CAESAR submission: KEYAK v2

Designed and submitted by:

Guido BERTONI[1]
Joan DAEMEN[1,2]
Michaël PEETERS[1]
Gilles VAN ASSCHE[1]
Ronny VAN KEER[1]

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

Version of KEYAK: 2
Version of the document: **2.2**
September 15, 2016

[1]STMicroelectronics
[2]Radboud University Nijmegen

Contents

This document specifies Keyak v2, a parameterized permutation-based authenticated encryption scheme with support for associated data and sessions. Its underlying permutation is Keccak-p and it is based on the Motorist mode for authenticated encryption. For Keyak v2, we formulate a generic definition and have 5 named instances. In the remainder of this document we denote Keyak v2 simply as Keyak. The named Keyak instances 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.

The remainder of this document is structured as follows. In Section 1 we specify Motorist and provide a motivation for introducing it. In Section 2 we specify Keyak, its components, named instances and security claim. In Section 3 we treat the provable generic security of Motorist, its implications for Keyak and discuss the state-of-the-art of cryptanalysis of Keyak. We explain how Keyak addresses the CAESAR call for proposals in Section 4. Finally, Appendix A contains a change log.

1 Definition of the Motorist authenticated encryption mode

The mode Motorist supports the authenticated encryption of sequences of messages in *sessions*. During a session, it processes messages and cryptograms. A message consists of a plaintext and possible associated data (called *metadata* in the remainder of this document). For each message, it *wraps* it by enciphering the plaintext into a ciphertext and computing a tag over the full sequence of messages. A cryptogram consists of a ciphertext, possible metadata and a tag. For each cryptogram, it *unwraps* it by deciphering the ciphertext into a plaintext, verifying the tag, and returning the plaintext if the tag is valid. A message can also consist of metadata alone and the corresponding cryptogram does not have any ciphertext. Within a session, the tag of a cryptogram authenticates the full sequence of messages sent/received since the start of the session. The start of a session requires a secret key and possibly a nonce, if the secret key is not unique for this session.

The mode Motorist is sponge-based and supports one or more duplex instances operating in parallel. It makes duplexing calls with input containing key, nonce, plaintext and metadata bits and uses its output as tag or as key stream bits.

The duplex instances in Motorist differ from the original duplex construction [3] in that they accept input blocks as large as the width of the permutation (after padding), instead of only the outer part. This variant, initialized with a secret key and denoted *full-state keyed duplex* (FSKD), was introduced by Mennink, Reyhanitabar and Vizár [14]. They proved a strong result on the generic security of the FSKD. More precisely they give an upper bound on the advantage of distinguishing a FSKD calling a random permutation from a random oracle, that is quite close to that of the original keyed duplex construction [1]. This means that increasing the input block length from the rate (r bits) to the width of the permutation (b bits) has no noticeable impact on the generic security, while allowing the injection of more bits per call to the underlying permutation, thus improving performance.

The mode Motorist supports a parameterized degree of parallelism. This allows exploiting resources such as single-instruction multiple-data (SIMD) instructions in modern CPUs or pipelining in dedicated hardware. The Motorist distributes the message (plaintext and metadata) over the different duplex instances, where each input bit is absorbed in a single duplex instance. To produce a tag that depends on the full message and not only on the message bits that have been injected in a single duplex instance, Motorist performs some dedicated processing at the end of each message called a *knot*. It extracts

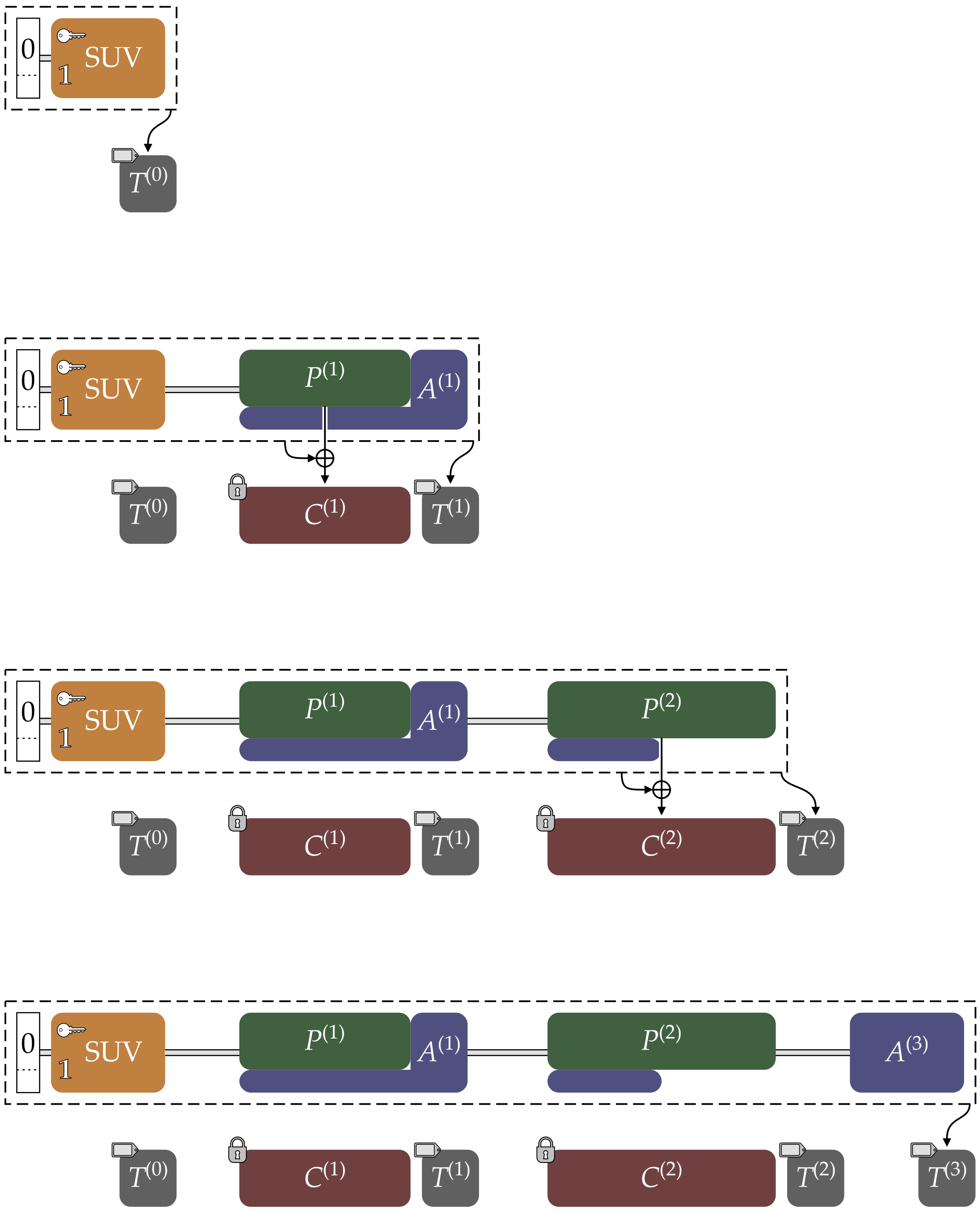

Figure 1 – *A session in Motorist. First, the session is started with a given secret and unique value* (SUV)*. Optionally, a tag $T^{(0)}$ on* SUV *can be produced or verified. Then, Motorist processes both the plaintext $P^{(1)}$ and metadata $A^{(1)}$ in parallel. The plaintext $P^{(1)}$ is encrypted into ciphertext $C^{(1)}$ and $T^{(1)}$ authenticates* $(\text{SUV}, P^{(1)}, A^{(1)})$*. After processing the second message, $T^{(2)}$ authenticates* $(\text{SUV}, P^{(1)}, A^{(1)}, P^{(2)}, A^{(2)})$*, and after the third message, $T^{(3)}$ authenticates the full session* $(\text{SUV}, P^{(1)}, A^{(1)}, P^{(2)}, A^{(2)}, P^{(3)}, A^{(3)})$*, where $P^{(3)}$ is the empty string.*

chaining values from each duplex instance, concatenates them, and injects them into all duplex instances. This makes the state of all duplex instances depend on the full sequence of messages. Then it extracts a tag from a single duplex object.

