CAESAR submission: Keyak v1

Designed and submitted by:

Guido Bertoni[1]
Joan Daemen[1]
Michaël Peeters[2]
Gilles Van Assche[1]
Ronny Van Keer[1]

http://keyak.noekeon.org/
keyak (at) noekeon (dot) org

Version **1.1**
March 25, 2014

[1]STMicroelectronics
[2]NXP Semiconductors

Contents

1 Definitions **3**
- 1.1 Notation . . . 3
- 1.2 Of bits and bytes . . . 3
- 1.3 Padding rules . . . 4
- 1.4 Key pack . . . 4
- 1.5 Converting a string into blocks . . . 4

2 The Keccak-p permutations **4**

3 The duplex construction **6**
- 3.1 Specification . . . 7
- 3.2 Rationale . . . 8

4 The serial authenticated encryption mode DuplexWrap **8**
- 4.1 Specification . . . 10
- 4.2 Comparison with SpongeWrap . . . 11
- 4.3 Rationale . . . 13

5 Keyak **14**
- 5.1 Serial instances . . . 14
- 5.2 The parallelizable authenticated encryption mode KeyakLines . . . 15
- 5.3 Parallelizable instances . . . 17
- 5.4 Security goals . . . 17
- 5.5 Rationale . . . 18

6 Using Keyak in the context of CAESAR **20**
- 6.1 Specification and security goals . . . 20
- 6.2 Security analysis and design rationale . . . 20
- 6.3 Features . . . 21
- 6.4 Intellectual property . . . 21
- 6.5 Consent . . . 21

A Change log **22**
- A.1 From 1.0 to 1.1 . . . 22

KEYAK is a set of four authenticated encryption functions with support for message associated data. They are aimed at a wide spectrum of platforms, both dedicated hardware and software ranging from 32-bit embedded processors to modern PC processors with SIMD units and multiple cores. For the more constrained devices, KETJE is a very interesting alternative.

KEYAK is described in layered fashion, each with its own functionality and properties. First, it builds on round-reduced versions of the KECCAK-f[800] and KECCAK-f[1600] permutations [6]. Then, it uses the duplex construction on top of one of these permutations [4]. Above the duplex construction is the DUPLEXWRAP authenticated encryption mode, which is an improved version of SPONGEWRAP [4]. Finally, KEYAK instances come on top of DUPLEXWRAP.

KEYAK includes both serial and parallel instances. The serial instances use the DUPLEXWRAP mode directly after absorbing the key and the nonce in a specific coding. The parallel instances are defined through the KEYAKLINES mode, which builds upon DUPLEXWRAP in a way that is consistent with the serial instance definitions. Instances that best exploit 128-bit and 256-bit SIMD instructions are proposed.

After introducing some notation, basic definitions and the KECCAK-p permutations in Sections 1 and 2, we recall the duplex construction in Section 3, followed by the specification of the DUPLEXWRAP mode in Section 4. We specify KEYAK and KEYAKLINES in Section 5 and finally explain how KEYAK addresses the CAESAR call for proposals in Section 6.

1 Definitions

1.1 Notation

A bit is an element of $\mathbb{Z}_2$. A n-bit string is a sequence of bits represented as an element of $\mathbb{Z}_2^n$. By convention the first bit in the sequence is written on the left side, i.e., the first element in the string $(b_0, b_1, \ldots, b_{n-1})$ is b_0. The set of bit strings of all lengths is denoted $\mathbb{Z}_2^*$ and is defined as

$$\mathbb{Z}_2^* = \cup_{i=0}^{\infty} \mathbb{Z}_2^i.$$

Similarly, the set of all binary strings of length 0 up to n is denoted by $\mathbb{Z}_2^{\leq n}$, i.e.,

$$\mathbb{Z}_2^{\leq n} = \cup_{i=0}^{n} \mathbb{Z}_2^i.$$

The length in bits of a string s is denoted $|s|$. The concatenation of two strings a and b is denoted $a||b$. In some cases, where it is clear from the context, the concatenation is simply denoted ab.

1.2 Of bits and bytes

A byte is a string of 8 bits, i.e., an element of $\mathbb{Z}_2^8$. The byte $(b_0, b_1, \ldots, b_7)$ can also be represented by the integer value $\sum_i 2^i b_i$ written in hexadecimal. E.g., the byte $(0,1,1,0,0,1,0,1)$ can be equivalently written as `0xA6`. The function $\text{enc}_8(x)$ encodes the integer x, with $0 \leq x \leq 255$, as a byte with value x. When the length of a bit string is a multiple of 8, it can also be represented as a sequence of bytes, and vice-versa. E.g., the bit string $(0,1,1,0,0,1,0,1,0,0,1,1,1,1,1,1)$ can also be written as the sequence $(0,1,1,0,0,1,0,1)$ $(0,0,1,1,1,1,1,1)$ or `0xA6 0xFC`.

1.3 Padding rules

We use two different padding rules:

- The *simple padding*, denoted $\text{pad10}^*[r](|M|)$, returns a bit string 10^q with $q = (-|M| - 1) \bmod r$. When r is divisible by 8 and M is a sequence of bytes, then $\text{pad10}^*[r](|M|)$ returns the byte string `0x01` $\texttt{0x00}^{(q-7)/8}$.
- The *multi-rate padding*, denoted $\text{pad10}^*1[r](|M|)$, returns a bitstring 10^q1 with $q = (-|M| - 2) \bmod r$ [4]. When r is divisible by 8 and M is a sequence of bytes, then $\text{pad10}^*1[r](|M|)$ returns the byte string `0x01` $\texttt{0x00}^{(q-14)/8}$ `0x80`.

1.4 Key pack

For a key K, we define a *key pack* of l bits as

$$\text{keypack}(K, l) = \text{enc}_8(l/8)||K||\text{pad10}^*[l - 8](|K|),$$

where the key K is at most $(l - 9)$-bit long and where l is a multiple of 8 not greater than 255×8. That is, the key pack consists of

- a first byte indicating its whole length in bytes, followed by
- the key itself, followed by
- simple padding.

For instance, the 64-bit key $K =$ `0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF` yields

$$\text{keypack}(K, 144) = \texttt{0x12 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF 0x01 0x00}^8.$$

The purpose of the key pack is to have a uniform way of encoding a secret key as prefix of a string input.

1.5 Converting a string into blocks

When the input strings (associated data, plaintext or ciphertext) need to be cut into blocks of some given length ρ bits, we will use the method defined in Algorithm 1. The function also receives a parameter P that specifies the number of parallel lines of blocks to generate. If $P = 1$, the string is simply cut into blocks of ρ bits, except for the last one, which can be shorter. For $P > 1$, the blocks are spread cyclically onto P parts, each containing the same number of blocks. The process is illustrated in Figure 1.

For a 1-dimensional sequence of blocks $X \in (\mathbb{Z}_2^*)^*$, the number of blocks in X is denoted $\|X\|$.

2 The Keccak-p permutations

The Keccak-p permutations are derived from the Keccak-f permutations [6] and have a tunable number of rounds. A Keccak-p permutation is defined by its width $b = 25 \times 2^\ell$, with $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, and its number of rounds n_r. In a nutshell, Keccak-$p[b, n_\text{r}]$ consists in the application of the *last* n_r rounds of Keccak-$f[b]$. When $n_\text{r} = 12 + 2\ell$, Keccak-$p[b, n_\text{r}] =$ Keccak-$f[b]$.

