building blocks" instead; their security must be evaluated in the context where they are used.

Notation and conventions. SNEIK is an ARX [KN10] type construction built from three very simple operations on 32-bit words:

A:	$x \boxplus y$	Addition modulo word size: $x + y \mod 2^{32}$.
R:	$x \oplus y$	Bitwise exclusive-or operation between x and y.
X:	$x \lll r$	Cyclic left rotation by r bits in 32-bit word.

We also use Boolean operators $\wedge$ and $\vee$ to denote bitwise "and" and "or" operations and double vertical $\|$ to denote concatenation of arrays and strings. Expression enclosed in single verticals $|v|$ refers to its size (length) in bits; we have $|t\|u| = |t| + |u|$.

C-style notation is used for bit and byte arrays (unit size depends on context); vectors are zero-indexed with index in square brackets. We use ranges to indicate sub-arrays; $v[i \cdots j]$ refers to concatenation of all entries from $v[i]$ to $v[j]$, inclusive.

All numerical values are stored and exchanged in little-endian fashion, with the least significant bit, byte, or vector array entry having index 0. Hexadecimal numbers (bytes or words) are prefixed with "`0x`". Bit and byte arrays are read from left to right, with index starting with 0. The 32-bit integer `0x12345678` (decimal 305419896) is therefore stored and transmitted as four bytes `0x78` $\|$ `0x56` $\|$ `0x34` $\|$ `0x12`.

Any integer $n \in (0, 2^m]$ has unique encoding as bit array $\mathsf{B}[m]$ with $n = \sum_i^{m-1} 2^i \mathsf{B}[i]$. Therefore bit i has numerical value 2^i. The first bit (bit 0) of a byte is therefore $2^0 =$ `0x01` and the last bit (bit 7) is $2^7 =$ `0x80`. One can always fetch bit i from a byte array `v[]` in C with an expression such as `(v[i >> 3] >> (i & 7)) & 1`.

```
// cyclic rotate left for 32-bit words
#define ROL32(x, y) (((x) << (y)) | ((x) >> (32 - (y))))

void sneik_f512(void *s, uint8_t dom, uint8_t rounds)
{
    const uint8_t rc[16] = {                        // round constant table
        0xEF, 0xE0, 0xD9, 0xD6, 0xBA, 0xB5, 0x8C, 0x83,
        0x10, 0x1F, 0x26, 0x29, 0x45, 0x4A, 0x73, 0x7C  // (only 8 used now)
    };

    int i, j;                                       // loop counters
    uint32_t t, *v = (uint32_t *) s;                // assume little endian!

    for (i = 0; i < rounds; i++) {                  // loop over rounds
        v[0] ^= (uint32_t) rc[i];                   // xor round constant
        v[1] ^= (uint32_t) dom;                     // xor domain constant
        for (j = 0; j < 16; j++) {
            t = v[j];                               // middle value
            t += v[(j - 1) & 0xF];                  // feedback previous
            t = t ^ ROL32(t, 24) ^ ROL32(t, 25);    // p(x) = x^25 + x^24 + x
            t ^= v[(j - 2) & 0xF];                  // outer feedback
            t += v[(j + 2) & 0xF];

            t = t ^ ROL32(t, 9) ^ ROL32(t, 17);     // q(x) = x^17 + x^9 + x
            t ^= v[(j + 1) & 0xF];                  // reverse feedback
            v[j] = t;                               // store the result
        }
    }
}
```

Listing 1: The SNEIK permutation $\mathsf{f512}_\delta^\rho(\mathsf{S})$ in C. We set $\mathsf{dom} = \delta$ and $\mathsf{rounds} = \rho$.

2 The SNEIK $f512$ Permutation

With π_δ^ρ we denote a family of unkeyed ρ-round permutations on b-bit state S, controlled by a domain identifier δ:

$$\mathsf{S}' = \pi_\delta^\rho(\mathsf{S}). \tag{1}$$

The security of π_δ^ρ should be evaluated in the context where it is used – we are not claiming it to be a hermetic sponge resistant to all structural distinguishers [BDPA11].

The permutation is easily invertible but the inverse permutation is not required by any of the modes proposed in this document.

Parameters. SNEIK is very flexible, but for the purposes of this specification we fix the state size to $b = 512$ bits, which is organized as sixteen 32-bit words ($n = 16$).

We note that the state required my any of the presented AEAD and hashing modes is limited to essentially the 512-bit S – not much more than 64 bytes of RAM is required to perform any operation from start to finish.

Implementation strategies. Listing 1 contains a compact C source code implementation of the SNEIK permutation instantiation $\pi = \mathsf{f512}$ (for $b = 512$) as used in our SNEIKEN, SNEIKHA, and SNEIGEN proposals. This is not an optimized implementation, but presented here for reference (the implementation is complete).

We note that the domain separator δ or `dom` is an 8-bit integer, defined in the context of BLNK2 modes (See Table 2). Round constants are defined for up to 16 rounds, even though this version never uses more than 8. Our current round counts are quite optimistic, so we reserve the right to increase them if deemed necessary due to future cryptanalysis.

There are two basic implementation methods, one organized as a non-linear feedback shift register (suitable for hardware) and a "register window" method suitable for lightweight software implementations.