To start a session, Motorist takes as input a string that must be secret and (globally) unique. We call this string the *secret and unique value* (SUV). If the SUV consists of a key and a nonce, we recommend the key comes first. Motorist injects the SUV into each duplex instance, appending a diversification string at the end to make their states different. Figure 1 illustrates a session in Motorist.

A single Motorist session can be used to secure two-way communication between two parties. In that case, one must clearly indicate for each message who is its sender. This can be done by including its identifier in the metadata of the message. Alternatively, one can rely on a strict convention, such as messages alternating in the two directions. In the case of a session that is dedicated to unwrapping only, the Motorist session being started does not have to impose the nonce requirement to the SUV.

1.1 Motivation for the introduction of the Motorist mode

From a bird's eye perspective, Motorist offers the same functionality as the modes underlying Keyak v1 and still builds on the security of the sponge construction, although rather a variant. Still, in the transition from Keyak v1 to Keyak v2, the modes have been significantly refactored. In this section we explain the reasons behind the change and its benefits.

The main reasons to migrate to a new permutation-based authenticated encryption mode taking the place of DuplexWrap and KeyakLines are the following:

Reducing computational cost for short messages Per message that contains plaintext, DuplexWrap makes at least two calls to the permutation f: one call for absorbing the (possibly empty) metadata and producing the key stream, and one call for absorbing the plaintext and producing the tag. By supporting output blocks to be used partially as tag and partially as key stream and supporting the combination of metadata and plaintext in a single input block, this can be reduced to one call to f.

Reducing computational cost for long messages After the publication of [14] we realized that increasing the length of input blocks from r to b bits (after padding) has no impact on the generic security bounds that can be proven for the keyed sponge and duplex construction. This allows absorbing up to $c = b - r$ additional bits per call to f.

Once the decision was taken to have a new mode, we decided that the following features of DuplexWrap and KeyakLines should be preserved:

In-place encryption and decryption In DuplexWrap, state bits before absorbing a block of plaintext correspond to key stream bits, and they become ciphertext bits afterwards. So the encryption/decryption operation coincides with the absorbing operation. This means that no buffer is necessary and plaintext or ciphertext bits can be processed as they arrive. To preserve this feature, this implies that the plaintext fragment is limited to the outer part of the input blocks.

Sessions During a session, a tag of a cryptogram authenticates the full sequence of messages since the start of the session and only a single nonce (if any) is required per session.

Authentication-only The mode supports the (efficient) generation of tags over messages consisting of metadata only.

Stream-compatible For its operation the mode does not require prior knowledge of the length of plaintext, ciphertext or metadata.

Word-alignment The mode can be instantiated such that it processes data in 64-bit or 32-bit units, without the need for additional bit- or byte-shuffling.

Universal The mode can be applied to any fixed-length permutation with sufficient width.

Additionally, we took into account the following requirements, which were not satisfied by DuplexWrap and KeyakLines:

Uniformity The specification of the mode should cover at the same time serial and parallel instances.

Synchronicity The parallel instances should run synchronously, with the calls to f appearing systematically at the same time on all instances, and with input blocks containing the same types of fragments.

As a result of the new design, two new features appeared:

Tag on session setup The setup of a session can return a tag, or can be subject to a tag. So when two communicating entities both start a Motorist session, one of them can send the tag (and if required the nonce) to the other one that can then set up the same session on the condition that the tag it receives is valid (for the common nonce). The benefit is that no unwrapping process can start unless a legitimate session is started.

Integrated forgetting The mechanism that Motorist uses for making the tag depend on the state of all duplex instances has as side effect that knowledge of the full state does not allow the reconstruction of the state prior to the wrapping (unwrapping) of the current message (cryptogram). We call this feature *forgetting*. It is also supported in the setup of a session and hence a key that is loaded during session setup cannot be recovered from the state. For serial instances, this feature can be switched off for increasing performance.

After the design, two important changes became apparent:

Metadata absorbing during and after plaintext While in DuplexWrap the metadata of a message is in the input blocks strictly before those with the plaintext, in Motorist metadata is input together with the plaintext and possibly in input blocks after it.

Length-coding instead of trailing frame bits and multi-rate padding For domain separation and decodability, Motorist makes use of length coding with a number of integers present in each input block delimiting messages and indicating where in the input blocks the plaintext and metadata fragments are. We call these the *fragment offsets*. In DuplexWrap it was important to reduce the overhead of frame and padding bits to a minimum as they reduce the usable rate. In Motorist this is less critical as these integers are in the inner part of the input blocks.

1.2 The layered structure

We specify Motorist in three layers, each handling a different aspect. The input and output strings processed in these layers are described in terms of *byte streams*, i.e., a string of bytes that can be read from and/or written to sequentially. Using streams instead of traditional strings brings the specification closer to the implementation, where, e.g., the input data is processed as it arrives and its length is not necessarily known in advance. We call a sequence of consecutive bytes from a stream a *fragment*.

The layers are, from bottom to top:

Piston This layer keeps a b-bit state and applies the permutation f to it. It performs the basic functions such as injecting data, possible simultaneous encryption or decryption, extracting tags and setting the fragment offsets. It has a *squeezing rate*, the classical sponge rate, and an *absorbing rate*, the state width minus the last part containing the fragment offsets. When being called to inject, it receives a reference to a byte stream and it puts a fragment that is as long as the input block can hold or that exhausts the input byte stream, and sets the corresponding fragment offsets to the correct value. When being called to encrypt or decrypt, it puts a plaintext fragment that covers the remaining outer part of the input block or that exhausts the input byte stream, and sets the corresponding fragment offset.

Engine This layer controls $\Pi \geq 1$ Piston objects that operate in parallel. It serves as a dispatcher keeping its Piston objects busy, imposing that they are all treating the same kind of request. It can also inject the same stream into all Piston objects collectively. The Engine also ensures that the SUV and message sequence can be reconstructed from the sponge input to each Piston object and that each output bit of its Piston objects is used at most once.

Motorist This layer implements the user interface. It supports the starting of a session and subsequent wrapping of messages and unwrapping of cryptograms by driving the Engine.

1.3 Conventions

Before we describe the three layers in details, we define the conventions we use.

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

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

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.

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`. 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`.

The function $\mathrm{enc}_8(x)$ encodes the integer x, with $0 \leq x \leq 255$, as a byte with value x.

In our specification we make use of *byte streams*. In actual implementations, they can take the form of pointers to some buffer, bytes arriving from, or sent to, some communication channel, and so on. What is important is that a realization supports the set of functions defined here. We indicate byte streams by capital letters such as X and denote operations using the convention $X.\textsc{DoSomething}()$, popular in object oriented programming. Concretely, a byte stream is a string of bytes that supports the following functions, similarly to a queue:

- $z \leftarrow X.\textsc{PullByte}()$ removes the first byte of stream X and assigns it to z;
- $X.\textsc{PushByte}(z)$ appends byte z to the end of the stream X;
- $X.\textsc{HasMore}$ returns a Boolean value that indicates whether stream is empty (False) or not (True);
- $(X = Y)$ returns a Boolean value that is True iff streams X and Y have the same content;
- $X.\textsc{Clear}()$: removes all bytes from stream X.

At some places we speak of input byte streams and output byte streams. An input byte stream does not have to support $\textsc{PushByte}(z)$ and an output byte stream does not have to support $\textsc{PullByte}()$.