Algorithm 1 Cutting a string into P lines of blocks of ρ bits, and putting it back together.

Interface: $S_{\star,\star} = \textsc{blocks}(S, \rho, P)$ with $S \in \mathbb{Z}_2^*$, $\rho, P \in \mathbb{N}_{>0}$ and $S_{\star,\star} \in (\mathbb{Z}_2^{\leq\rho})^{P \times n}$
Initialize $S_{\star,\star}$ as an array of $P \times n$ empty blocks, with $n = \left\lceil \frac{|S|}{P \times \rho} \right\rceil$.
Cut S into blocks of ρ bits, except for the last block, which can be shorter.
Distribute these blocks in the first column, then in the second column, …, i.e., starting from $S_{0,0}$, then $S_{1,0}$, …, then $S_{P-1,0}$, then $S_{0,1}$, etc.
return $S_{\star,\star}$

Interface: $S = \textsc{blocks}^{-1}(S_{\star,\star}, \rho, P)$ with $S_{\star,\star} \in (\mathbb{Z}_2^{\leq\rho})^{P \times n}$, $\rho, P \in \mathbb{N}_{>0}$, $n \in \mathbb{N}$ and $S \in \mathbb{Z}_2^*$
Initialize S to the empty string
for $j = 0$ to $n - 1$ **do**
  for $i = 0$ to $P - 1$ **do**
    $S \leftarrow S || S_{i,j}$
return S

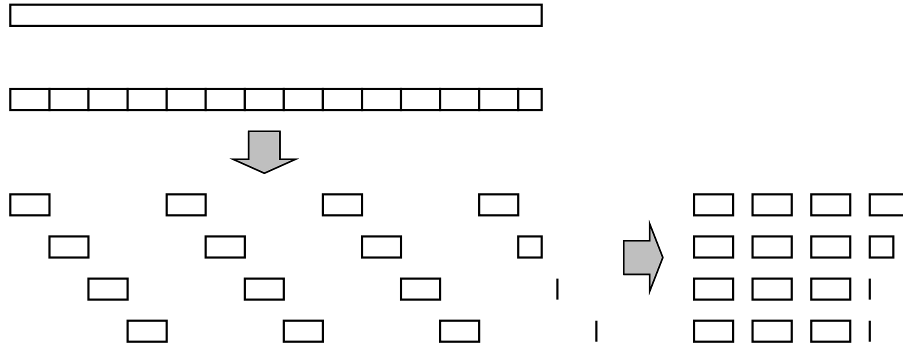

Figure 1: Illustration of $\textsc{blocks}(S, \rho, P)$ in Algorithm 1. In this figure, the input string S, with a length comprised between 13ρ and 14ρ, is cut in $P = 4$ lines. Each line contains 4 blocks: the first line contains 4 full blocks, the second line contains 3 full blocks and 1 partial block, and the last two lines contain 3 full blocks and 1 empty block.

The permutation Keccak-$p[b, n_r]$ is described as a sequence of operations on a state a that is a three-dimensional array of elements of GF(2), namely $a[5,5,w]$, with $w = 2^\ell$. The expression $a[x,y,z]$ with $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_w$, denotes the bit at position (x,y,z). It follows that indexing starts from zero. The mapping between the bits of s and those of a is $s[w(5y+x)+z] = a[x,y,z]$. Expressions in the x and y coordinates should be taken modulo 5 and expressions in the z coordinate modulo w. We may sometimes omit the $[z]$ index, both the $[y,z]$ indices or all three indices, implying that the statement is valid for all values of the omitted indices.

Keccak-$p[b, n_r]$ is an iterated permutation, consisting of a sequence of n_r rounds R, indexed with i_r from $12+2\ell-n_r$ to $12+2\ell-1$. Note that i_r, the round number, does not necessarily start from 0. A round consists of five steps:

$$\mathrm{R} = \iota \circ \chi \circ \pi \circ \rho \circ \theta, \text{ with}$$

$$\begin{aligned}
\theta: \quad a[x,y,z] &\leftarrow a[x,y,z] + \sum_{y'=0}^{4} a[x-1,y',z] + \sum_{y'=0}^{4} a[x+1,y',z-1],\\
\rho: \quad a[x,y,z] &\leftarrow a[x,y,z-(t+1)(t+2)/2],\\
&\quad \text{with } t \text{ satisfying } 0 \le t < 24 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in } \mathrm{GF}(5)^{2\times 2},\\
&\quad \text{or } t = -1 \text{ if } x = y = 0,\\
\pi: \quad a[x,y] &\leftarrow a[x',y'], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix},\\
\chi: \quad a[x] &\leftarrow a[x] + (a[x+1]+1)a[x+2],\\
\iota: \quad a &\leftarrow a + \mathrm{RC}[i_r].
\end{aligned}$$

The additions and multiplications between the terms are in GF(2). With the exception of the value of the round constants $\mathrm{RC}[i_r]$, these rounds are identical. The round constants are given by (with the first index denoting the round number)

$$\mathrm{RC}[i_r][0,0,2^j-1] = \mathrm{rc}[j+7i_r] \text{ for all } 0 \le j \le \ell,$$

and all other values of $\mathrm{RC}[i_r][x,y,z]$ are zero. The values $\mathrm{rc}[t] \in \mathrm{GF}(2)$ are defined as the output of a binary linear feedback shift register (LFSR):

$$\mathrm{rc}[t] = \left(x^t \bmod x^8 + x^6 + x^5 + x^4 + 1\right) \bmod x \text{ in } \mathrm{GF}(2)[x].$$

Note that the round index i_r can be considered modulo 255, the period of the LFSR above.

3 The duplex construction

The duplex construction allows both to input and to output a data block per call to the underlying permutation [4]. It is shown to formally reduce to the sponge construction, although the supported functionality of both constructions differ and typically address different applications. Besides authenticated encryption, the duplex construction finds a number of applications, such as a reseedable pseudo-random bit sequence generators and a sponge variant that overwrites part of the state with the input block rather than to XOR it in.

3.1 Specification

The *duplex construction* $\textsc{duplex}[f, \text{pad}, r]$ uses a fixed-length transformation (or permutation) f, a padding rule "pad" and a parameter bitrate r [4]. Unlike a sponge function that is stateless in between calls, the duplex construction accepts calls that take an input string and returns an output string depending on all inputs received so far. We call an instance of the duplex construction a *duplex object*, which we denote D in our descriptions. We prefix the calls made to a specific duplex object D by its name D and a dot.

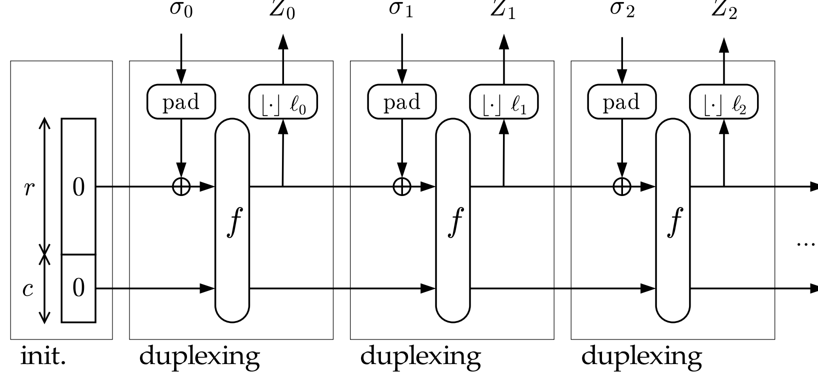

Figure 2: The duplex construction

The duplex construction works as follows. A duplex object D has a state of b bits. Upon initialization all the bits of the state are set to zero. From then on, one can send to it D.duplexing(σ, ℓ) calls, with σ an input string and ℓ the requested number of bits.