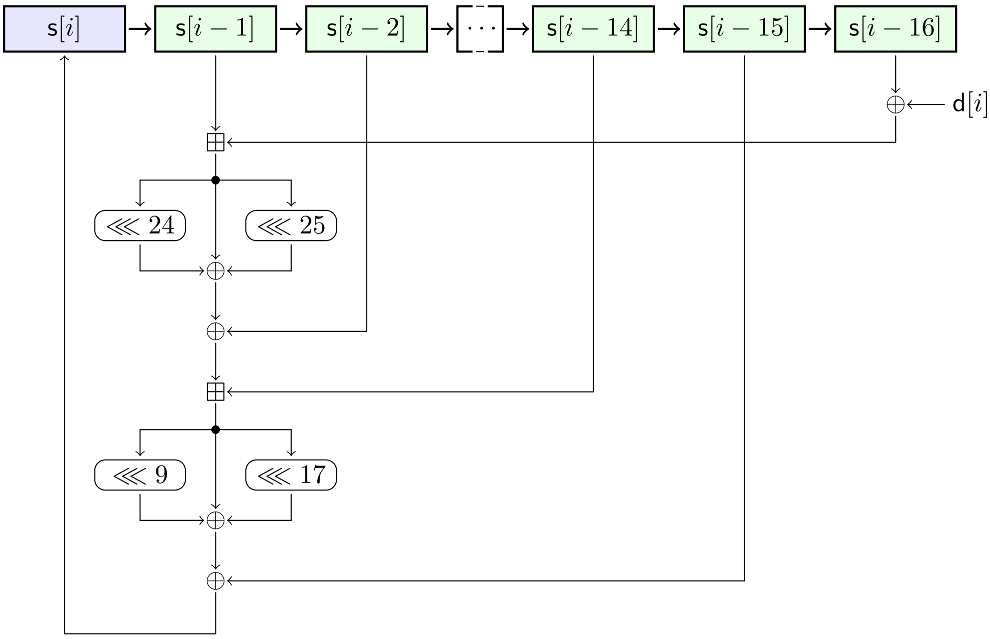

Figure 1: The SNEIK permutation, viewed as a non-linear feedback shift register (NLFSR), Equation 2. We can observe its structural similarity to stream ciphers. In a simple hardware implementation the sixteen registers are moved right for each clock, while a new value computed and loaded into the leftmost 32-bit register.

Non-linear feedback shift register. Let $n \geq 5$ be the size of the initial state $\mathsf{s}[0 \cdots n-1]$ of 32-bit words (with the f512 instantiation we have $n = 16$). Recurrence of Equation 2 defines a nonlinear feedback expander sequence $\mathsf{s}[i]$ for any $i \geq n$. The seven arithmetic steps t_j are numbered just for referencing. Figure 1 illustrates the operation of the NLFSR.

$$
\begin{aligned}
t_1 &= \mathsf{s}[i-n] \oplus \mathsf{d}[i] \\
t_2 &= t_1 \boxplus \mathsf{s}[i-1] \\
t_3 &= t_2 \oplus (t_2 \lll 24) \oplus (t_2 \lll 25) \\
t_4 &= t_3 \oplus \mathsf{s}[i-2] \\
t_5 &= t_4 \boxplus \mathsf{s}[i-n+2] \\
t_6 &= t_5 \oplus (t_5 \lll 9) \oplus (t_5 \lll 17) \\
t_7 &= t_6 \oplus \mathsf{s}[i-n+1] \\
\mathsf{s}[i] &= t_7
\end{aligned}
\tag{2}
$$

To compute r rounds of the SNEIK permutation, we initialize the state $\mathsf{s}[0 \cdots n-1]$ with input, run the expander sequence for nr steps and return $\mathsf{s}[nr \cdots n(r+1)-1]$.

The domain separation constant $\mathsf{d}[i]$ is nonzero only when $i \bmod n \in \{0, 1\}$. We interpret round constants to be just an another kind of "domain separator", separating rounds from each other. We set $\mathsf{d}[nj] = \mathsf{rc}[j]$ from vector in Equation 3 and $\mathsf{d}[nj+1] = \delta$.

The domain identifier value of δ is set by higher level BLNK2 primitive (see Table 2 in Section 3). The first 16 round constants are:

$$
\begin{aligned}
\mathsf{rc}[0..15] = &\mathtt{0xEF}, \mathtt{0xE0}, \mathtt{0xD9}, \mathtt{0xD6}, \mathtt{0xBA}, \mathtt{0xB5}, \mathtt{0x8C}, \mathtt{0x83}, \\
&\mathtt{0x10}, \mathtt{0x1F}, \mathtt{0x26}, \mathtt{0x29}, \mathtt{0x45}, \mathtt{0x4A}, \mathtt{0x73}, \mathtt{0x7C}.
\end{aligned}
\tag{3}
$$

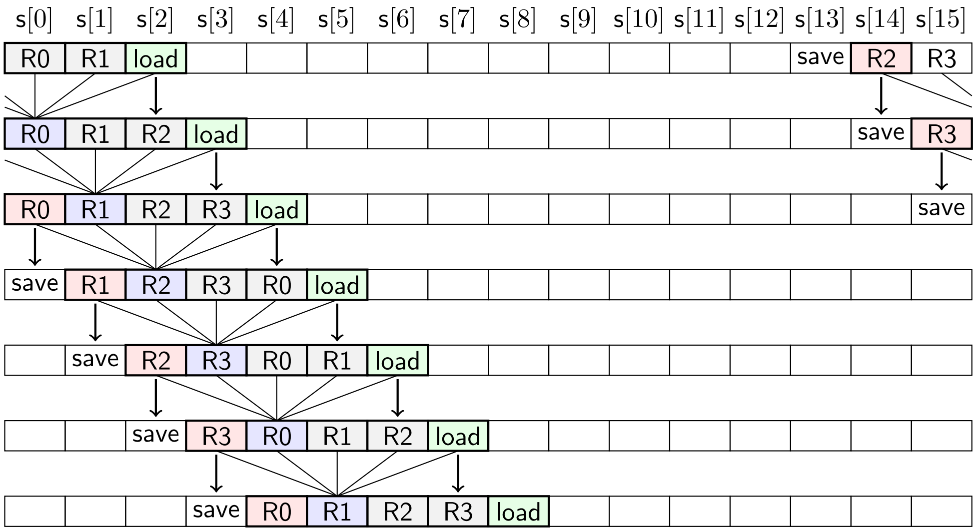

Figure 2: The sliding window implementation technique. Since five consecutive words (with wrap-around) from the state are used to compute a new value for the "middle word" (Equation 4), we can organize the computation in a way that there is only a single load and save per step. A set of four registers can be used in a way that avoids shifting values from one register to another. We can therefore efficiently unroll by 4, 8, or 16 steps.