In the specification of Motorist we define a number of types (classes) of objects, each having a specific set of attributes and supporting a specific set of functions. When instantiating an object, the value of a number of parameters are determined and the attributes are initialized. Once an object is instantiated, it can be used by calling its functions. In between calls, the attributes of the object keep their values. We denote objects by a name, such as Piston and their functions (attributes) by the name followed by a dot and the name of the function (attribute), possibly with some arguments, such as $\text{Piston}.\textsc{Inject}(X)$. When a byte stream figures as the parameter in a function call, it should be seen as a *reference* to the byte stream being passed. The object supporting the called function can use this reference to perform operations on the byte stream.

1.4 The Piston

Piston is specified in Algorithm 1. It uses a permutation f operating on b-bit state denoted as s. During instantiation, the Piston state is initialized to all-zero. In the algorithm, we use $s[i]$ to denote byte i of the state s, where indexing starts from 0. The other parameters of Piston are the squeezing byte rate R_s and the absorbing byte rate R_a with $R_s \leq R_a$.

At Piston-level there is no distinction between metadata, SUV and chaining values and we will use the term metadata to cover all three. When properly used (i.e., through an Engine), the Piston builds a full-width input block from plaintext, metadata and encoding of fragments offsets, formatted as follows:

- possibly a number of zero bytes, starting at index 0;
- possibly a plaintext fragment, starting after the zero bytes, and finishing at most at index R_s;
- possibly a metadata fragment, starting at index 0 (if no plaintext fragment) or at index R_s (otherwise), and finishing at most at index R_a;
- the fragment offsets.

Algorithm 1 Definition of $\textsc{Piston}[f, R_s, R_a]$

```
Require: $R_s$ is the squeezing rate in bytes
Require: $R_a$ is the absorbing rate in bytes, with $R_s \leq R_a \leq \frac{b-32}{8} < 248$
Convention: $I, X$ input and $O, T$ output byte streams

Instantiation: Piston $\leftarrow \textsc{Piston}[f, R_s, R_a]$
  State: $s \leftarrow 0^b$
  Offsets: $(\text{EOM}, \text{Crypt End}, \text{Inject Start}, \text{Inject End}) \leftarrow (R_a, R_a + 1, R_a + 2, R_a + 3)$
  Crypt and Inject offsets: $(\omega_C, \omega_I) \leftarrow (0, 0)$

Interface: Piston.$\textsc{Crypt}(I, O, \textsc{decryptFlag})$
  while $(I.\textsc{HasMore} = \textsc{True})$ AND $(\omega_C < R_s)$ do
    $x \leftarrow I.\textsc{PullByte}()$
    $O.\textsc{PushByte}(s[\omega_C] \oplus x)$
    if $\textsc{decryptFlag} = \textsc{True}$ then
      $s[\omega_C] \leftarrow x$
    else
      $s[\omega_C] \leftarrow s[\omega_C] \oplus x$
    $\omega_C \leftarrow \omega_C + 1$
  $s[\text{Crypt End}] \leftarrow s[\text{Crypt End}] \oplus \text{enc}_8(\omega_C)$
  $(\omega_C, \omega_I) \leftarrow (0, R_s)$

Interface: Piston.$\textsc{Inject}(X)$
  $s[\text{Inject Start}] \leftarrow s[\text{Inject Start}] \oplus \text{enc}_8(\omega_I)$
  while $(X.\textsc{HasMore} = \textsc{True})$ AND $(\omega_I < R_a)$ do
    $s[\omega_I] \leftarrow s[\omega_I] \oplus X.\textsc{PullByte}()$
    $\omega_I \leftarrow \omega_I + 1$
  $s[\text{Inject End}] \leftarrow s[\text{Inject End}] \oplus \text{enc}_8(\omega_I)$
  $(\omega_C, \omega_I) \leftarrow (0, 0)$

Interface: Piston.$\textsc{Spark}()$
  $s \leftarrow f(s)$

Interface: Piston.$\textsc{GetTag}(T, \ell)$ with $\ell \leq R_s$
  if $\ell = 0$ then
    $s[\text{EOM}] \leftarrow s[\text{EOM}] \oplus \text{enc}_8(255)$
  else
    $s[\text{EOM}] \leftarrow s[\text{EOM}] \oplus \text{enc}_8(\ell)$
  Piston.$\textsc{Spark}()$
  for $i \leftarrow 0$ to $\ell - 1$ do $T.\textsc{PushByte}(s[i])$
  $\omega_C \leftarrow \ell$
```

Piston remembers in the offset attribute ω_C how many output bytes were used as tag or chaining value to avoid reusing the same bits as key stream during plaintext encryption or decryption. It also remembers whether the current block contains a plaintext fragment and stores in offset attribute ω_I the position where metadata bits must be absorbed.

After the application of f, the bytes of the outer part of the state are used as follows:

- possibly a number of bytes used as tag, starting at index 0;
- possibly a number of bytes used as key stream, starting after the possible tag.

There are four fragment offsets:

EOM This fragment offset has a double function. First, it codes the number of bytes in the next output block that are used as tag, and that will consequently not be used as key stream. Second, it delimits messages by having a non-zero value if it is part of an input block that is the last of a message or of a string that is injected collectively. In case no tag is requested at the end of message or string that is injected collectively, EOM takes the value 255. The values 248 and above have a special meaning and are reserved for future use.

Crypt End This codes the end of the plaintext fragment in the current input block. (The start of the plaintext fragment is coded by EOM in the previous input block, where the value 255 means that the plaintext fragment starts at index 0.)

Inject Start This codes the start of the metadata fragment in the current input block. If there is also a plaintext fragment in the current input block, then the metadata fragment starts at Inject Start $= R_s$. Otherwise, the metadata fragment starts at Inject Start $= 0$.

Inject End This codes the end of the metadata fragment in the current input block.

In the algorithm, the attributes EOM, Crypt End, Inject Start and Inject End are the indexes where the fragment offsets are coded.

The function Piston.$\textsc{Crypt}(I, O, \textsc{decryptFlag})$ supports the combined encryption of plaintext (or decryption of ciphertext) and absorbing of the corresponding plaintext into the outer part of the state. The Boolean $\textsc{decryptFlag}$ indicates whether it is encryption ($\textsc{False}$) or decryption ($\textsc{True}$). Here I denotes the input byte stream containing the plaintext to be encrypted or ciphertext to be decrypted and O the output byte stream where the result will be written to. The plaintext absorbtion starts at index given by Piston offset ω_C and will end at index R_s or earlier if the input stream is exhausted. It codes the end of the plaintext fragment in the offset Crypt End, resets the plaintext absorbtion index ω_C and sets offset ω_I to indicate the presence of a plaintext fragment.

The function Piston.$\textsc{Inject}(X)$ injects metadata taken from the input stream X. Piston starts injecting from index ω_I. The metadata fragment will end at index R_a or earlier if the input stream is exhausted. It codes the start of the metadata fragment in the offset Inject Start and its end in Inject End, and finally resets both offsets (ω_C, ω_I).

The function Piston.$\textsc{Spark}()$ simply applies the underlying permutation f to the state. It is called by the parent Engine when there are still plaintext or metadata bytes waiting to be absorbed in the current session.

Finally, the function Piston.$\textsc{GetTag}(T, \ell)$ also applies the underlying permutation f to the state, but in addition writes the first ℓ bytes of the state to output byte stream T, to be used as a tag or chaining value. Before it does that, it codes in the data element EOM the number ℓ of bytes of the state after the application of f that are reserved as tag, or 255 if no tag was requested. In both case, this non-zero value indicates that the last input block of a message was absorbed.

The description of Piston assumes that the plaintext and metadata input streams do not refill between inject and crypt calls. More exactly, if the function X.HasMore returns False for an input stream X, it must keep doing so for that stream until next call to Piston.GetTag(). This also means that if an input block contains a plaintext fragment, this must be announced before injecting metadata.