Algorithm 2 The duplex construction $\textsc{duplex}[f, \text{pad}, r]$

Require: $r < b$
Require: $\rho_{\max}(\text{pad}, r) > 0$
Require: $s \in \mathbb{Z}_2^b$ (maintained across calls)

Interface: D.initialize()
$s = 0^b$

Interface: $Z = D.\text{duplexing}(\sigma, \ell)$ with $\ell \le r$, $\sigma \in \mathbb{Z}_2^{\le \rho_{\max}(\text{pad},r)}$, and $Z \in \mathbb{Z}_2^{\ell}$
$P = \sigma || \text{pad}[r](|\sigma|)$
$s = s \oplus (P || 0^{b-r})$
$s = f(s)$
return $\lfloor s \rfloor_\ell$

The maximum number of bits ℓ one can request is r and the input string σ shall be short enough such that after padding it results in a single r-bit block. We call the maximum length of σ the *maximum duplex rate* and denote it by $\rho_{\max}(\text{pad}, r)$. For the multi-rate padding, we have

$$\rho_{\max}(\text{pad}10^*1, r) = r - 2.$$

The duplex construction is illustrated in Figure 2, and Algorithm 2 provides a formal definition.

3.2 Rationale

The generic security of the duplex construction is equivalent to that of the sponge construction, thanks to the Duplexing-sponge lemma [4], which we repeat here:

Lemma 1 (Duplexing-sponge lemma [4]). *If we denote the input to the i-th call to a duplex object by (σ_i, ℓ_i) and the corresponding output by Z_i we have:*

$$Z_i = D.\text{duplexing}(\sigma_i, \ell_i) = \text{sponge}(\sigma_0||\text{pad}_0||\sigma_1||\text{pad}_1||\dots||\sigma_i, \ell_i)$$

with pad_i *a shortcut notation for* $\text{pad}[r](|\sigma_i|)$.

The generic security of the sponge construction is in turn analyzed in the framework of indifferentiability in [2] and, when keyed, for indistinguishability from a random oracle in [5].

Theorem 1 ([2]). *The $\mathcal{RO}$ differentiating advantage of the sponge construction calling a random permutation is upper bound by:*

$$1 - \prod_{i=0}^{N-1} \left(\frac{1 - \frac{i+1}{2^c}}{1 - \frac{i}{2^{r+c}}} \right) \approx \frac{N^2}{2^{c+1}}$$

with N the cost of the queries, i.e., the total number of blocks queried to the sponge function, the permutation or its inverse.

Theorem 2 ([5]). *The advantage of distinguishing* $\textsc{sponge}[f, \text{pad}, r](K||\cdot, \ell)$ *from a random oracle, with f a random permutation and K uniformly distributed over $\mathbb{Z}_2^k$, is upper bounded by:*

$$\frac{M^2 + 4MN}{2^{c+1}} + \frac{N}{2^k},$$

where M is the data complexity (i.e., the total number of blocks queried to the keyed sponge function) and N the computational complexity (i.e., the total number of blocks queried to the permutation or its inverse).

Note that the proof of Theorem 2 is work in progress. Our current investigations tend to confirm the above result, but we might need to slightly refine the bound once the proof is completed.

4 The serial authenticated encryption mode DuplexWrap

We consider authenticated encryption as a process that takes as input a header A and a data body B and that returns a cryptogram C and a tag T. We denote this operation by the term *wrapping* and the reverse operation of taking a header A, a cryptogram C and a tag T and returning the data body B if the tag T is correct by the term *unwrapping*.

Similar to SpongeWrap, DuplexWrap supports *sessions*, allowing the processing of several messages (each with associated data), where the tag for each message authenticates the full sequence of messages rather than only the message to which it was appended. The requirement of nonce uniqueness plays at the level of the session. Within a session, different messages have no explicit message number or nonce. However, they must be processed in order for the tags to verify. An alternative way to see this concept of session is that it supports intermediate tags.

In other words, the process of authenticating and encrypting works on a sequence of header-body pairs $(\overline{A,B}) = (A^{(1)}, B^{(1)}, A^{(2)}, \ldots, A^{(n)}, B^{(n)})$ in such a way that the authenticity is guaranteed not only on each (A, B) pair but also on the sequence received so far. The use of such a sequence and intermediate tags is illustrated in Figure 3 and further formalized in [4, Section 2.1].

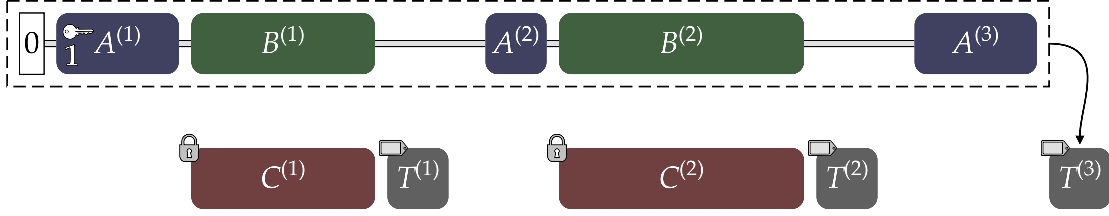

Figure 3: A session in DuplexWrap

The requirements on the way DuplexWrap is used are the following. The first header $A^{(1)}$ must contain the secret key and can also contain a nonce and/or associated data. For the encryption of a body $B^{(1)}$ in the first wrap call, $A^{(1)}$ must be secret and unique. This can be either because $A^{(1)}$ contains a key that has not been used already (and will never be used again), or because $A^{(1)}$ contains a nonce. As for a stream cipher, not respecting this requirement means that the adversary can learn the bitwise difference between two plaintext bodies.

Second, the encryption of a body $B^{(n)}$ requires that the previously processed sequence of header-body pairs is unique. This is also known as the *nonce requirement*. More formally, it specifies that for any two queries $(\overline{A,B})$ and $(\overline{A',B'})$ of equal length n, we have

$$\text{pre}(\overline{A,B}) = \text{pre}(\overline{A',B'}) \Rightarrow B^{(n)} = B'^{(n)},$$

with $\text{pre}(\overline{A,B}) = (A^{(1)}, B^{(1)}, A^{(2)}, \ldots, B^{(n-1)}, A^{(n)})$ the sequence with the last body omitted. In other words, the encryption of two different bodies requires that the blocks preceding these bodies are different as well.

Note that generating tags requires that $A^{(1)}$ is secret (i.e., it contains the secret key) but not necessarily that it is unique.

We now overview the way DuplexWrap works. Like for duplex, we use an object-oriented description. A DuplexWrap object W internally uses a duplex object D and it is parameterized with the permutation f and the maximum block length ρ. The multi-rate padding is fixed as the padding rule. Upon initialization of a DuplexWrap object, it initializes D, i.e., its state is set to zero.

When receiving a $W.\text{wrap}(A, B, \ell)$ request, it forwards the blocks of the header A and the body B to D. It generates the cryptogram C block by block $C_i = B_i \oplus Z_i$ with Z_i the response of D to the previous $D.\text{duplexing}()$ call. The ℓ-bit tag T is the response of D to the last body block (possibly extended with the response to additional $D.\text{duplexing}()$ calls in case $\ell > \rho$). Finally it returns the cryptogram C and the tag T.

