SNEIKEN and SNEIKHA

Authenticated Encryption and Cryptographic Hashing (Preliminary version of Monday 25th February, 2019)

Markku-Juhani O. Saarinen

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

Abstract. We describe the lightweight SNEIK permutation and two derived sponge modes: the SNEIKEN Authenticated Encryption with Associated Data (AEAD) algorithm and the SNEIKHA Cryptographic Hash. The permutation is a simple ARX design with very efficient feedback mixing, optimized for low-end microcontrollers. The overall design emphasizes simplicity and ease of integration with lightweight cryptographic protocols and post-quantum schemes.

Keywords: Lightweight Cryptography · Sponge Modes · SNEIKEN · SNEIKHA

Contents

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}	Section 4.
SNEIKEN256	AEAD	2^{256}	Section 4.
SNEIQEN128	AEAD		Section 4.
SNEIKHA256	HASH	2^{128}	Section 5.
SNEIKHA384	HASH	2^{192}	Section 5.
SNEIGEN128	XOF		Section 5

The classical security for (SNEIKEN) AEADs indicates the effort required to breach the confidentiality of given plaintext, 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 effort required to produce collisions on a classical computer. (Second) pre-image attacks may require more effort, especially for fixed-format or short messages, as used in some hash-based signatures.

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

The SNEIQEN128 AEAD and SNEIGEN128 XOF are included as "informational". Even though they have clear use cases in lightweight cryptography, they may not meet the most stringent security criteria for all applications. They but are intended as "building blocks" instead; their security must be evaluated in the context where they are used.

Shared features between AEAD and Hash. The SNEIKEN (AEAD) and SNEIKHA (hash algorithm) proposals share the underlying SNEIK permutation $\mathsf{f512}^{\rho}_{\delta}$ (Section 2), and the BLNK2 padding mechanism (Section 3). Implementations of the two algorithms may share up to 90% common code, as can be seen from the reference implemtations provided.

We note that SNEIK is intended as a full-featured suite that fulfills all symmetric cryptographic needs of a lightweight application. The BLNK2 modes are based on Author's BLINKER framework for lightweight Sponge-based protocols [Saa14a], which has inspired derivative works such Mike Hamburg's lightweight STROBE protocol [Ham17].

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 vertical $\|$ to denote concatenation of arrays and strings.