As long as this constraint is respected, one could implement Piston differently such that it allows more freedom in the order that the plaintext and metadata are absorbed. These may be offered in short chunks and even in an alternating fashion.

1.5 The Engine

Engine is specified in Algorithm 2. It controls and relies on an array of Π Piston objects that operate in parallel. Engine does not maintain any state in itself. It relies on each Piston for maintaining the bit state and offsets, and on Motorist for the consistency of operation sequence.

Algorithm 2 Definition of Engine[Pistons]

```
Require: Pistons is an array of $\Pi$ pistons, with $1 \leq \Pi \leq 255$
Convention: $I, A, X$ input and $O, T$ output byte streams

Instantiation: Engine ← Engine[Pistons]

Interface: Engine.Wrap($I, O, A$, decryptFlag)
  if ($I$.HasMore = True) then
    for $i \leftarrow 0$ to $\Pi - 1$ do Pistons[$i$].Crypt($I, O$, decryptFlag)
  for $i \leftarrow 0$ to $\Pi - 1$ do Pistons[$i$].Inject($A$)
  if ($I$.HasMore = True) OR ($A$.HasMore = True) then
    for $i \leftarrow 0$ to $\Pi - 1$ do Pistons[$i$].Spark()

Interface: Engine.GetTags($T, \ell$) with $\ell \in \mathbb{N}^{\Pi}$
  for $i \leftarrow 0$ to $\Pi - 1$ do Pistons[$i$].GetTag($T, \ell[i]$)

Interface: Engine.InjectCollective($X$, diversifyFlag)
  Let $Y$ be an array of $\Pi$ local byte streams, initially empty
  while $X$.HasMore = True do
    $x \leftarrow X$.PullByte()
    for $i \leftarrow 0$ to $\Pi - 1$ do $Y[i]$.PushByte($x$)
  if diversifyFlag = True then
    for $i \leftarrow 0$ to $\Pi - 1$ do $Y[i]$.PushByte(enc$_8$($\Pi$))
    for $i \leftarrow 0$ to $\Pi - 1$ do $Y[i]$.PushByte(enc$_8$($i$))
  while $Y[0]$.HasMore = True do
    for $i \leftarrow 0$ to $\Pi - 1$ do Pistons[$i$].Inject($Y[i]$)
    if $Y[0]$.HasMore = True then
      for $i \leftarrow 0$ to $\Pi - 1$ do Pistons[$i$].Spark()
```

Engine has three interfaces. In each of them the processing terminates by the application of f on the Π states, either via call Piston.SPARK() or to Piston.GETTAG(). An actual implementation can either perform these calls sequentially or in parallel. This is also possible for the calls to Piston.CRYPT() and Piston.INJECT() interfaces.

If the input stream I is not exhausted, the function Engine.WRAP(I, O, A, DECRYPTFLAG) starts by dispatching the input to the Π Piston objects and collecting the corresponding Π output in O. Each Piston object takes a fragment from I, so all objects process in total up to ΠR_s bytes. The DECRYPTFLAG is as for Piston.CRYPT(). Next, Engine dispatches the metadata A to the Piston objects. Each Piston object takes a fragment from A, so all objects process in total up to $\Pi(R_a - R_s)$ bytes (if Piston.CRYPT() was called before) or ΠR_a bytes (otherwise). Note that Piston.INJECT() is called even if A is exhausted such that to set appropriately the offsets in the Π Piston states. Finally, Engine calls Piston.SPARK() for each Piston, unless both the input and the metadata streams are exhausted. In that case it delays the application of f until the call to Engine.GETTAGS().

The function Engine.GETTAGS(T, ℓ) calls Piston.SPARK() on all Π Piston objects and collects the corresponding tags into the output stream T. The last parameter, ℓ, is in fact a vector, allowing one to take a different number of bits in each Piston.

The function Engine.INJECTCOLLECTIVE(X, DIVERSIFYFLAG) aims at injecting the same metadata X to all Π Piston objects. It is used to inject the SUV and the chaining values. When DIVERSIFYFLAG = TRUE, as set when injecting the SUV, it appends to X two bytes:

1. one byte encoding the degree of parallelism Π, for domain separation between instances with a different number of Piston objects, and
2. one byte encoding the index of the Piston object, for domain separation between Piston objects and in particular to avoid identical key streams.

1.6 The Motorist

Motorist is specified in Algorithm 3. It uses an Engine object, calling a parameterized number Π of Piston objects. A Motorist object is also parameterized by the *alignment unit* W in bits, typically 32 or 64. This ensures that the fragment start offsets and the length of tags, chaining values and fragments (except when a stream is exhausted) are a multiple of W, allowing data to be manipulated in multi-byte chunks. The remaining parameters determine the security level: the *capacity* c and the *tag length* τ. From these, the Motorist object derives the following quantities:

- the squeezing byte rate R_s, the largest multiple of W such that at least $\max(c, 32)$ bits (for the inner part and for the fragment offsets) of the state are never used as output;
- the absorbing byte rate R_a, the largest multiple of W that reserves at least 32 bits at the end of the state for absorbing the fragment offsets;
- the chaining value length c', the smallest multiple of W greater than or equal to the capacity c.

Motorist maintains its own state machine via the attribute PHASE. The possible phases are:

ready The Motorist object is initialized and no input has been given yet.

riding The Motorist object processed the SUV and is able to (un)wrap. The object stays in this phase until an error occurs.

failed The Motorist object received an incorrect tag.

In order for a tag to depend on the state of the $\Pi > 1$ Piston objects, or when $\Pi = 1$ and forgetting is requested, the Motorist object performs an operation that we call a *knot*. This is the purpose of the Motorist.MakeKnot() function. This function first retrieves a c'-bit chaining values from each Piston object, concatenates these to make a $\Pi \times c'$-bit string and collectively injects it into all Piston objects. For $\Pi > 1$, this makes the state of all Piston objects depend on each other. A fortiori this is also the case for Pistons[0], from which the tag of a message is extracted.

For the chaining values we have a length of at least c bits so that the probability of collisions in the chaining values is not larger than that of collisions in the inner part of the state (see Section 3.2). In addition, the chaining value of Pistons[0] is injected exactly where it was extracted, resulting into setting c' bits of the outer part to zero. This chaining value is also injected in the remaining $\Pi - 1$ states. To compute backwards in any of the Piston objects, an adversary would then have to guess $c' \geq c$ bits, hence protecting the Π state(s) before the knot, if some leakage occurs after the knot. The knot is illustrated in Figure 2.

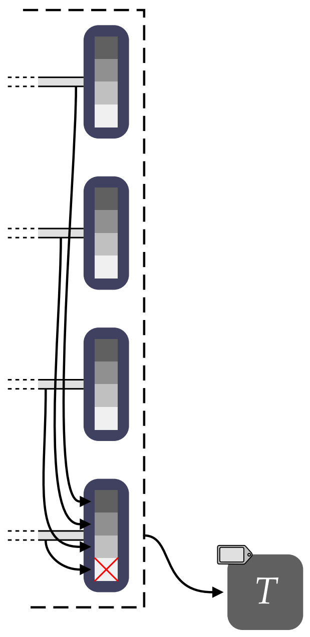

Figure 2 – *A knot. In this case, $\Pi = 4$. Each line represents the state of a piston, from* Pistons[0] *at the bottom to* Pistons[3] *at the top. Chaining values taken from all the pistons are injected collectively into the four pistons. The arrows show how the chaining values are injected in* Pistons[0]*, and the same values are injected symmetrically in the other pistons, but to avoid overloading the figure we did not draw the corresponding arrows. The c' first bits of* Pistons[0]*'s state are injected back, thereby setting them to zero, as symbolized by the red cross. If a tag is taken, it is taken from* Pistons[0]*, whose state now depends on all pistons.*

The function Motorist.StartEngine(SUV, tagFlag, T, decryptFlag, forgetFlag) begins a session with the given SUV read from the SUV byte stream. It collectively injects it, with diversifyFlag = True for domain separation as explained above. The parameter forgetFlag tells whether a knot is necessary. The starting of a session supports the generation or verification of a tag by setting the parameter tagFlag to True. If decryptFlag $=$ False, it returns a tag in the byte stream T and otherwise it verifies the tag read from T. Unless the tag verification fails, it switches the phase to *riding*.