When receiving a $W.\text{unwrap}(A, C, T)$ request, it forwards the blocks of the header A to D. It decrypts the data body B block by block $B_i = C_i \oplus Z_i$ with Z_i the response of D to the previous $D.\text{duplexing}()$ call. The response of D to the last body block (possibly extended) is compared with the tag T received as input. If the tag is valid, it returns the data body B; otherwise, it returns an error. Note that in implementations one may impose additional constraints, such as DuplexWrap objects dedicated to either only wrapping

or only unwrapping. Additionally, if message origin authentication is a requirement, the application using the DuplexWrap object should not accept tags with length below a minimum length t, otherwise it would trivially accept messages with an empty tag. As a countermeasure against (side channel) attacks, an application may abort the session as soon as it receives an incorrect tag.

Before being forwarded to D, every block is extended with so-called *frame bits*. The purpose of the frame bits is twofold. First, it delimits blocks belonging to the different headers and bodies. Second, it ensures domain separation between the production of the key stream and of the tag blocks. The rate ρ of the DuplexWrap mode determines the size of the blocks and hence the maximum number of bits processed per call to f. Its upper bound is $r - 4$ due to the inclusion of two frame bits and two padding bits per block. Note that there are two distinct rates: the rate r of the duplex construction, which determines the attainable security strength with $c = b - r$, and the rate ρ of DuplexWrap, which determines the maximum length of blocks.

DuplexWrap puts no restriction in the way the input strings are cut into blocks, provided that each block is at most $\rho = r - 4$ bits and that the same sequence of block lengths is given when wrapping and unwrapping. A block is thus an element of $\mathbb{Z}_2^{\leq \rho}$. The inputs and outputs of the wrapping and unwrapping calls are sequences of blocks, except for the tag. The motivation for the application or mode calling DuplexWrap to control the size of blocks is to allow that parallel lines have the same number of blocks to process in KeyakLines.

In the case where only authentication is needed, the body can be absent (no blocks) and DuplexWrap avoids an unnecessary call to the duplex object. In all cases, however, the header must contain at least one (possibly empty) block.

DuplexWrap provides an explicit *"forget" call* that processes the state in an irreversible way so as to ensure forward secrecy. If for some reason the state is recovered by an attacker, she cannot determine the state before the "forget" call without guessing at least c bits.

4.1 Specification

DuplexWrap is defined in Algorithms 3 and 4. The process of wrapping is illustrated in Figures 4 and 5. The "forget" call is illustrated in Figure 6.

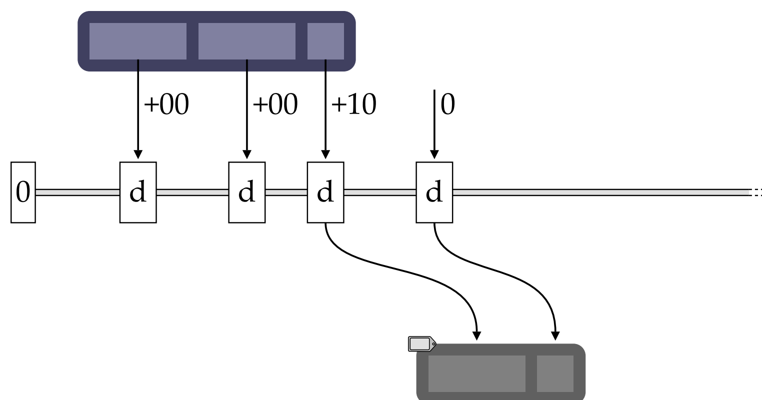

Figure 4: Wrapping a header only with DuplexWrap

Algorithm 3 Wrapping in DuplexWrap$[f, \rho]$.

Require: $D = \text{DUPLEX}[f, \text{pad}10^*1, r = \rho + 4]$, initialized before the first (un)wrap call

Interface: $(C, T) = W.\text{wrap}(A, B, \ell)$ with $A \in (\mathbb{Z}_2^{\leq \rho})^+$, $B \in (\mathbb{Z}_2^{\leq \rho})^*$, sequences of blocks ($B$ can be empty), $C \in (\mathbb{Z}_2^{\leq \rho})^*$, $\ell \geq 0$, and $T \in \mathbb{Z}_2^\ell$

for $i = 0$ to $\|A\| - 2$ **do**
  $D.\text{duplexing}(A_i||00, 0)$
if $\|B\| \geq 1$ **then**
  $Z = D.\text{duplexing}(A_{\|A\|-1}||01, |B_0|)$
  $C_0 = B_0 \oplus Z$
  for $i = 0$ to $\|B\| - 2$ **do**
    $Z = D.\text{duplexing}(B_i||11, |B_{i+1}|)$
    $C_{i+1} = B_{i+1} \oplus Z$
  $T = D.\text{duplexing}(B_{\|B\|-1}||10, \rho)$
else
  $T = D.\text{duplexing}(A_{\|A\|-1}||10, \rho)$
while $|T| < \ell$ **do**
  $T = T||D.\text{duplexing}(0, \rho)$
$T = \lfloor T \rfloor_\ell$
return (C, T)

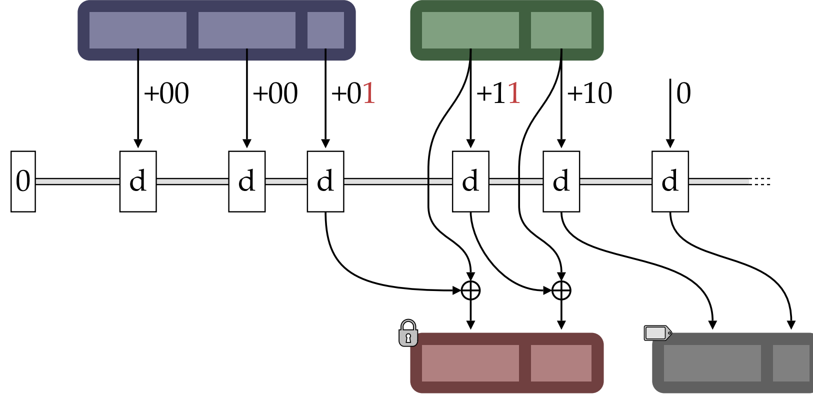

Figure 5: Wrapping a header and a body with DuplexWrap

4.2 Comparison with SpongeWrap

DuplexWrap is very similar to SpongeWrap [4]. The differences are the following:

1. DuplexWrap puts no restriction in the way the input strings are cut into blocks. To reflect this in the interface, the inputs and outputs of the wrapping and unwrapping calls are sequences of blocks, except for the tag.

2. In the case only authentication is needed, the body can be absent and DuplexWrap avoids one call to the duplex object compared to SpongeWrap.

Algorithm 4 Unwrapping and forgetting in DuplexWrap$[f, \rho]$.