Sliding register window. Since there are no references beyond $\mathsf{s}[i-n]$ back in the sequence, the recurrence of Equation 2 may be implemented with a static n-word table – as was done in Listing 1. We may use "mod n" addressing and write $\mathsf{s}[i-n \pm j]$ as $\mathsf{s}[i \pm j]$ while i repeatedly scans the values $i = 0, 1, \cdots, n-1$ for each round.

We see that the operation uses a "window" of five inputs to evaluate each new value:

$$\mathsf{s}[i] = f_{\mathrm{win}}(\mathsf{s}[i-2], \mathsf{s}[i-1], \mathsf{s}[i], \mathsf{s}[i+1], \mathsf{s}[i+2]) \tag{4}$$

Four 32-bit state words can be used to store the f inputs as the window moves; the value $\mathsf{s}[i-2]$ is used at step t_4 before a replacement value $\mathsf{s}[i+2]$ is loaded for step t_5. This is illustrated in Figure 2.

The standard implementation method is therefore to unroll computation of at least four iterations of Equation 2. Table 1 gives some implementation metrics for the permutation on popular microcontrollers using this method.

Table 1: SNEIK permutation performance on 32-bit ARM Cortex-M4 (NXP/Freescale MK20DX256 @ 24 MHz) and 8-bit AVR (Atmel ATMEGA2560 @ 16 MHz) architectures. The "RAM" size is the input/output state + stack usage while "ROM" indicates the required Flash memory. Cycles per round was measured with $\rho = 8$.

MCU	**Unroll**	**RAM**	**ROM**	**Cycles/Round**
AVR	16-step	64 + 14	1974	1078.1
AVR	4-step	64 + 19	618	1126.0
Cortex M4	16-step	64 + 16	560	188.0
Cortex M4	4-step	64 + 28	232	211.8

3 BLNK2 Primitive Sponge Operations

Our proposals are built using "BLINKER-style" [Saa14a] primitives. The new version is called BLNK2 (version number is optional). In addition to authenticated encryption and hashing, these primitives can be used to build more complex yet lightweight protocols where two (or more) parties have synchronized, continuously authenticated states.

For these modes a tuple (S, i) defines the entire state: $\mathsf{S} \in \{0,1\}^b$ is the permutation input/output block and $i \in [0, b)$ is a "next bit"read/write index to it, pointing at bit $\mathsf{S}[i]$.

As is usual in permutation-based cryptography, the block size $b = 512$ is split into two halves, "rate" of r bits and "capacity" of c bits; $r + c = b$. We have $\mathsf{S} = \mathsf{S}_r \parallel \mathsf{S}_c$ where $\mathsf{S}_r \in \{0,1\}^r$ and $\mathsf{S}_c \in \{0,1\}^c$. The security of the construction is largely determined by capacity while rate is almost directly proportional to its processing speed.

These primitives may set additional flags on domain parameter δ before passing them to the cryptographic permutation π^ρ_δ. This 8-bit domain identifier is constructed from fields given in Table 2. The primitive operations are:

$\mathsf{S.clr}()$	Clear the state: $\mathsf{S} \leftarrow 0^b$, $i \leftarrow 0$.
$\mathsf{S.fin}(\delta)$	Mark the end of given domain (Algorithm 2).
$\mathsf{S.ratchet}()$	Clear the "rate" part for forward security: $\mathsf{S} \leftarrow 0^r \parallel \mathsf{S}_c$, $i \leftarrow 0$.
$\mathsf{S.put}(\mathsf{D}, \delta)$	Absorb input data D (Algorithm 3).
$\mathsf{D} \leftarrow \mathsf{S.get}(n, \delta)$	Squeeze out n bits into D (Algorithm 4).
$\mathsf{C} \leftarrow \mathsf{S.enc}(\mathsf{P}, \delta)$	Encrypt plaintext P into ciphertext C (Algorithm 5).
$\mathsf{P} \leftarrow \mathsf{S.dec}(\mathsf{C}, \delta,)$	Decrypt ciphertext C into plaintext P (Algorithm 6).

Additionally, we have a utility function $\mathsf{S.inc}(\delta)$ (Algorithm 1) which updates the index i by one and invokes the permutation π^ρ_δ if it reaches the limit set by rate r or block b, depending on the full bit in the domain indicator δ.

Algorithm 1 Increment index: $\mathsf{S.inc}(\delta)$.

Input: Input state (S, i), domain δ

```
i ← i + 1                                     Increment index.
if (δ ∧ full = 0 and i = r) or
   (δ ∧ full = full and i = b) then
    S ← π^ρ_δ(S)                              Apply permutation if rate or block is full.
    i ← 0                                     Reset index.
end if
```

Output: Updated state (S, i).

Algorithm 2 End a data element (padding): $\mathsf{S.fin}(\delta)$.

Input: Input state (S, i), domain δ

```
S[i] ← S[i] ⊕ 1                               Add padding bit, typically byte 0x01.
if δ ∧ full = 0 then
    S[r − 1] ← S[r − 1] ⊕ 1                   Normal capacity; last rate byte gets 0x80.
end if
S ← π^ρ_(δ ∨ last)(S)                         Permutation with domain end marker last.
i ← 0                                         Reset index.
```

Output: Updated state (S, i).

Algorithm 3 Absorb data: $\mathsf{S.put}(\mathsf{D}, \delta)$.

Input: Input state (S, i), data $\mathsf{D} \in \{0,1\}^*$, domain δ.

for $j = 0, 1, .., |\mathsf{D}| - 1$ **do**
  $\mathsf{S}[i] \leftarrow \mathsf{S}[i] \oplus \mathsf{D}[j]$ *Add (xor) input data to the state.*
  $\mathsf{S.inc}(\delta)$ *Increment index i.*
end for

Output: Updated state (S, i).