The function Motorist.Wrap$(I, O, A, T,$ decryptFlag, forgetFlag$)$ wraps a message or unwraps a cryptogram.

- To wrap, the function must be called with decryptFlag $=$ False, I (resp. A) an input byte stream containing the plaintext (resp. the metadata), O (resp. T) an output byte stream ready to get the ciphertext (resp. the tag) and forgetFlag.
- To unwrap, the function must be called with decryptFlag $=$ True, I (resp. A, T) an input byte stream containing the ciphertext (resp. the metadata, the tag) and O an output byte stream ready to get the plaintext and forgetFlag. The function returns True if the tag is correct and False otherwise. In addition, it clears the byte stream O if the tag is incorrect.

The function starts by calling repeatedly Engine.Wrap() until both the input and the metadata streams are exhausted. Then, if forgetFlag $=$ True or $\Pi > 1$, the function calls Motorist.MakeKnot$()$. Finally, it generates or verifies the tag.

Once a session is started with Motorist.StartEngine$()$, the Motorist object can receive as many calls to Motorist.Wrap() as desired. The nonce requirement (i.e., that the SUV is unique) plays at the level of the session. Within a session, messages have no explicit message number or nonce. However, the communicating parties must process them in the same order for the tags to verify. An alternative way to see this concept of session is that it supports intermediate tags. This allows the two parties to communicate in both directions in a single session by setting appropriately decryptFlag in the calls to Motorist.Wrap(). Note that, as the state of the Piston objects depends on whether a tag is requested or not (when calling Motorist.StartEngine$()$) and whether a knot is performed or not, the communicating parties must use synchronized values for the tagFlag and forgetFlag parameters.

Algorithm 3 Definition of $\textsc{Motorist}[f, \Pi, W, c, \tau]$.

```
Require: Π is the number of pistons, with $1 \le \Pi \le 255$
Require: $W$ is the alignment unit in bits, with $W$ a strictly positive multiple of 8
Require: $c$ is the required capacity in bits, with $\lceil \frac{c}{W} \rceil \le \lfloor \frac{b-\max(c,32)}{W} \rfloor$
Require: $\tau$ is the tag length in bits, a multiple of $W$ and $\tau \le W \lceil \frac{c}{W} \rceil$

Instantiation: Motorist $\leftarrow$ $\textsc{Motorist}[f, \Pi, W, c, \tau]$
  Squeezing byte rate: $R_s \leftarrow \frac{W}{8} \lfloor \frac{b-\max(c,32)}{W} \rfloor$
  Absorbing byte rate: $R_a \leftarrow \frac{W}{8} \lfloor \frac{b-32}{W} \rfloor$
  Chaining value length: $c' \leftarrow W \lceil \frac{c}{W} \rceil$
  Engine: Engine $\leftarrow$ $\textsc{Engine}[\textsc{Piston}[f, R_s, R_a]^{\Pi}]$
  Phase: $\textsc{phase} \leftarrow$ $\textit{ready}$

Interface: $\textsc{res} \leftarrow$ Motorist.$\textsc{StartEngine}(SUV, \textsc{tagFlag}, T, \textsc{decryptFlag}, \textsc{forgetFlag})$
  if $\textsc{phase} \neq \textit{ready}$ then return error
  Engine.$\textsc{InjectCollective}(SUV, \textsc{True})$
  if $\textsc{forgetFlag} = \textsc{True}$ then Motorist.$\textsc{MakeKnot}()$
  $\textsc{phase} \leftarrow \textit{riding}$
  return Motorist.$\textsc{HandleTag}(\textsc{tagFlag}, T, \textsc{decryptFlag})$

Interface: $\textsc{res} \leftarrow$ Motorist.$\textsc{Wrap}(I, O, A, T, \textsc{decryptFlag}, \textsc{forgetFlag})$
  if $\textsc{phase} \neq \textit{riding}$ then return error
  repeat
    Engine.$\textsc{Wrap}(I, O, A, \textsc{decryptFlag})$
  until $(I.\textsc{HasMore} = \textsc{False})$ AND $(A.\textsc{HasMore} = \textsc{False})$
  if $(\Pi > 1)$ OR $(\textsc{forgetFlag} = \textsc{True})$ then Motorist.$\textsc{MakeKnot}()$
  $\textsc{res}$ = Motorist.$\textsc{HandleTag}(\textsc{True}, T, \textsc{decryptFlag})$
  if $\textsc{res} = \textsc{False}$ then $O$.$\textsc{Clear}()$
  return $\textsc{res}$

Internal interface: Motorist.$\textsc{MakeKnot}()$
  Let $T'$ be a local byte stream, initially empty
  Engine.$\textsc{GetTags}(T', [c'/8]^{\Pi})$
  Engine.$\textsc{InjectCollective}(T', \textsc{False})$

Internal interface: $\textsc{res} \leftarrow$ Motorist.$\textsc{HandleTag}(\textsc{tagFlag}, T, \textsc{decryptFlag})$
  Let $T'$ be a local byte stream, initially empty
  if $\textsc{tagFlag} = \textsc{False}$ then
    Engine.$\textsc{GetTags}(T', 0^{\Pi})$
  else
    Engine.$\textsc{GetTags}(T', [\tau/8, 0^{\Pi-1}])$
    if $\textsc{decryptFlag} = \textsc{False}$ then
      Copy $T'$ into $T$
    else if $T' \neq T$ then
      $\textsc{phase} \leftarrow \textit{failed}$
      return $\textsc{False}$
  return $\textsc{True}$
```

1.7 Illustrations

In this subsection, we illustrate the Motorist mode by showing the input block constructed by the mode and its underlying layers Engine and Piston. By "input block", we mean the sequence of bytes that are absorbed into the state between calls to the permutation f. Note that the input blocks are the same for wrapping and unwrapping.

We do not depict output blocks as they can be easily deduced:

- a tag is output by extracting the first $\tau/8$ bytes (16 bytes in the examples here) of the state after the last block is processed;
- key stream bytes used to encrypt (or decrypt) a given plaintext fragment are taken before the plaintext fragment is XORed, at the corresponding location in the state.

1.7.1 Conventions

The conventions we use in this subsection are illustrated in Figures 3 and 4. Figure 3 shows how we draw input blocks in the case of $\Pi = 1$. Two types of input blocks are illustrated: one containing both plaintext and metadata fragments, and another containing only a metadata fragment. The figure also gives the location of the fragment offsets. Figure 4 displays the convention used when $\Pi > 1$, where input blocks that can be simultaneously processed are "glued" together.

(plaintext fragment)	(metadata fr.)	EOM	CE	IS $= R_s$	IE
(metadata fragment)		EOM	CE	IS $= 0$	IE

Figure 3 – *Convention for displaying input blocks. Each input block is enclosed in a rectangle. Distinct blocks are separated by a small space. Within a block, we distinguish between the location containting the plaintext fragment (possibly empty), the one for the metadata fragment and the four fragment offsets. The fragment offsets Crypt End, Inject Start and Inject End are abbreviated into CE, IS and IE, respectively. Note that Inject Start can take only two values,* 0 *or* R_s, *depending on the presence or absence of a plaintext fragment.*

(piston #0's metadata fragment)	EOM	CE	IS	IE
(piston #1's metadata fragment)	EOM	CE	IS	IE
(piston #2's metadata fragment)	EOM	CE	IS	IE
(piston #3's metadata fragment)	EOM	CE	IS	IE

Figure 4 – *Convention for displaying input blocks when* $\Pi > 1$. *The convention is illustrated for* $\Pi = 4$ *as an example. The* Π *blocks that are processed together by the Engine have no space in between.*

1.7.2 Detailing Figure 1

We now illustrate what happens for the session depicted in Figure 1 with one call to Motorist.StartEngine() and then wrapping three messages $(A^{(1)}, P^{(1)})$, $(A^{(2)}, P^{(2)})$ and $(A^{(3)}, P^{(3)})$, with $P^{(3)}$ the empty string.