Require: $D = \text{duplex}[f, \text{pad}10^*1, r = \rho + 4]$, initialized before the first (un)wrap call

Interface: $(B, T) = W.\text{internalUnwrap}(A, C, \ell)$ with $A \in (\mathbb{Z}_2^{\leq\rho})^+$, $C \in (\mathbb{Z}_2^{\leq\rho})^*$ sequences of blocks (C can be empty), $B \in (\mathbb{Z}_2^{\leq\rho})^*$, $\ell \geq 0$, and $T \in \mathbb{Z}_2^{\ell}$

for $i = 0$ to $\|A\| - 2$ **do**
  $D.\text{duplexing}(A_i||00, 0)$
if $\|C\| \geq 1$ **then**
  $Z = D.\text{duplexing}(A_{\|A\|-1}||01, |C_0|)$
  $B_0 = C_0 \oplus Z$
  for $i = 0$ to $\|C\| - 2$ **do**
    $Z = D.\text{duplexing}(B_i||11, |C_{i+1}|)$
    $B_{i+1} = C_{i+1} \oplus Z$
  $T = D.\text{duplexing}(B_{\|B\|-1}||10, \rho)$
else
  $T = D.\text{duplexing}(A_{\|A\|-1}||10, \rho)$
while $|T| < \ell$ **do**
  $T = T||D.\text{duplexing}(0, \rho)$
return $(B, \lfloor T \rfloor_\ell)$

Interface: $B = W.\text{unwrap}(A, C, T)$ with $A \in (\mathbb{Z}_2^{\leq\rho})^+$, $C \in (\mathbb{Z}_2^{\leq\rho})^*$ sequences of blocks (C can be empty), $B \in (\mathbb{Z}_2^{\leq\rho})^* \cup \{\text{error}\}$, and $T \in \mathbb{Z}_2^*$

Let $(B, T') = W.\text{internalUnwrap}(A, C, |T|)$
if $T = T'$ **then**
  return B
else
  return error

Interface: $W.\text{forget}()$, requiring $\rho \geq c$
$Z = D.\text{duplexing}(\text{empty string}, \rho)$
$D.\text{duplexing}(Z, 0)$

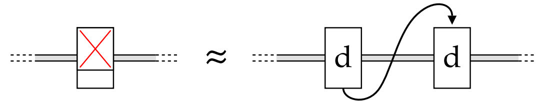

Figure 6: Forgetting in DuplexWrap

3. For uniformity of the description, DuplexWrap does not propose a specific initialization call for absorbing the key. In the first wrap (or unwrap) call, the header $A^{(1)}$ must contain the secret key but can also contain a nonce and/or associated data.
4. In addition to the unwrap call, we define the W.internalUnwrap(A, C, ℓ) interface that returns the expected tag and the ciphertext. This function will be needed in the layer above.
5. DuplexWrap provides an explicit "forget" call that processes the state in an irreversible way so as to ensure forward secrecy.
6. DuplexWrap uses two frame bits per duplex call instead of one in SpongeWrap.
7. DuplexWrap is instantiated explicitly with the multi-rate padding. So the rate is fixed to $r = \rho + 4$ so that $\rho \leq \rho_{\max}(\text{pad10}^*1, r) - 2$, in order to allow for two DuplexWrap-level frame bits and two multi-rate padding bits.

4.3 Rationale

The security of DuplexWrap follows the proof in [4, Section 5.2], with Lemma 7 in that paper replaced by Lemma 2 below.

Lemma 2. *Let $(\overline{A, B})$ be a sequence of header-body pairs, each made of sequences of blocks. Then, the mapping from $(\overline{A, B})$ to the corresponding sequence of inputs $(\sigma_0, \sigma_1, \ldots, \sigma_n)$ to the duplexing calls in Algorithm 3 is injective.*

Proof. We show that from $(\sigma_0, \sigma_1, \ldots, \sigma_n)$ we can always recover $(\overline{A, B})$.

The first header $A^{(1)}$ can be found by looking for the first block σ_i that ends with frame bits 01 or 10; the blocks of $A^{(1)}$ are made of the blocks σ_j, $j \leq i$, each with their last two bits removed.

If the last block σ_i ended with 01, at least one body block follows. To find the first body $B^{(1)}$, we follow the same procedure as for $A^{(1)}$, except that we look for the first block $\sigma_{i'}$, $i' > i$, that ends with 10. Otherwise, i.e., if σ_i ended with 10, then $B^{(1)}$ contains no blocks.

We then repeat the procedure, but instead of recovering $A^{(1)}$ and $B^{(1)}$, we recover $A^{(2)}$ and $B^{(2)}$, and then $A^{(3)}$ and $B^{(3)}$, and so on.

At all times, blocks containing only the frame bit 0 are skipped, as they can only be produced when extending the tag in line 14 of Algorithm 3. Also, every time an empty block is encountered, this block and the next one are skipped, as they can only be produced in a "forget" call in lines 23 and 24 of Algorithm 4. □

Combined with the results of Theorem 1, and using the fact that the number of queries is lower bounded by the number of blocks queried N, this yields the following result.

Theorem 3. *The authenticated encryption mode DuplexWrap$[f, \rho]$ defined in Section 4 satisfies*

$$\mathrm{Adv}^{\mathrm{priv}}_{\textsc{DuplexWrap}[f,\rho]}(\mathcal{A}) < N2^{-k} + \frac{N(N+1)}{2^{c+1}} \text{ and}$$
$$\mathrm{Adv}^{\mathrm{auth}}_{\textsc{DuplexWrap}[f,\rho]}(\mathcal{A}) < N2^{-k} + \frac{N(N+1)}{2^{c+1}} + 2^{-t},$$

against any single adversary $\mathcal{A}$ if $K \xleftarrow{\$} \mathbb{Z}_2^k$, tags of $\ell \geq t$ bits are used, f is a randomly chosen permutation, and N is the total number of blocks queried to DuplexWrap, to f and to its inverse.

Additionally, if $A^{(1)}$ starts with the secret key, we can use Theorem 2. Using the fact that the number of queries is lower bounded by the number of blocks queried M, this yields the following result. Theorem 4 provides a better bound than the theorem above by separating the queries made to the keyed duplex object under attack (that is, the data complexity), from those made separately to the permutation or its inverse (that is, the time complexity). Thanks to this theorem, we can go beyond the usual bound of $2^{c/2}$ for the time complexity by imposing a limit to the data complexity.

Theorem 4. *The authenticated encryption mode* $\textsc{DuplexWrap}[f,\rho]$ *defined in Section 4 satisfies*

$$\begin{aligned}\mathrm{Adv}^{\mathrm{priv}}_{\textsc{DuplexWrap}[f,\rho]}(\mathcal{A}) &< (M+N)2^{-k} + \frac{M^2+4MN}{2^{c+1}} \text{ and} \\ \mathrm{Adv}^{\mathrm{auth}}_{\textsc{DuplexWrap}[f,\rho]}(\mathcal{A}) &< (M+N)2^{-k} + \frac{M^2+4MN}{2^{c+1}} + 2^{-t},\end{aligned}$$

against any single adversary $\mathcal{A}$ *if* $K \xleftarrow{\$} \mathbb{Z}_2^k$, *tags of* $\ell \geq t$ *bits are used,* f *is a randomly chosen permutation,* M *is the data complexity (i.e., the total number of blocks queried to the keyed sponge function or duplex object) and* N *the time complexity (i.e., the total number of blocks queried to permutation or its inverse).*

5 Keyak

There are four instances of Keyak, parameterized by the permutation f used and by their degree of parallelism P. In order of increasing state sizes, the instances are:

- **River Keyak**, with $f = \textsc{Keccak}\text{-}p[800, n_r = 12]$ and $P = 1$;
- **Lake Keyak**, with $f = \textsc{Keccak}\text{-}p[1600, n_r = 12]$ and $P = 1$; (this is the primary recommendation)
- **Sea Keyak** with $f = \textsc{Keccak}\text{-}p[1600, n_r = 12]$ and $P = 2$; and
- **Ocean Keyak** with $f = \textsc{Keccak}\text{-}p[1600, n_r = 12]$ and $P = 4$.