Algorithm 4 Squeeze data: $\mathsf{D} = \mathsf{S.get}(n, \delta)$.

Input: Input state (S, i), length of output n, domain δ.

for $j = 0, 1, .., n - 1$ **do**
  $\mathsf{D}[j] \leftarrow \mathsf{S}[i]$ *Get a bit from the state.*
  $\mathsf{S.inc}(\delta)$ *Increment index i.*
end for

Output: Output data $\mathsf{D}[0 \cdots n-1]$, updated state (S, i).

Algorithm 5 Encrypt data: $\mathsf{C} = \mathsf{S.enc}(\mathsf{P}, \delta)$.

Input: Input state (S, i), plaintext P, domain δ.

for $j = 0, 1, .., |\mathsf{P}| - 1$ **do**
  $\mathsf{C}[j] \leftarrow \mathsf{S}[i] \oplus \mathsf{P}[j]$ *Xor plaintext with the state.*
  $\mathsf{S}[i] \leftarrow \mathsf{C}[j]$ *Ciphertext goes into the state.*
  $\mathsf{S.inc}(\delta)$ *Increment index i.*
end for

Output: Ciphertext $\mathsf{C}[0 \cdots |\mathsf{P}| - 1]$, updated state (S, i).

Algorithm 6 Decrypt data: $\mathsf{P} = \mathsf{S.dec}(\mathsf{C}, \delta)$.

Input: Input state (S, i), ciphertext C, domain δ.

for $j = 0, 1, .., |\mathsf{P}| - 1$ **do**
  $\mathsf{P}[j] \leftarrow \mathsf{S}[i] \oplus \mathsf{C}[j]$ *Xor ciphertext with the state.*
  $\mathsf{S}[i] \leftarrow \mathsf{C}[j]$ *Ciphertext goes into the state.*
  $\mathsf{S.inc}(\delta)$ *Increment index i.*
end for

Output: Plaintext $\mathsf{P}[0 \cdots |\mathsf{C}| - 1]$, updated state (S, i).

Table 2: Domain indicator δ bits and fields.

Name	Value	Class	Purpose
last	0x01	Flag	Final (padded) block marker.
full	0x02	Flag	Full state indicator.
ad	0x10	Input	Authenticated Data / Hash input.
adf	0x12	Input	Full-state AAD ($\mathsf{adf} = \mathsf{ad} \vee \mathsf{full}$).
key	0x20	Input	Secret key material.
keyf	0x22	Input	Initialization block ($\mathsf{keyf} = \mathsf{key} \vee \mathsf{full}$).
hash	0x40	Output	Hash, MAC, or XOF.
ptct	0x70	In/out	Plaintext/ciphertext duplex block.

4 SNEIKEN: Authenticated Encryption

The SNEIKEN authenticated encryption with associated data (AEAD) algorithm is characterized by the following six variables:

Var	Description	Length
K	Secret key	Fixed k
N	Nonce or IV	Fixed n
A	Associated data	Any a
P	Plaintext	Any p
T	Authentication tag	Fixed t
C	Ciphertext	$p+t$

The algorithm provides integrity and confidentiality protection for P and C but only integrity protection for A. Capacity $c = b - r$ is equivalent to the key size k in encryption and decryption. Associated data is processed at full-state rate ($r = b$). Generally speaking, the confidentiality is at k-bit security level and integrity is at t-bit level.

SNEIKEN128 is the primary member of the family:

Name	Rate	Rounds	Key	Nonce	Tag
SNEIKEN128	$r = 384$	$\rho = 6$	$k = 128$	$n = 128$	$t = 128$
SNEIKEN192	$r = 320$	$\rho = 7$	$k = 192$	$n = 128$	$t = 128$
SNEIKEN256	$r = 256$	$\rho = 8$	$k = 256$	$n = 128$	$t = 128$

Encryption and decryption. We define a 6-byte "variant identifier block" as follows:

$$\mathsf{ID}[0..5] = \texttt{0x61}, \texttt{0x65}, r/8, k/8, n/8, t/8 \tag{5}$$

The first two bytes are ASCII 'a' and 'e', followed by byte lengths for rate, key, nonce, and tag. We denote the encryption process by $C \leftarrow \mathsf{SNEIKEN}(K, N, A, P)$. Algorithm 7 contains the full procedure for SNEIKEN using the BLNK2 primitives from Section 3.

Algorithm 7 Authenticated encryption $C \leftarrow \mathsf{SNEIKEN}(K, N, A, P)$.

Input: Secret key K, (public) nonce N, associated data A, and plaintext P.

$\mathsf{S.clr}()$	*Initialize the state:* $\mathsf{S} = 0^b, i = 0$
$\mathsf{S.put}(\mathsf{ID} \parallel K \parallel N,\ \mathsf{keyf})$	*Identifier, secret key, and nonce.*
$\mathsf{S.fin}(\mathsf{keyf})$	*Pad and permute the key block.*
$\mathsf{S.put}(A,\ \mathsf{adf})$	*Associated authenticated data.*
$\mathsf{S.fin}(\mathsf{adf})$	*Pad and permute, even if* $a = 0$.
$C' \leftarrow \mathsf{S.enc}(P, \mathsf{ptct})$	*Actual ciphertext.*
$\mathsf{S.fin}(\mathsf{ptct})$	*Pad and permute, even if* $p = 0$.
$T \leftarrow \mathsf{S.get}(t, \mathsf{hash})$	*Authentication tag,* t *bits.*
$C \leftarrow C' \parallel T$	*Authenticated ciphertext.*

Output: Ciphertext C.