First, the Motorist object processes the secret and unique value SUV and produces a tag $T^{(0)}$. Figure 5 illustrates this for $\Pi = 1$ and assuming that SUV fits in one block, while Figure 6 illustrates the case $\Pi = 4$.

SUV 1 0 0^*	16	0	0	$\leq R_a$

Figure 5 – *Example of input block corresponding to the absorbing of* SUV *fitting in one block. The two bytes with value* 1 *and* 0 *that follow* SUV *encode* $\Pi = 1$ *and* $i = 0$*. Then a number of* 0 *bytes fill the rest of the metadata fragment. EOM* $= 16$ *as* 16 *bytes of tag are requested. There is no plaintext fragment, hence Crypt End* $=$ *Inject Start* $= 0$*. The value of Inject End is the length of* SUV *plus* 2*.*

SUV 4 0 0^*	16	0	0	$\leq R_a$
SUV 4 1 0^*	255	0	0	$\leq R_a$
SUV 4 2 0^*	255	0	0	$\leq R_a$
SUV 4 3 0^*	255	0	0	$\leq R_a$

Figure 6 – *Same as Figure 5 but with* $\Pi = 4$*. Notice that the* 16*-byte tag is taken only from the first piston (EOM* $= 16$*) and not from the others (EOM* $= 255$*).*

Then, the Motorist object receives the first message $(A^{(1)}, P^{(1)})$, where we assume that $\frac{|A^{(1)}|}{R_a - R_s} > \frac{|P^{(1)}|}{R_s}$, so that the plaintext is exhausted before the metadata is, as suggested on Figure 1. Figure 7 illustrates this case for $\Pi = 1$.

Note that if no tag was requested upon calling Motorist.StartEngine(), we would see EOM $= 255$ on all pistons in Figures 5 and 6, and the first plaintext fragment would be P_0 with $|P_0| = R_s$ (instead of $0^{16}||P_0$). See also Figure 10.

0^{16} P_0	A_0	0	R_s	R_s	R_a
P_1	A_1	0	R_s	R_s	R_a
		$\ldots$			
$P_\Diamond$ 0^*	A_x	0	$\leq R_s$	R_s	R_a
A_{x+1}		0	0	0	R_a
		$\ldots$			
$A_\triangle$ 0^*		16	0	0	$\leq R_a$

Figure 7 – *Input blocks for processing* $(A^{(1)}, P^{(1)})$*. We assume that* $A^{(1)} = A_0||\ldots||A_x||A_{x+1}||\ldots||A_\triangle$*, with* $|A_i| = R_a - R_s$ *for* $i \leq x$*,* $|A_i| = R_a$ *for* $x < i \neq \triangle$ *and* $|A_\triangle| \leq R_a$*. Similarly, we assume that* $P^{(1)} = P_0||\ldots||P_\Diamond$*, with* $|P_0| = R_s - 16$*,* $|P_i| = R_s$ *for* $0 < i \neq \Diamond$ *and* $|P_\Diamond| \leq R_s$*.*

Next, the Motorist object receives the second message $(A^{(2)}, P^{(2)})$, where we assume that $\frac{|A^{(2)}|}{R_a - R_s} < \frac{|P^{(2)}|}{R_s}$. This is somehow the opposite case as the first message, because now the metadata is exhausted first, again in line with what Figure 1 suggests. Figure 8 illustrates this case for $\Pi = 1$.

Finally, the last message that the Motorist object receives is $(A^{(3)},)$, containing only metadata. Figure 9 illustrates this case for $\Pi = 1$. Notice that the first block does not start

0^{16} P_0	A_0	0	R_s	R_s	R_a
P_1	A_1	0	R_s	R_s	R_a
	…				
P_x	$A_\triangle$ 0^*	0	R_s	R_s	$\leq R_a$
P_{x+1}	0^*	0	R_s	R_s	R_s
	…				
$P_\Diamond$ 0^*	0^*	16	$\leq R_s$	R_s	R_s

Figure 8 – *Input blocks for processing $(A^{(2)}, P^{(2)})$. We assume that $A^{(2)} = A_0||\dots||A_\triangle$, with $|A_i| = R_a - R_s$ for $i \neq \triangle$ and $|A_\triangle| \leq R_a - R_s$. Similarly, we assume that $P^{(2)} = P_0||\dots||P_\Diamond$, with $|P_0| = R_s - 16$, $|P_i| = R_s$ for $0 < i \neq \Diamond$ and $|P_\Diamond| \leq R_s$.*

with 0^{16}, even if a tag was requested for the previous message, since metadata require no key stream output.

A_0	0	0	0	R_a
A_1	0	0	0	R_a
	…			
$A_\triangle$ 0^*	16	0	0	$\leq R_a$

Figure 9 – *Input blocks for processing $(A^{(3)}, P^{(3)})$ with $P^{(3)}$ the empty string. We assume that $A^{(3)} = A_0||\dots||A_\triangle$, with $|A_i| = R_a$ for $i \neq \triangle$ and $|A_\triangle| \leq R_a$.*

1.7.3 Session of short messages

Figure 10 illustrates a session with short messages. When the plaintext fits in the outer part and the metadata in the inner part, the user can encrypt and get a tag in just one call to the permutation per message.

SUV 1 0 0^*		255	0	0	$\leq R_a$
$P^{(1)}$ 0^*	$A^{(1)}$ 0^*	16	$\leq R_s$	R_s	$\leq R_a$
0^{16} $P^{(2)}$ 0^*	$A^{(2)}$ 0^*	16	$\leq R_s$	R_s	$\leq R_a$
0^{16} $P^{(3)}$ 0^*	$A^{(3)}$ 0^*	16	$\leq R_s$	R_s	$\leq R_a$
	…				

Figure 10 – *A session with short messages. Here, we assume that $|P^{(1)}| \leq R_s$, $|P^{(i)}| \leq R_s - 16$ for $i > 1$, and $|A^{(i)}| \leq R_a - R_s$ for all i.*

1.7.4 Parallelized message and knot

As a last illustration, we display the processing of a message in a parameterized instance, including a knot. Figure 11 gives the input blocks when $\Pi = 4$ for a message (A, P) that can be processed in 2Π calls to the permutation before the knot. Notice that all pistons always have the same value for Inject Start. Hence, even if A_6 does not have a plaintext counterpart, we have Inject Start $= R_s$ since other pistons process some plaintext.

0^{16} P_0	A_0	0	R_s	R_s	R_a
P_1	A_1	0	R_s	R_s	R_a
P_2	A_2	0	R_s	R_s	R_a
P_3	A_3	0	R_s	R_s	R_a
P_4	A_4	32	R_s	R_s	R_a
P_5 0^*	A_5	32	$\leq R_s$	R_s	R_a
0^*	A_6 0^*	32	0	R_s	$\leq R_a$
0^*	0^*	32	0	R_s	R_s
$T'_0T'_1T'_2T'_3$ 0^*		16	0	0	128
$T'_0T'_1T'_2T'_3$ 0^*		255	0	0	128
$T'_0T'_1T'_2T'_3$ 0^*		255	0	0	128
$T'_0T'_1T'_2T'_3$ 0^*		255	0	0	128

Figure 11 – *Input blocks for processing a message* (A, P) *when* $\Pi = 4$. *In this figure, we assume that* $P = P_0|| \ldots ||P_5$, *with* $|P_0| = R_s - 16$, $|P_i| = R_s$ *for* $1 \leq i \leq 4$ *and* $|P_5| \leq R_s$. *Similarly, we assume that* $A = A_0|| \ldots ||A_6$, *with* $|A_i| = R_a - R_s$ *for* $0 \leq i \leq 5$ *and* $|A_6| \leq R_a - R_s$. *The chaining values are assumed to be* 32*-byte long, and therefore we see that EOM* $= 32$ *after absorbing the last blocks of message. Together, the chaining values make up a* 128*-byte string* $T'_0||T'_1||T'_2||T'_3$.