All these instances share the same capacity, namely, $c = 252$. They target 128-bit security if the online data complexity is below 2^{123} blocks, by using Theorem 4 with $M \leq 2^{123}$.

All these instances take a 128-bit public message number (or nonce), but no private message number, and produce a 128-bit MAC, which can be truncated by the user if desired. If not truncated, the gap between the ciphertext and the plaintext length is exactly 128 bits. The key size is variable, with a minimum of 128 bits for the targeted security, and up to a maximum of 224 bits, as a possible countermeasure against multi-target attacks.

The serial ($P = 1$) instances are defined on top of DuplexWrap directly, while the parallel ($P > 1$) instances are described on top of KeyakLines, in turn using DuplexWrap.

5.1 Serial instances

A *serial* Keyak instance is an application of DuplexWrap parameterized by the permutation f and assumes $P = 1$. It uses a DuplexWrap object

$$W = \textsc{DuplexWrap}[f, \rho],$$

with $\rho = b - c - 4$, where b is f's width.

With inputs M for the input message, AD for the associated data, N for the 128-bit public message number and K for the key, the encryption returns the authenticated ciphertext $C||T$, with

$$(C, T) = W.\text{wrap}(\text{keypack}(K, 240)||\text{enc}_8(1)||\text{enc}_8(0)||N||\text{AD}, M, 128),$$

where the input strings are converted into blocks via $\textsc{blocks}(\cdot, \rho, 1)$ and output blocks are converted back to strings via $\textsc{blocks}^{-1}(\cdot, \rho, 1)$.

With inputs $C||T$ for the authenticated ciphertext (with $|T| = 128$ by default), AD for the associated data, N for the 128-bit public message number and K for the key, the decryption returns the plaintext M or an error, with

$$M = W.\text{unwrap}(\text{keypack}(K, 240)||\text{enc}_8(1)||\text{enc}_8(0)||N||\text{AD}, C, T),$$

where the input strings are converted into blocks via $\textsc{blocks}(\cdot, \rho, 1)$ and output blocks are converted back to strings via $\textsc{blocks}^{-1}(\cdot, \rho, 1)$.

Further input message and associated data pairs can be processed as a session by calling

$$(C, T) = W.\text{wrap}(\text{AD}, M, 128) \quad \text{or} \quad M = W.\text{unwrap}(\text{AD}, C, T)$$

using the same DuplexWrap object W and without the need of a new public message number. The produced tags also cover the previously processed input message and associated data pairs, as explained in Section 4. Optionally, forward secrecy can be ensured by calling W.forget() between pairs.

The proposed instances are:

- **Lake Keyak** is the primary recommendation. We set $f = \text{Keccak-}p[1600, n_r = 12]$ and $c = 252$ so that $\rho = 1344$.
- **River Keyak** is a secondary recommendation, which may be of interest for its smaller state size. We set $f = \text{Keccak-}p[800, n_r = 12]$ and $c = 252$ so that $\rho = 544$.

5.2 The parallelizable authenticated encryption mode KeyakLines

In Algorithm 5, we define a parallel application of DuplexWrap. KeyakLines is parameterized by P, the degree of parallelism, that is, the number of DuplexWrap objects whose bulk work can be performed independently.

KeyakLines is specific to Keyak, as it uses specific coding to make all Keyak instances consistent, including the (non-KeyakLines) serial ones, i.e., the first header always starts with

$$\text{keypack}(K, 240)||\text{enc}_8(P)||\text{enc}_8(i)||N$$

with $0 \leq i \leq P - 1$, regardless whether $P = 1$ or $P > 1$. Unlike DuplexWrap, there is an explicit initialization method, which takes a key and a nonce as input. Then, the wrap and unwrap methods can process P parts of the associated data and then P parts of the input message in parallel.

The input strings (associated data, plaintext or ciphertext) are cut into blocks of ρ bits using Algorithm 1. The P sequences contain an equal number of blocks, so that each DuplexWrap object processes the same number of blocks, and this is expected to make the implementation easier.

For tag production, the DuplexWrap object with index 0 gathers tags produced by the remaining $P - 1$ other objects as chaining value, of size $8\lceil \frac{c}{8} \rceil = 256$ bits each, before producing its own.

Algorithm 5 Wrapping and unwrapping in KeyakLines$[f, \rho, P]$.

Require: P, an integer between 2 and 255
Require: $W[0 \dots P-1]$, an array of P objects DuplexWrap$[f, \rho]$

Interface: W.initialize(K, N) with $K \in \mathbb{Z}_2^{\leq 224}$, $N \in \mathbb{Z}_2^*$
for $i = 0$ to $P-1$ **do**
  $W[i]$.wrap(blocks(keypack$(K, 240)$||enc$_8(P)$||enc$_8(i)$||$N, \rho, 1)$, no blocks, 0)

Interface: $(C, T) = W.\text{wrap}(A, B, \ell)$ with $A, B \in \mathbb{Z}_2^*$, $\ell \geq 0$, $C \in \mathbb{Z}_2^{|B|}$ and $T \in \mathbb{Z}_2^{\ell}$
Let $A_{\star,\star} = \text{blocks}(A, \rho, P)$, or $A_{\star,\star}$ is a $P \times 1$-array of empty blocks if $|A| = 0$
Let $B_{\star,\star} = \text{blocks}(B, \rho, P)$
for $i = 0$ to $P-1$ **do**
  $(C_{i,\star}, T'_i) = W[i].\text{wrap}(A_{i,\star}, B_{i,\star}, 8\lceil \frac{c}{8} \rceil)$
$T = W[0].\text{wrap}(\text{blocks}(T'_1 || T'_2 || \dots || T'_{P-1}, \rho, 1), \text{no blocks}, \ell)$
return $(\text{blocks}^{-1}((C_{0,\star}, \dots, C_{P-1,\star}), \rho, P), T)$

Interface: $B = W.\text{unwrap}(A, C, T)$ with $A, C, T \in \mathbb{Z}_2^*$, $B \in \mathbb{Z}_2^{|C|} \cup \{\text{error}\}$
Let $A_{\star,\star} = \text{blocks}(A, \rho, P)$, or $A_{\star,\star}$ is a $P \times 1$-array of empty blocks if $|A| = 0$
Let $C_{\star,\star} = \text{blocks}(C, \rho, P)$
for $i = 0$ to $P-1$ **do**
  $(B_{i,\star}, T'_i) = W[i].\text{internalUnwrap}(A_{i,\star}, C_{i,\star}, 8\lceil \frac{c}{8} \rceil)$
Let result $= W[0].\text{unwrap}(\text{blocks}(T'_1 || T'_2 || \dots || T'_{P-1}, \rho, 1), \text{no blocks}, T)$
if result = success **then**
  return $\text{blocks}^{-1}((B_{0,\star}, \dots, B_{P-1,\star}), \rho, P)$
else
  return error

Interface: W.forget()
for $i = 0$ to $P-1$ **do**
  $W[i]$.forget()

5.3 Parallelizable instances

A *parallelizable* KEYAK instance is an application of KEYAKLINES parameterized by the permutation f and the degree of parallelism $P > 1$. It uses a KEYAKLINES object

$$W = \text{KeyakLines}[f, \rho, P],$$

with $\rho = b - c - 4$, where b is f's width.