Algorithm 8 specifies the corresponding decryption and authentication function

$$\{P, \mathsf{FAIL}\} \leftarrow \mathsf{SNEIKEN}^{-1}(K, N, A, C). \tag{6}$$

Decryption must output **only** FAIL upon integrity check failure (no partial plaintext!)

Code Size. Compiling size-optimized `encrypt.c` that implements the NIST AEAD API (for Encryption and Decryption) resulted in 1100 bytes of executable code and data on AVR and 626 bytes on Cortex-M4. This is the only component required for implementation in addition to the permutation (Table 1). Full assembler implementation or co-implementation with SNEIKHA may yield smaller code size.

Algorithm 8 Authenticated decryption $\{P, \mathsf{FAIL}\} \leftarrow \mathsf{SNEIKEN}^{-1}(K, N, A, C)$.

Input: Secret key K, (public) nonce N, associated data A, and ciphertext C ($p + t$ bits).

$\mathsf{S.clr}()$	*Initialize the state:* $\mathsf{S} = 0^b, i = 0$
$\mathsf{S.put}(\mathsf{ID} \parallel K \parallel N,\ \mathsf{keyf})$	*Identifier, secret key, and nonce.*
$\mathsf{S.fin}(\mathsf{keyf})$	*Pad and permute the key block.*
$\mathsf{S.put}(A,\ \mathsf{adf})$	*Associated authenticated data.*
$\mathsf{S.fin}(\mathsf{adf})$	*Pad and permute, even if* $a = 0$.
$P \leftarrow \mathsf{S.dec}(C[0 \cdots p-1], \mathsf{ptct})$	*Decrypt plaintext from first* p *bits of* C.
$\mathsf{S.fin}(\mathsf{ptct})$	*Pad and permute, even if* $p = 0$.
$T = \mathsf{S.get}(t, \mathsf{hash})$	*Authentication tag,* t *bits.*
if $T = C[p \cdots p+t-1]$ **then**	
return P	*Last* t *bits of* C *matches with tag* T.
else	
return FAIL	*Authentication failure.*
end if	

Output: Plaintext P or FAIL.

MAC-and-continue in lightweight setting. Lightweight protocols can avoid per-message rekeying by padding the MAC with $\mathsf{S.fin}(\mathsf{hash})$, and then directly continuing to process the next message (from step 4 in Algorithm 7). The decryption side must of course mirror these operations to keep both parties synchronized.

A protocol that use SNEIKEN in a MAC-and-continue setting can incorporate a ratchet operation $\mathsf{S.ratchet}()$ that explicitly clears the "rate" portion. This enforces forward security. The "capacity" still contains enough secret entropy to maintain security, but the $\mathsf{S.ratchet}()$ operation is clearly irreversible.

Therefore MAC-and-continue is not only a significant speedup but also saves memory and provides forward security. There is no longer any need to retain the original secret key or nonce after initialization. Also all exchanged messages are "continuously authenticated" which simplifies handshake protocol design as separate hashes are not required.

5 SNEIKHA: A Cryptographic Hash

SNEIKHA hash functions produce a h-bit hash H from input data A of arbitrary bit length a. The security against collision search for SNEIKHA algorithms is expected to be $2^{c/2}$ – which is equivalent to $2^{h/2}$ for these fixed-length hashes. Complexity of (second) pre-image search may be higher for format-restricted inputs.

SNEIKHA256 is the primary member of the family:

Name	Hash	Rate	Rounds	Security
SNEIKHA256	$h = 256$	$r = 256$	$\rho = 8$	2^{128}
SNEIKHA384	$h = 384$	$r = 128$	$\rho = 8$	2^{192}

Algorithm 9 specifies SNEIKHA using the BLNK2 primitives of Section 3. We note that if the squeezing step $\mathsf{S.get}()$ is implemented literally (as in Algorithm 4), there may be a final permutation call which is unnecessary if SNEIKHA is not used as a part of some intermediate-hash scheme. This is because these hashes algorithms are internally extensible-output functions (XOFs) cut to certain length h. We may define explicit XOF padding modes in the future if a need arises for "XOF use cases".

Algorithm 9 Cryptographic hash $H \leftarrow \mathsf{SNEIKHA}(A)$.

Input: Data to be hashed A.

1: $\mathsf{S.clr}()$ — *Initialize the state:* $\mathsf{S} = 0^b, i = 0$
2: $\mathsf{S.put}(A,\ \mathsf{ad})$ — *Absorb input data – only to rate.*
3: $\mathsf{S.fin}(\mathsf{ad})$ — *Pad and permute.*
4: $H \leftarrow \mathsf{S.get}(h, \mathsf{hash})$ — *Squeeze hash, h bits.*

Output: Hash H of A.

Code Size. The size-optimized `hash.c` file implementing the NIST hash API compiles into 288 bytes on AVR and 180 bytes on Cortex-M4. This is the only component required for implementation in addition to the permutation (Table 1). Full assembler implementation or co-implementation with SNEIKEN may yield smaller code size. Incremental and keyed hashing constructions are straightforward.

6 Design Rationale

Shared features between AEAD and Hash. The SNEIKEN and SNEIKHA proposals share the underlying SNEIK permutation $\mathsf{f512}_\delta^\rho$ (Section 2), and the BLNK2 padding mechanism (Section 3). Implementations of the two algorithms may have up to 90% common code, as can be seen from the reference implementations provided. We note that SNEIK family is intended as a fully-featured suite that fulfills all symmetric cryptographic needs of a lightweight application; encryption, authentication, PRNG, etc.

The BLNK2 modes are based on author's BLINKER framework for Sponge-based protocols [Saa14a], which has inspired and influenced constructions such as Mike Hamburg's lightweight STROBE protocol framework [Ham17], David Wong's DISCO [Won19], and the Xoodyak suite from the Xoodoo/Keccak team [DHP+18].

We have also used SNEIK to create a "R5SNEIK" variant of the Round5 post-quantum encryption and KEM scheme [BBF+19a, BBF+19b], which is demonstrably more efficient than the original using NIST primitives SHAKE and AES-GCM – See Appendix A.