2 Definition of Keyak

In this section we provide a definition of the parameterized Keyak authenticated encryption scheme, its five named instances parameters fixed and the underlying permutations and specify the security goals.

2.1 The Keccak-p permutations

The Keccak-p permutations are derived from the Keccak-f permutations [4] 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-$p[b, n_\mathrm{r}]$ consists in the application of the *last* n_r rounds of Keccak-$f[b]$. When $n_\mathrm{r} = 12 + 2\ell$, Keccak-$p[b, n_\mathrm{r}]$ = Keccak-$f[b]$.

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

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

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

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

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

and all other values of RC$[i_\mathrm{r}][x, y, z]$ are zero. The values rc$[t] \in$ 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.

2.2 The key pack

We encode the key in what we call a *key pack*. Its purpose is to have a uniform way of encoding a secret key as prefix of an SUV.

The key pack makes use of *simple padding* denoted $\text{pad}10^*[r](|M|)$. This padding rule 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 $\text{pad}10^*[r](|M|)$ returns the byte string `0x01` $\texttt{0x00}^{(q-7)/8}$.

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

$$\text{keypack}(K, \ell) = \text{enc}_8(\ell)||K||\text{pad}10^*[8\ell - 8](|K|),$$

where the length of the key K is limited to $8(\ell - 1) - 1$ bits and with $\ell < 256$. That is, the key pack consists of

- a first byte indicating the full length of the key pack 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

$$\text{keypack}(K, 18) = \texttt{0x12 0x01 0x23 0x45 0x67 0x89 0xAB 0xCD 0xEF 0x01 0x00}^8.$$

2.3 Generic definition of Keyak

Keyak makes use of $\text{Motorist}[f, \Pi, W, c, \tau]$, with f an instance of Keccak-p. We have:

$$\text{Keyak}[b, n_\text{r}, \Pi, c, \tau] = \text{Motorist}[f, \Pi, W, c, \tau],$$

with $f = \text{Keccak-}p[b, n_\text{r}]$ and $W = \max(\frac{b}{25}, 8)$.

The SUV consists of $\text{keypack}(K, \ell_\text{k})||N$ with $\ell_\text{k} = \frac{W}{8}\left\lceil \frac{c+9}{W} \right\rceil$ and $N \in \mathbb{Z}_2^*$ with no limitation on its length.

2.4 Named instances of Keyak

We have five named instances of Keyak, taking on specific parameter values in the available range. For all five instances, we have $n_\text{r} = 12$, $c = 256$ and $\tau = 128$. In order of increasing state sizes, the instances are:

Name	b	Π	Main use case	2nd use case
River Keyak	800	1	defense-in-depth	lightweight
Lake Keyak	1600	1	defense-in-depth	high performance
Sea Keyak	1600	2	defense-in-depth	high performance
Ocean Keyak	1600	4	defense-in-depth	high performance
Lunar Keyak	1600	8	defense-in-depth	high performance

Lake Keyak is the primary recommendation. For River Keyak, $W = 32$ and the length of the key pack ℓ_k is 36 bytes. For the other instances, $W = 64$ and $\ell_\text{k} = 40$ bytes.

All these instances take a variable-length public message number (or nonce) N, but no private message number. If the data element N has to have a fixed length, we propose that it takes 58 bytes for River Keyak and 150 bytes for the other instances. Note that our security claim covers any length of N. These lengths are chosen so that $\text{keypack}(K, \ell_\text{k})||N$ and the two bytes of diversification all fit in exactly one block.

	KEYAK
plaintext confidentiality	$\min(c/2, \|K\|)$
plaintext integrity	$\min(c/2, \|K\|, \|T\|)$
associated data integrity	$\min(c/2, \|K\|, \|T\|)$
public message number integrity	$\min(c/2, \|K\|, \|T\|)$

Table 1 – *Claimed security strength for KEYAK*

All KEYAK instances 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 at least 256 bits (determined by ℓ_k), as a possible countermeasure against multi-target attacks.

LAKE KEYAK can absorb up to 192 bytes of metadata per call to f or up to 168 of plaintext, with additionally 24 bytes of metadata. For SEA, OCEAN and LUNAR KEYAK, these sizes are multiplied by Π for every Π parallel calls to f. RIVER KEYAK may be of interest for its smaller state size. It can absorb up to 96 bytes of metadata per call to f or up to 68 of plaintext, with additionally 28 bytes of metadata.

The KEYAK instances with $\Pi > 1$ 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 [6]. SEA KEYAK best exploits 128-bit SIMD, while OCEAN KEYAK best exploits 256-bit SIMD and LUNAR KEYAK 512-bit SIMD.

2.5 Security goals

Before stating our security goals, we define some terminology related to attacks and resistance against them. Attacks against keyed cryptographic schemes make use of two types of resources:

Data complexity The total amount of data processed by the keyed cryptographic scheme. This is sometimes also called the online complexity. For sponge-based crypto we quantify it by M: the number of evaluations of the permutation f by the keyed cryptographic scheme under attack.

Computational complexity The total computational effort of the attack. this is sometimes also called the offline complexity. For sponge-based crypto we quantify it by N: the computation where the evaluation of the underlying permutation f is considered as the unit. In generic attacks N corresponds to the number of evaluations of f or f^{-1}.

Although data and computational complexity are very different, they are counted using the same unit and we call their sum $M + N$ the *total complexity of an attack*.

Definition 1 (security strength)**.** *We say a cryptographic scheme has security strength s if the success probability of an attack with total complexity $M + N$ is below $2^{-s}(M + N)$.*

Our security claims for KEYAK are summarized in Table 1 with $|T|$ is the tag size (i.e., $|T| = \tau$, unless truncated). In our named instances we target security strength 128 bits by taking $c = 256$, $\tau = 128$ and $|K| \geq 128$.

The security claim in Table 1 assumes adversaries targeting a single key. 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 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: $|K| \geq 128 + \log_2 n_{\max}$. Alternatively, an option that avoids erosion without increasing the length of keys consists in imposing universal nonce uniqueness (see also the definition of q_{iv} in Section 3.2). By this we mean that not only the combination (K, N) must be unique, but N has to be unique among all Keyak instances. Many use cases actually allow this. For example, one can take as nonce the combination of the unique IDs of the two communicating devices and a strictly incrementing session counter.

2.5.1 Security in the case of misuse

The security strengths claimed in Table 1 are for the *nominal case* as defined here.

Definition 2 (nominal case security). *We call the* nominal case *security of a Keyak instance one in which the nonce requirement on the data element N (mapping to public message number in CAESAR terminology) is enforced and that only releases decrypted ciphertext upon unwrapping if the cryptogram has a valid tag.*

We also discuss the security of less disciplined implementations as covered by the *misuse case*.

Definition 3 (misuse case security). *We call the* misuse case *security of a Keyak instance one in which the nonce requirement on the data element N (mapping to public message number in CAESAR terminology) may be violated and that may release decrypted ciphertext upon unwrapping even if the cryptogram has no valid tag.*

In the misuse case security degrades and hence we strongly advise implementers and users to respect the nonce requirement on N at all times and never release unverified decrypted ciphertext. We detail security degradation in the following paragraphs.