With inputs M for the input message, AD for the associated data, N for the public message number and K for the key, the encryption returns the authenticated ciphertext $C||T$, starting by calling $W.\text{initialize}(K, N)$ and then

$$(C, T) = W.\text{wrap}(\text{AD}, M, 128).$$

With inputs $C||T$ for the authenticated ciphertext (with $|T| = 128$ by default), AD for the associated data, N for the public message number and K for the key, the decryption returns the plaintext M or an error, starting by calling $W.\text{initialize}(K, N)$ and then

$$M = W.\text{unwrap}(\text{AD}, C, T).$$

Further input message and associated data pairs can be processed as a session by calling

$$(C, T) = W.\text{wrap}(\text{AD}, M, 128) \quad \text{or} \quad M = W.\text{unwrap}(\text{AD}, C, T)$$

using the same KEYAKLINES object W. Optionally, forward secrecy can be ensured by calling $W.\text{forget}()$ between pairs.

For both instances, we set $f = \text{Keccak-}p[1600, n_r = 12]$ and $c = 252$ so that $\rho = 1344$. The two instances differ by their degree of parallelism, namely

- **SEA KEYAK** sets $P = 2$ and
- **OCEAN KEYAK** sets $P = 4$.

These instances can be interesting in a number of cases, in particular for exploiting SIMD architectures that the parallel evaluation of the KECCAK round function can benefit from [9]. SEA KEYAK best exploits 128-bit SIMD, while OCEAN KEYAK best exploits 256-bit SIMD.

5.4 Security goals

The claimed security is summarized in Table 1, where the security strength is indicated with the logarithm base 2 of the attack cost and the unit is the execution of the underlying permutation.

For the confidentiality of the plaintext, the public message number N has to be a **nonce**. Reusing the same value of N for more than one session in general breaks the confidentiality of the plaintext. It leaks the bitwise difference between the plaintext messages encrypted under the same M. In the case of a single, accidental, nonce reuse, the situation is just a little bit better than with a stream cipher, as the leakage is limited to the first block of ρ bits where the input messages start to differ. Due to the way DUPLEXWRAP works, the subsequent blocks will not lose confidentiality.

For the integrity of the plaintext and associated data, however, N is not required to be a nonce.

In multi-target attacks against KEYAK the resistance against exhaustive keys may erode from $|K|$ to $|K| - \log_2 n$ with n the number of targets. This is the case if n KEYAK instances

	KEYAK
plaintext confidentiality	$\min(128, \|K\|)$
plaintext integrity	$\min(128, \|K\|, \|T\|)$
associated data integrity	$\min(128, \|K\|, \|T\|)$
public message number integrity	$\min(128, \|K\|, \|T\|)$

Table 1: Security claims for KEYAK, assuming that the online data complexity is below 2^{123} blocks.

are loaded with different keys but the same nonce $|N|$, and an attacker has access to their output when processing the same input. Note that if an upper limit to n is known, one can have a security strength of 128 bits by taking sufficiently long keys: $|K| \geq 128 + \log_2 n_{\max}$. An option that avoids erosion without increasing the length of keys is to impose universal nonce uniqueness. By this we mean that not only the combination (K, N) must be unique, but the nonce N for each KEYAK instance must be unique. Many use cases actually allow this. For example, one can take as nonce the combination of the universally unique IDs of the two communicating devices and a strictly incrementing session counter.

5.5 Rationale

In this section, we give a rationale on the security of KEYAK. The proofs are sketched, rather than formalized, so as to favor the clarity and simplicity of the explanation.

Our security claims for KEYAK match the level of generic attacks of the chosen modes and chosen parameters (mainly the capacity set at $c = 252$). This is because we believe that the permutations KECCAK-$p[1600, n_r = 12]$ and KECCAK-$p[800, n_r = 12]$ are strong enough to avoid exploitable properties, which would translate into attacks in the chosen modes with a complexity lower than the generic ones. In this approach, we have two things to look at:

1. First, we need to analyze generic attacks and their success probability. This usually comes down to proving upper bounds on such probabilities.
2. Second, we need to consider the properties of the underlying permutations and whether they are not exploitable.

Regarding the properties of the underlying permutations, we refer to [3, Chapter 8] for examples of properties that are relevant in the scope of sponge functions, as well as our own and all the third-party cryptanalysis of KECCAK [7]. We note in particular that the algebraic degree of the permutation as a function of the number of rounds most likely reaches a high enough level after 12 rounds [10, 11].

The remainder of this section deals with the generic attacks. We start with the confidentiality part and then discuss the authentication part.

5.5.1 Confidentiality

In KEYAK, the output for key stream and tag are domain separated by the underlying DUPLEXWRAP object(s). It thus makes sense to discuss these two aspects separately.

With $P = 1$, the generic security depends directly on that of DUPLEXWRAP. When $P > 1$, we note that KEYAKLINES uses DUPLEXWRAP in such a way that the key streams generated by the different lines are domain separated by the byte $\text{enc}_8(i)$. We refer to Section 4.3 for more details.

5.5.2 Authentication

We base the reasoning on the fact that the tag generated as output of the wrap call in KeyakLines is formally equivalent to the application of a tree hash mode on top of a keyed sponge function. Thanks to the duplexing-sponge lemma (Section 3.2), the tag (and the key stream) in DuplexWrap can be seen as the output of a sponge function. Furthermore, since the input is prefixed by a secret key, this falls in the definition of a keyed sponge function [5]. Finally, the mode KeyakLines uses P duplex objects, hence the mode equivalently produces P nodes.

We first define the mode $\mathcal{T}$, which takes as input the parameter P and a message m and returns a tag T. It encompasses both the serial Keyak instances with $P = 1$ by following the definitions in Section 5.1 and the parallelizable Keyak instances with $P > 1$ by using KeyakLines. By studying both cases at the same time, proving the soundness of $\mathcal{T}$ will ensure the joint security of multiple Keyak instances with possibly different parameters.

We study the advantage of an adversary trying to distinguish $(\mathcal{T}[\mathcal{KS}[f]], f)$ from $(\mathcal{G}[\mathcal{RO}], f)$, where $\mathcal{KS}$ is the keyed sponge function, f is a random permutation and $\mathcal{G}[\mathcal{RO}]$ is a system calling a random oracle after mapping $(P, m) \in \mathbb{N} \times \mathbb{Z}_2^*$ to a string $s \in \mathbb{Z}_2^*$ in an injective way so as to provide the same interface as $\mathcal{T}$.

We can show that the advantage of the adversary is upper bounded by

$$\frac{2M^2 + 4MN}{2^{c+1}},$$

with M the total cost of the queries to the first component ($\mathcal{T}[\mathcal{KS}[f]]$ or $\mathcal{G}[\mathcal{RO}]$) and N the number of calls made to f or its inverse. Here, the cost M is defined as the number of blocks that $\mathcal{T}$ sends to the keyed sponge. For a query (P, m), this corresponds to a cost of $P\lceil \frac{|m|}{P\rho} \rceil$ plus some term proportional to P.

We follow the game hopping technique [12, 1].