Design goals. Our main design goal was to create fast permutation-based primitives suitable for prominent 8, 16, and 32-bit embedded microcontrollers – primarily ARM Cortex-M and Atmel AVR families. The 32-bit Cortex-M target directly led to the use of a 32-bit primary datapath, while AVR somewhat limited the use of rotations (which are essentially "free" in Cortex M3/4).

We observe that the size of the permutation n is almost entirely flexible – smaller and larger permutations can be easily constructed. This was one of the original design goals, although it is not used in the current proposals. However, it was clear that the entire permutation state would not fit into the register file of either of the main target platforms, so processing would have to be "localized" to some degree. This led to the "window" design of Equation 4. This is quite different from proposals such as Gimli [BKL+17], whose designers chose to have more localized mixing.

It was obvious that the design should not have any table lookups or conditional branches in order to make it naturally resistant to timing attacks and some other simple side-channel attacks. We toyed for a while with designs inspired by Ascon [DEMS16] and Xoodoo [DHP+18], but the availability of "free" addition on the main target platforms finally made the decision to use ARX an easy one. The efficiency of NSA's SPECK [BSS+13] was a strong inspiration. The overall structure is clearly influenced by a large number of previous proposals, starting with the "Block TEA" algorithm by Wheeler and Needham (which the author cryptanalyzed more than two decades ago [WN98]).

Strong feedback for fast avalanche. Since multiple-issue or superscalar processing is generally not available on lightweight targets, instruction and data path parallelism was not a great concern. Indeed, we decided to take an opposite route and maximize the critical path instead of minimizing it. As a result, we use immediate feedback from one processed word to the next, which helps to diffuse the state extremely rapidly. The design achieves complete avalanche (each input bit affecting each output bit) in only two rounds.

Round structure. The security of SNEIK relies largely on very effective feedback diffusion when the permutation is computed in either direction.

It is easy to see that each step in Equation 2 is invertible. The weight-3 rotation-xor operations at steps t_3 and t_6 can be interpreted as polynomial multiplications in the binary polynomial ring $\mathbb{Z}_2[x]/(x^{32}+1)$:

$$t_3 = p * t_2 \mod x^{32}+1, \text{ with } p = x^{25}+x^{24}+1 \tag{7}$$
$$t_6 = q * t_4 \mod x^{32}+1, \text{ with } q = x^{17}+x^{9}+1. \tag{8}$$

The inverse polynomials have Hamming weight 9:

$$p * (x^{28}+x^{21}+x^{20}+x^{14}+x^{12}+x^{7}+x^{6}+x^{5}+x^{4}) \equiv 1 \pmod{x^{32}+1} \tag{9}$$
$$q * (x^{27}+x^{19}+x^{18}+x^{17}+x^{11}+x^{9}+x^{3}+x^{2}+1) \equiv 1 \pmod{x^{32}+1} \tag{10}$$

The choice of p and q guarantees that input (differentials) of weight less than 6 at t_2 and t_5 will always have output weight of at least 3 at t_3 and t_6. Ignoring the nonlinear operation at step t_5, the composite $p*q$ also has this property, but with guaranteed output weight of 4. The coefficients were chosen in a way to allow for a reasonably efficient implementation on AVR, which only has instructions for single-bit shifts of bytes.

There are some potentially problematic 4-bit to 4-bit rotational differentials such as $\texttt{0x80808080}_{\lll}$, but we could not cancel out the strong feedback propagation in our attack (with this particular p and q selection), which made exploitation difficult.

Round constants. The round constants of Equation 3 are just bytes from a maximum distance separable (MDS) code in decreasing-increasing order. The Hamming distance between each pair is at least 4. Efficient digital circuits can be constructed to generate this code – the last 8 bytes are just logical inverses of the first 8, for example. The modes described in this document only use the first 8 so implementations may choose not to include the last 8.

Doubling the number of rounds. Table 3 defines sixteen round constants as we reserve the option of doubling the number of rounds to $\rho = 12/16$ for extra margin of security.

The current ρ choices are based on quite optimistic estimates from avalanche and simple differential cryptanalysis. We encourage developers to choose the round-doubled versions for applications where throughput is not the main selection criteria (e.g. when hashing is only required only verifying signatures of firmware updates). A notation such as SNEIKEN-k-ρ and SNEIKHA-c-ρ may be used for these variants.

Note that schemes such as ChaCha are used with a wide array of different round selections [Ber08]. The Xoofff proposal uses a six-round Xoodoo [DHAK18], which is known to be vulnerable to algebraic distinguishers (Xoodoo has only been proposed as a secure permutation with 12 rounds).

Sponge modes. As noted, the BLNK2 modes are based on Author's BLINKER framework for lightweight Sponge-based protocols [Saa14a]. The mode implementation is derived from the one used for CBEAM [Saa14b] and WHIRLBOB [SB15] proposals.

Table 3: Performance comparison of some primitives. For Sponge permutations we give cycles/byte estimates for the entire round (full block) and for 128-bit and 256-bit capacity, corresponding to the security of an AEAD or square of security of a hash. The Keccak speeds are derived from a 11,785 cycle assembler implementation of the permutation – the 256-bit security maps to SHAKE128. Note that the RAM usage is not uniformly reported in the literature; clearly one needs RAM for the permutation in Sponge modes and for expanded keys in case of AES. We are reporting just the stack usage in this table.