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

```c
// 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 $\mathsf{f512}^{\rho}_{\delta}(\mathsf{S})$ in C. We set $\mathsf{dom} = \delta$ and $\mathsf{rounds} = \rho$.

All numerical values are stored and exchanged in little-endian fashion, with the least significant byte, bit, 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 4 byte bit vector 0x78 ‖ 0x56 ‖ 0x34 ‖ 0x12.

2 The SNEIK Permutation

With π^{ρ}_{δ} we denote a family of ρ-round permutations on b-bit state S, controlled by a domain identifier δ:

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

Listing 1 contains a compact C source code implementation of the SNEIK permutation instantiation $\pi = \mathsf{f512}$ (with $b = 512$) used in our SNEIKEN and SNEIKHA proposals.

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

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

Table 1: SNEIK permutation performance on 32-bit ARM Cortex M4 (NXP/Freescale MK20DX256) and 8-bit AVR (Atmel ATMEGA2560) architectures. The "RAM" size is the input/output state + stack usage. 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

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 $\mathsf{d}[n * j] = \mathsf{rc}[j]$ from vector in Equation 3 and $\mathsf{d}[n * j + 1] = \delta$. The domain identifier value of δ is set by higher level primitive.

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

Implementation: "Sliding window". Since there are no references beyond $\mathsf{s}[i - n]$ back in the sequence, the recurrence may be implemented with a static n-word table as is 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:

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

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.

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 implementation method.

3 BLNK2 Primitive Sponge Operations

Our proposals are built from lower-level "BLINKER-style" [Saa14a] primitives. In addition to authenticated encryption and hashing, these primitives can be used to build more complex 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 state and $i \in [0, b)$ is a "next bit" $\mathsf{S}[i]$ read/write index to it. The 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.

$\mathsf{S.clr}()$	Clear the state: $\mathsf{v} \leftarrow 0^b$, $i \leftarrow 0$.
$\mathsf{S.fin}(\delta)$	Mark the end of given domain (Algorithm 2).
$\mathsf{S.put}(\mathsf{D}, \delta)$	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 the rate or block is full, 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

1: $i \leftarrow i + 1$ *Increment index.*
2: **if** $(\delta \wedge \mathsf{full} = 0$ and $i = r)$ or
$(\delta \wedge \mathsf{full} = \mathsf{full}$ and $i = b)$ **then**
3: $\mathsf{S} \leftarrow \pi^{\rho}_{\delta}(\mathsf{S})$ *Apply permutation if rate or block is full.*
4: $i \leftarrow 0$ *Reset index.*
5: **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

1: $\mathsf{S}[i] \leftarrow \mathsf{S}[i] \oplus 1$ *Add padding bit, typically byte* `0x01`.
2: **if** $\delta \wedge \mathsf{full} = 0$ **then**
3: $\mathsf{S}[r-1] \leftarrow \mathsf{S}[r-1] \oplus 1$ *Normal capacity; last rate byte gets* `0x80`.
4: **end if**
5: $\mathsf{S} \leftarrow \pi^{\rho}_{(\delta \ \vee \ \mathsf{last})}(\mathsf{S})$ *Permutation with domain end marker* last.
6: $i \leftarrow 0$ *Reset index.*

Output: Updated state S.

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

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

1: **for** $j = 0, 1, .., \mathsf{length}(\mathsf{D}) - 1$ **do**
2: $\mathsf{S}[i] \leftarrow \mathsf{S}[i] \oplus \mathsf{D}[j]$ *Add (xor) input data to the state.*
3: $\mathsf{S.inc}(\delta)$ *Increment index i.*
4: **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.

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

Output: Output data D, 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.

1: $\mathsf{C} = \{\}$ *Empty ciphertext.*
2: **for** $j = 0, 1, .., \mathsf{length}(n)(\mathsf{P}) - 1$ **do**
3: $\mathsf{C}[j] \leftarrow \mathsf{S}[i] \oplus \mathsf{P}[j]$ *Xor plaintext with the state.*
4: $\mathsf{S}[i] \leftarrow C[j]$ *Ciphertext goes into the state.*
5: $\mathsf{S.inc}(\delta)$ *Increment index i.*
6: **end for**

Output: Ciphertext C, 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.

1: $\mathsf{P} = \{\}$ *Empty plaintext.*
2: **for** $j = 0, 1, .., \mathsf{length}(n)(\mathsf{P}) - 1$ **do**
3: $\quad \mathsf{P}[j] \leftarrow \mathsf{S}[i] \oplus \mathsf{C}[j]$ *Xor ciphertext with the state.*
4: $\quad \mathsf{S}[i] \leftarrow C[j]$ *Ciphertext goes into the state.*
5: $\quad \mathsf{S.inc}(\delta)$ *Increment index i.*
6: **end for**

Output: Plaintext P, updated state (S, i).

Table 2: Domain indicator δ bits and fields. Not all are used in current proposals.

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 The SNEIKEN Authenticated Encryption Algorithm

The SNEIKEN family of authenticated encryption with associated data (AEAD) algorithms are characterized by six bit string 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	$c = p + t$

The algorithms aim to provide integrity and confidentiality protection for P and C but only integrity protection for A. Generally speaking the confidentiality should be at k-bit security level and integrity at t-bit level (this may not hold for SNEIQEN128 in all attack models, however.) SNEIKEN128 is the primary member of the family:

Name	Rate	Rounds	Key	Nonce	Tag