- The first game is the real mode, with an ideal permutation, $(\mathcal{T}[\mathcal{KS}[f]], f)$. The tree hash mode $\mathcal{T}$ receives q queries with total cost M, which are transmitted to the keyed sponge $\mathcal{KS}$. The keyed sponge in turn calls the permutation f. The permutation f (or its inverse) receives N direct queries, in addition to those via $\mathcal{KS}$.

- The second game is the same but with the keyed sponge replaced by a random oracle, $(\mathcal{T}[\mathcal{RO}], f)$. Using Theorem 2, moving from the first game to this second game adds $\frac{M^2+4MN}{2^{c+1}}$ to the adversary's advantage.

- The third and last game is the same but with the tree hash mode being replaced by a random oracle and an injective mapping, $(\mathcal{G}[\mathcal{RO}], f)$. Assuming that $\mathcal{T}$ is sound, moving from the second game to this third game adds $\frac{(Pq)^2}{2^{c+1}}$ to the adversary's advantage, as proved in [8]. A query to the tree hash mode generates at least P nodes of at least one block each, so we can further bound $Pq \leq M$.

It remains to show that the mode $\mathcal{T}$ is sound. We can do so by using the three sufficient conditions of our paper [8], and we refer to it for the related concepts and terminology.

- Tree-decodability. First, there are no tree instances that are both compliant and final-subtree-compliant with $\mathcal{T}$ as the number of nodes (P) is coded in a byte at a fixed location in the final node. Then, the A_{decode} algorithm would work as follows. Starting from the final node, the value P can be decoded. Lemma 2 says that the complete session of the DuplexWrap object with index 0 (i.e., the final node) can be recovered.

- If $P = 1$, the tree has only one node, and A_{decode} shall return "compliant" if it is formatted as specified and "incompliant" otherwise.
- If $P > 1$, the last header contains the concatenation of tags from the other $P - 1$ DuplexWrap objects. These are chaining values in the language of [8]. The other parts can be identified as frame bits or message pointer bits, following the definition of KeyakLines. In the child nodes, there are no such chaining values and the frame and message pointer bits can similarly be identified. If the $P - 1$ child nodes are present and correctly formatted, A_{decode} shall return "compliant". Otherwise, returning an expanding index amounts to pointing to one of the missing nodes. In all cases, A_{decode} shall return "incompliant" if an incorrectly formatted node is encountered.

- Message completeness. All input bits are used after being cut by $\text{blocks}(\cdot, \rho, P)$. Given a compliant tree instance, one can recover the input headers and bodies, then the individual blocks as detailed in Lemma 2. Finally, the input strings can be recovered by following the definition of blocks^{-1} in Section 1.5.

- Final node separability. At a fixed position at the beginning of the input to the DuplexWrap objects, there is a byte $\text{enc}_8(i)$, which is $i = 0$ only for the final node.

6 Using Keyak in the context of CAESAR

In this section we explain how to use Keyak in the context of the CAESAR competition.

6.1 Specification and security goals

The specifications can be found in Section 5 and the security goals in Section 5.4.

6.2 Security analysis and design rationale

For the security analysis of Keyak and its building blocks we refer to the sections that explain the rationale behind them: Section 5.5 for Keyak, Section 4.3 for DuplexWrap, and Section 3.2 for the duplex construction.

As a generic property of sponge-based schemes, note that in a block cipher based scheme, the block length n puts a limit of about $2^{n/2}$ before collisions occur in the input blocks. In contrast, in sponge-based schemes, the capacity c takes the place of the block length in this limit. In Keyak, the capacity is $c = 252$.

Keyak has the following security assurance features:

- Generic security of the mode.

- Security assurance from cryptanalysis of Keccak. Note that thanks to the Matryoshka property, most analysis performed on Keccak-$f[1600]$ transfers to Keccak-$f[800]$.

The designers have not hidden any weaknesses in this cipher or any of its components. We believe this to be impossible. For Keccak-f and its round-reduced versions, all design choices are documented and explained in [6]. For the layers above, rationales are given in Sections 3.2, 4.3 and 5.5.

6.3 Features

We would like to highlight the following features of KEYAK, for which our proposal compares favorably to AES-GCM.

- As a functional feature not present in most authenticated ciphers, KEYAK supports sessions. In a session, sequences of messages can be authenticated rather than a single message. The session is initialized by loading the key and nonce and the tag for each message authenticates the complete sequence of messages preceding it. During the session, the communicating entities have to keep state.
- An important advantage of KEYAK is its hardware efficiency, with a better performance/cost trade-off compared to AES-GCM. It is based on the same primitive as that of SHA-3, therefore allowing to re-use resources when hashing is also needed.
- The round function can be easily protected against different types of side channel attacks.

6.4 Intellectual property

We did not submit any patents on KEYAK and do not intend to do so. If any of this information changes, the submitters will promptly (and within at most one month) announce these changes on the crypto-competitions mailing list.

6.5 Consent

The submitters hereby consent to all decisions of the CAESAR selection committee regarding the selection or non-selection of this submission as a second-round candidate, a third-round candidate, a finalist, a member of the final portfolio, or any other designation provided by the committee. The submitters understand that the committee will not comment on the algorithms, except that for each selected algorithm the committee will simply cite the previously published analyses that led to the selection of the algorithm. The submitters understand that the selection of some algorithms is not a negative comment regarding other algorithms, and that an excellent algorithm might fail to be selected simply because not enough analysis was available at the time of the committee decision. The submitters acknowledge that the committee decisions reflect the collective expert judgments of the committee members and are not subject to appeal. The submitters understand that if they disagree with published analyses then they are expected to promptly and publicly respond to those analyses, not to wait for subsequent committee decisions. The submitters understand that this statement is required as a condition of consideration of this submission by the CAESAR selection committee.

References

[1] M. Bellare and P. Rogaway, *The security of triple encryption and a framework for code-based game-playing proofs*, Advances in Cryptology – Eurocrypt 2006 (S. Vaudenay, ed.), Lecture Notes in Computer Science, vol. 4004, Springer, 2006, pp. 409–426.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, http://sponge.noekeon.org/, pp. 181–197.

[3] ———, *Cryptographic sponge functions*, January 2011, http://sponge.noekeon.org/.

[4] ———, *Duplexing the sponge: single-pass authenticated encryption and other applications*, Selected Areas in Cryptography (SAC), 2011.

[5] ———, *On the security of the keyed sponge construction*, Symmetric Key Encryption Workshop (SKEW), February 2011.

[6] ———, *The* KECCAK *reference*, January 2011, http://keccak.noekeon.org/.

[7] ———, *The* KECCAK *sponge function family*, 2013, http://keccak.noekeon.org/.

[8] ———, *Sufficient conditions for sound tree and sequential hashing modes*, International Journal of Information Security (2013), http://dx.doi.org/10.1007/s10207-013-0220-y.

[9] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, KECCAK *implementation overview*, May 2012, http://keccak.noekeon.org/.

[10] C. Boura, A. Canteaut, and C. De Cannière, *Higher-order differential properties of Keccak and Luffa*, Fast Software Encryption 2011, 2011.

[11] M. Duan and X. Lai, *Improved zero-sum distinguisher for full round Keccak-f permutation*, Cryptology ePrint Archive, Report 2011/023, 2011, http://eprint.iacr.org/.

[12] V. Shoup, *Sequences of games: a tool for taming complexity in security proofs*, IACR Cryptology ePrint Archive **2004** (2004), 332.

A Change log

A.1 From 1.0 to 1.1

Only Section 6.3 (“Features”) changed to include a brief comparison with AES-GCM.