				– Cycles / Byte –		
Algorithm		**ROM**	**Stack**	Round	128-bit	256-bit
Atmel AVR ATmega						
SNEIK	Fast [This work]	1974	14	16.8	135	270
SNEIK	Small [This work]	618	19	17.6	141	282
AES	Fast [Poe07]	3411	?	15.5	155	
AES	Small [Poe07]	1570	?	17.1	171	
Gimli	Fast [BKL+17]	19218	45	8.88	320	639
Gimli	Small [BKL+17]	778	44	17.2	620	1239
ARM Cortex M3/M4						
SNEIK	Fast [This work]	560	16	2.94	23.5	47.0
SNEIK	Small [This work]	232	28	3.31	26.4	52.0
Gimli	[BKL+17]	3972	44	0.875	31.5	63.0
AES	Unprotected CTR [SS16]	2192/2960	72	≈ 3.5	34.7	49.5
Keccak	XKCP Asm [BDH+19]	7052	?	2.46	64.0	70.1

We use an updated variant with a full-state keying mechanism and also a full-state keyed sponge method for associated data [GPT15, MRV15]. This full-state use case motivated us to move domain separation from capacity to be an “out-of-band” parameter of the cryptographic permutation itself. The capacity of SNEIKEN capacity matches the intended security level, as discussed in [JLM14]. The capacity of SNEIKHA matches with the output size; this means reduced resistance to pre-image attacks, but is widely deemed a satisfactory compromise.

Comparison to other schemes. Table 3 gives a performance comparison against some other candidates. See Table 1 for more detailed SNEIK permutation data. It is clear that SNEIK has advantageous performance and especially code size characteristics over current NIST ciphers; furthermore it is a fit for a wider array of applications.

References

[BBF+19a] Hayo Baan, Sauvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. In *PQCrypto 2019 – The Tenth International Conference on Post-Quantum Cryptography. Chongqing, China, May 8-10, 2019*, volume to appear of *Lecture Notes in Computer Science*. Springer, 2019. URL: https://eprint.iacr.org/2019/090.

[BBF+19b] Hayo Baan, Sauvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: KEM and PKE based on (ring) learning with rounding. Round5 submission to NIST PQC standardization (Second Round), 2019. URL: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[BDH+19] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. XKCP: eXtended Keccak Code Package. Official Keccak family implementation collection., 2019. URL: https://github.com/XKCP/XKCP.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011. URL: https://keccak.team/files/CSF-0.1.pdf.

[Ben14] Josh Benaloh, editor. *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*. Springer, 2014. doi:10.1007/978-3-319-04852-9.

[Ber08] Daniel J. Bernstein. Chacha, a variant of salsa20, 2008. URL: https://cr.yp.to/chacha/chacha-20080128.pdf.

[BFM+18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! faster frodo for the ARM Cortex-M4. *IACR Cryptology ePrint Archive*, 2018:1116, 2018. URL: https://eprint.iacr.org/2018/1116.

[BKL+17] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli : A cross-platform permutation. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017. doi:10.1007/978-3-319-66787-4_15.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013. URL: https://eprint.iacr.org/2013/404.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to the CAESAR Competition, 2016. URL: https://competitions.cr.yp.to/round3/asconv12.pdf.

[DHAK18] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of xoodoo and xoofff. *IACR Trans. Symmetric Cryptol.*, 2018(4):1–38, 2018. URL: https://doi.org/10.13154/tosc.v2018.i4.1-38, doi:10.13154/tosc.v2018.i4.1-38.

[DHP+18] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. IACR Cryptology ePrint Archive 2018/767, November 2018. URL: https://eprint.iacr.org/2018/767.

[Fer06] Niels Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista. Microsoft Technical Report, August 2006. URL: http://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/bitlockercipher200608.pdf.

[GPT15] Peter Gazi, Krzysztof Pietrzak, and Stefano Tessaro. The exact PRF security of truncation: Tight bounds for keyed sponges and truncated CBC. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 368–387. Springer, 2015. doi:10.1007/978-3-662-47989-6_18.

[Ham17] Mike Hamburg. The STROBE protocol framework. *IACR Cryptology ePrint Archive*, 2017:3, 2017. URL: http://eprint.iacr.org/2017/003.

[JLM14] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond 2 c/2 security in sponge-based authenticated encryption modes. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 85–104. Springer, 2014. doi:10.1007/978-3-662-45611-8_5.

[KN10] Dmitry Khovratovich and Ivica Nikolic. Rotational cryptanalysis of ARX. In Seokhie Hong and Tetsu Iwata, editors, *Fast Software Encryption, 17th International Workshop, FSE 2010, Seoul, Korea, February 7-10, 2010, Revised Selected Papers*, volume 6147 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 2010. URL: https://doi.org/10.1007/978-3-642-13858-4_19, doi:10.1007/978-3-642-13858-4_19.

[MKJR16] Kathleen M. Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA cryptography specifications version 2.2. *RFC*, 8017:1–78, 2016. doi:10.17487/RFC8017.

[MRV15] Bart Mennink, Reza Reyhanitabar, and Damian Vizár. Security of full-state keyed sponge and duplex: Applications to authenticated encryption. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, volume 9453 of *Lecture Notes in Computer Science*, pages 465–489. Springer, 2015. doi:10.1007/978-3-662-48800-3_19.

[NIS01] NIST. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication FIPS 197, November 2001. doi:10.6028/NIST.FIPS.197.

[NIS07] NIST. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. NIST Special Publication SP 800-38D, November 2007. doi:10.6028/NIST.SP.800-38D.

[NIS15] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication FIPS 202, August 2015. doi:10.6028/NIST.FIPS.202.

[NIS16] NIST. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash. NIST Special Publication SP 800-185, December 2016. doi:10.6028/NIST.SP.800-185.

[Poe07] B. Poettering. Avraes: The aes block cipher on avr controllers, 2007. URL: http://point-at-infinity.org/avraes/.

[Saa14a] Markku-Juhani O. Saarinen. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. In Benaloh [Ben14], pages 270–285. doi:10.1007/978-3-319-04852-9_14.

[Saa14b] Markku-Juhani O. Saarinen. CBEAM: efficient authenticated encryption from feebly one-way ϕ functions. In Benaloh [Ben14], pages 251–269. doi:10.1007/978-3-319-04852-9_13.