SNEIKEN128	$r = 384$	$\rho = 6$	$k = 128$	$n = 128$	$t = 128$
SNEIKEN256	$r = 256$	$\rho = 8$	$k = 256$	$n = 128$	$t = 128$
SNEIQEN128	$r = 384$	$\rho = 4$	$k = 128$	$n = 96$	$t = 128$

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

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

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 BLNK primitives defined in 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.

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

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.

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

Output: Plaintext P or FAIL.

Code Size. Compiling `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.

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 do the same. This is not only a significant speedup but also saves memory and provides "forward security" since there is no longer any need to retain the original secret key or nonce.

SNEIQEN Use Cases. The 4-round SNEIQEN may not be suitable as universally as the main SNEIKEN algorithms. It is intended for applications where attacker has only a limited ability to perform chosen plaintext- or ciphertext queries – which is often the case with low-bandwidth lightweight devices. The suitability of SNEIQEN must be evaluated individually for each application.

5 The SNEIKHA Cryptographic Hash

The SNEIKHA family of 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 $2^{\frac{b-r}{2}}$ – 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
SNEIKHA256	$h = 256$	$r = 256$	$\rho = 8$
SNEIKHA384	$h = 384$	$r = 128$	$\rho = 8$
SNEIGEN128	$h =$ any	$r = 384$	$\rho = 4$

Algorithm 9 specifies SNEIKHA using the BLNK primitives of Section 3. We note that if the squeezing step S.get() is implemented literally (as in Algorithm 4), there will be a final permutation call which is unnecessary if SNEIKHA is not used as part of some intermediate-hash scheme. This is because internally the SNEIKHA algorithms are really extensible-output functions (XOFs). We may define explicit XOF padding modes in the future if a need arises to distinguish XOF use cases from fixed-length hashes.

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

Input: Data to be hashed A.

1: S.clr() *Initialize the state:* $\mathsf{S} = 0^b, i = 0$
2: S.put(A, adf) *Absorb input data.*
3: $A \leftarrow$ S.get(h, hash) *Squeeze hash, h bits.*

Output: Hash H of A.

Code Size. The `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 the 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.

SNEIGEN Use Cases. We are also including SNEIGEN, which is really not a hash function but a seed expander with limited cryptographic strength. It is intended for cryptographic applications that need "random-like stuffing". One such example is the padding in PKCS #1 [MKJR16]. An another example is the expansion of a short seed into public value **A** in many lattice-based public key algorithms, including Round5 [BBF+19]. The authors of [BFM+18] argue that "good statistical properties" are sufficient for the public matrix **A** in a lightweight implementation of the Frodo PQC encryption algorithm.

If the SNEIK permutation is used to build a general-purpose random number generator, this is also called "SNEIGEN". New randomness can be added at any point with S.put(). If cryptographic security is required from the generator, we suggest increasing the number of rounds to $\rho = 8$ and limiting rate to $r \leq b/2$.

6 Design Rationale

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 limited the use of rotations (which are essentially "free" in Cortex M3/4). It was clear that the entire permutation state would not fit into the register file of either of these targets, so processing would have to be "localized" to some degree. This lead to the "window" design of Equation 4.

Embracing the sequential. Since multiple-issue or superscalar processing is generally not available on lightweight targets, instruction and data path parallelism is not a great concern. Indeed, we decided to go an opposite route and maximize the critical path instead of minimizing it. As a result, we can 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.

The permutation design 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].)

Round structure. 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 reasonably efficient implementation on AVR, which only has instructions for single bit shifts of bytes.

There are some problematic 4×4-bit rotational differentials such as $\texttt{0x80808080}_{\lll}$, but in our analysis these didn't "cancel out" feedback so they could not be exploited. Generally the security relies on very effective feedback diffusion when the permutation is computed in either direction.

Sponge modes. The BLNK2 modes are based on Author's BLINKER framework for lightweight Sponge-based protocols [Saa14a], which has inspired derivative works such Mike Hamburg's STROBE [Ham17]. The mode implementation is derived from the one used for CBEAM [Saa14b] and WHIRLBOB [SB15] proposals.

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. Otherwise the capacity matches the intended security level, as discussed in [JLM14].

References

[BBF+19] 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.

[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.

[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.

[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.

[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.

[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.