[SB15] Markku-Juhani O. Saarinen and Billy Bob Brumley. Whirlbob, the whirlpool based variant of STRIBOB. In Sonja Buchegger and Mads Dam, editors, *Secure IT Systems, 20th Nordic Conference, NordSec 2015, Stockholm, Sweden, October 19-21, 2015, Proceedings*, volume 9417 of *Lecture Notes in Computer Science*, pages 106–122. Springer, 2015. doi:10.1007/978-3-319-26502-5_8.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on cortex-M3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016. URL: https://doi.org/10.1007/978-3-319-69453-5_10, doi:10.1007/978-3-319-69453-5_10.

[WN98] David J. Wheeler and Roger M. Needham. Correction to xtea. Informal Manuscript or Report, 1998. URL: https://www.mjos.fi/doc/misc/xxtea.pdf.

[Won19] David Wong. Disco: Modern session encryption. *IACR Cryptology ePrint Archive*, 2019:180, 2019. URL: https://eprint.iacr.org/2019/180.

A SNEIGEN Entropy Distribution Functions

SNEIGEN is a seed expander with limited cryptographic strength − it is not an authenticated encryption or hash function algorithm per se, and therefore not part of the main proposal. It is intended for cryptographic applications that need "random-like stuffing" or "lightweight mixing" with well-understood entropy flow properties, or a deterministic PRNG source with good statistical properties.

Name	Entropy	Rate	Rounds	Security
SNEIGEN128	$c = 128$	$r = 384$	$\rho = 3$	limited
SNEIGEN192	$c = 192$	$r = 320$	$\rho = 4$	limited
SNEIGEN256	$c = 256$	$r = 256$	$\rho = 5$	limited

Algorithmically SNEIGEN works exactly like SNEIKHA (Algorithm 9); it is basically a XOF (Extensible Output Function). One can squeeze any amount of bits out with successive calls to $\mathsf{S.get}(*, \mathsf{hash})$. However we are not ruling out more complex interactions with the state (such as ratcheting or reseeding, see below for PRNG use) when SNEIGEN is used to build higher-level primitives.

Properties. The main security requirement for a lightweight mixing function is captured in the term "Entropy Distribution Function" (EDF); once seeded with $n \leq c$ truly random bits (n bits of entropy), any n-bit output should also have close to n bits of randomness (entropy) when observed without joint information.

SNEIGEN is **not** claimed to be collision resistant, but full collisions are unlikely for outputs that are much larger than c. We note that given more than c bits of output, an attacker may be able to distinguish it from random, and may also derive the state or even the input seed from it. However, specific cryptanalytic effort is required for this.

Since the SNEIK permutation has a $2 \times 32 = 64$ - bit feedback "accumulator" (words $\mathsf{s}[i-1]$ and $\mathsf{s}[i-2]$ in Equation 2) diffusion to the right, and the first round does not achieve much diffusion to the left, the number of rounds is chosen as $\rho = \lfloor c/64 \rfloor + 1$.

Algebraic Interaction. The output of an EDF should not "interact algebraically" with arithmetic operations of the higher-level cryptographic primitive that uses it. This means that, as an example, a completely linear EDF probably should not be used to distribute entropy between other linear components; there is a possibility that some of the entropy will algebraically cancel out or that the shared algebraic structure can somehow be used to attack the higher-level primitive.

We argue that the ARX structure of SNEIK should not interact with common rings, lattices, and other similar number theoretical structures. However this must be analyzed on case-by-case basis.

R5SNEIK. The SNEIGEN EDF is used by the R5SNEIK variants of the Round5 post-quantum public-key cryptosystem [BBF+19a, BBF+19b]. This is not an official part of the Round5 submission, but a result of separate ongoing research[1]. SNEIGEN replaces NIST standard SHAKE [NIS15] and cSHAKE [NIS16] XOFs in this Round5 variant for public vector/matrix $\mathbf{A}$ computation and also for derivation of secret ternary polynomials.

When instantiated as a public key encryption algorithm (rather than simply as a KEM), the more secure SNEIKEN algorithms (Section 4) are used to replace AES-GCM [NIS01, NIS07] as a Data Encapsulation Mechanism (DEM) to transport bulk data.

The overall speed-up is significant, but the main advantage is that full-featured BLNK2-based protocols can be built from these simple lightweight primitives; the Round5 KEM and PKE are used to provide an (authenticated) key exchange in this framework.

Other Applications. The authors of [BFM+18] argue that "good statistical properties" are sufficient for the public matrix $\mathbf{A}$ in a lightweight implementation of the FrodoKEM, an another NIST PQC candidate. They use `xoshiro128**`, a very simple, fully XOR-linear PRNG (actually a seed expander) with a 128-bit state.

An example of a lightweight mixing function used to support symmetric cryptography is the "Elephant" diffuser used with AES-CBC in the original version of Microsoft's Bitlocker disk encryption system [Fer06].

Another traditional example is the padding in RSA PKCS #1 [MKJR16]; the padding of the RSA message really does not require absolute cryptographic security − lack of algebraic interaction with the RSA operation is sufficient.

Random Number Generation. If the SNEIK permutation is used to build a general-purpose random number generator, this may also be called "SNEIGEN". New randomness can be added after $\mathsf{S.fin}(\mathsf{hash})$ with $\mathsf{S.put}(\mathsf{R}, \mathsf{ad})$, $\mathsf{S.fin}(\mathsf{ad})$. Further random bits may then be extracted with $\mathsf{S.get}(*, \mathsf{hash})$. If cryptographic security is required from the generator, we suggest increasing the number of rounds to $\rho = 8$ or even $\rho = 16$ and having capacity of at least $c \geq 192$. Furthermore, $\mathsf{S.ratchet}()$ can be used for PRNG forward security.

[1] An R5SNEIK implementation is included with https://github.com/r5embed/r5embed