A nonce-violation on N in general breaks confidentiality of part of the plaintext. In particular, two Sessions that have the same input sequence (K, N, metadata fragments, plaintext fragments) will result in the same output (ciphertext, tag). We call such a pair of sessions in-sync. Clearly, in-sync sessions leak equality of inputs and hence also plaintexts. As soon as in-sync sessions get different input blocks, they lose synchronicity. If these input blocks are plaintext blocks, the corresponding ciphertext blocks leak the bitwise difference of the corresponding plaintext blocks. In case the parallelism is larger than 1, this happens independently in each Piston. In short, Pistons that are in-sync in two different sessions leak equality of input up to the first differing block and leak the bitwise difference of this differing block. We call this the *nonce-misuse leakage*.

Release of unverified decrypted ciphertext also has an impact on confidentiality as it allows an adversary to harvest key stream that may be used in the future by legitimate parties. An adversary can harvest Π key stream blocks.

Nonce violation and release of unverified decrypted ciphertext have no consequences for integrity and do not put the key in danger for Keyak. With the exception of key stream harvesting and nonce-misuse leakage, the claims in Table 1 remain valid.

2.6 Implementations

The reference implementation in C++ can be found in KeccakTools [7]. In addition, a number of optimized implementations can be found in the Keccak Code Package [8].

3 Security rationale

For its generic security, Motorist relies on the *full-state keyed duplex* (FSKD) construction. This construction differs from the plain duplex (or sponge) construction in that it allows absorbing data over the complete width of the state, rather than just its outer part. Squeezing, however, remains limited to the outer part of the state.

We will first formally define FSKD and discuss its generic security, Then we reduce the generic security of Motorist via the demonstration of decodability. Finally, we discuss the generic and specific security of Keyak.

3.1 The full-state keyed duplex construction

We define FSKD in Algorithm 4. It calls a b-bit permutation f and operates on a b-bit state. The state is initialized with the concatenation of a secret key K and a string σ_0 with $|K| + |\sigma_0| = b$. Then it supports duplexing calls, each one taking a b-bit input block σ_i and returning an r-bit output block Z_i. The FSKD is illustrated in Figure 12.

Algorithm 4 The full-state keyed duplex construction FSKD$[f, r]$

```
**Require:** $r < b$
**Instantiation:** FSKD $\leftarrow$ FSKD$[f, r]$
  State: FSKD.$s \leftarrow 0^b$

**Interface:** $Z$ = FSKD.Init$(K, \sigma_0)$ with $K \in \mathbb{Z}_2^*$, $\sigma_0 \in \mathbb{Z}_2^{b-|K|}$ and $Z \in \mathbb{Z}_2^r$
  $s \leftarrow K||\sigma_0$
  $s \leftarrow f(s)$
  **return** $\lfloor s \rfloor_r$

**Interface:** $Z$ = FSKD.Duplexing$(\sigma)$ with $\sigma \in \mathbb{Z}_2^b$, and $Z \in \mathbb{Z}_2^r$
  $s \leftarrow s \oplus \sigma_i$
  $s \leftarrow f(s)$
  **return** $\lfloor s \rfloor_r$
```

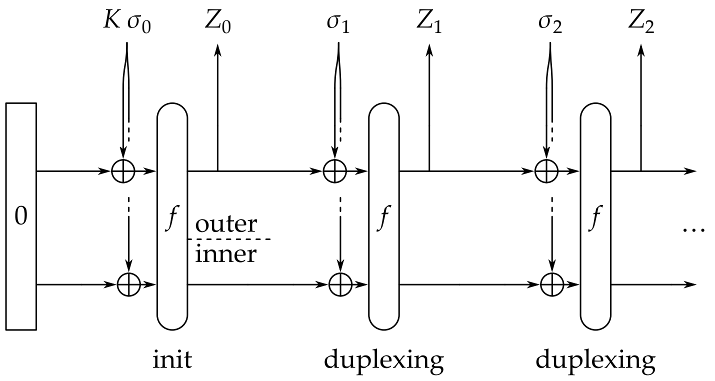

Figure 12 – *The full-state keyed duplex construction*

Clearly, the operation of Motorist can be expressed in terms of calls to FSKD objects.

3.2 Generic security of FSKD

When discussing the generic security, we express bounds in terms of the following resources of the attacker:

M Data complexity, total number of blocks fed to the FSKD object;

N Computational complexity, total number of calls to f or f^{-1};

q Number of calls to FSKD.Init(K, σ_0);

q_{iv} Maximum over all σ_0 values of the number of different K values in calls to FSKD.Init(K, σ_0).

The generic security of the FSKD construction was investigated by Mennink, Reyhanitabar and Vizár [14]. The FSKD is actually a slight variant of the object they consider, as they absorb the key in the inner part, whereas our definition puts the key in the outer part. When f is a random permutation, they prove an upper bound for the advantage of distinguishing it from a random oracle, namely:

$$(1 + 2^{-r})\frac{M^2}{2^c} + \frac{\mu N}{2^{|K|}}, \tag{1}$$

with b the width of f, c the capacity and $r = b - c$, and where K is uniformly distributed over $\mathbb{Z}_2^{|K|}$ for $|K| \leq c$. The parameter $\mu \leq M$ is the total maximum multiplicity [1], whose expected value depends on the circumstances of an attack.

The factor $\frac{\mu N}{2^{|K|}}$ in the bound (1) suggests that the key strength erodes for adversaries who can set $\mu \gg 1$. This is however a side-effect of the proof in [14]. In order to obtain a bound that does not have this artefact and that does not contain the somewhat speculative multiplicity μ, we have been working on a proof for an improved bound. Our proof combines elements of [14] and [1] with insights from other papers and some new ones. It is applicable to most of the sponge-based authenticated encryption schemes proposed to date and is the subject of a paper we hope to publish soon. In this document we just state it without proof, for deriving from it bounds for the generic security of Motorist in the nominal case and in the misuse case.

3.2.1 Distinguishing bounds for FSKD

Theorem 1 (FSKD distinguishing bound). *The advantage of distinguishing between:*

- *an array of FSKD$[f, r]$ objects, with f a random permutation and initialized with an array of keys K sampled uniformly but without replacement from $\mathbb{Z}_2^{|K|}$ with $|K| \leq r$,*
- *an array of as many random-oracle based objects with the same interface,*

by any adversary $\mathcal{A}$ is upper bounded by:

$$\frac{q_{\text{iv}} N}{2^{|K|}} + \frac{MN}{2^c} + \frac{M^2}{2^{c+1}}.$$

We can define a restricted adversary that is meaningful in the context of the generic security of Motorist. In particular, this restricted adversary corresponds to the nominal case for Motorist, with the exception of forgetting. Before stating this bound, we define an auxiliairy function that we use in its expression.

Definition 4. *The* multicollision limit function $\nu(R, M, C)$*, with R, M and C natural numbers returns a natural number and is defined as follows. Assume we randomly distribute M balls in R urns and we call the number of balls in the urn with the highest number of balls* $\mu(R, M)$*. Then* $\nu(R, M, C)$ *is defined as the smallest natural number x that satisfies:*

$$\Pr\left(\mu(R, M) > x\right) \leq \frac{x}{C}\,.$$

In words, when randomly sampling (with replacement) M elements from a set of R elements, the probability that there is an multicollision with more than ν elements is smaller than ν/C. The multicollision limit function actually allows us to bound the multiplicity in an attack where the adversary cannot control the value of the outer part of the state.

Theorem 2 (FSKD distinguishing bound against restricted adversary)**.** *The advantage of distinguishing between:*

- *an array of FSKD*$[f, r]$ *objects, with f a random permutation and initialized with an array of keys K sampled uniformly but without replacement from* $\mathbb{Z}_2^{|K|}$ *with* $|K| \leq r$*,*
- *an array of as many random-oracle based objects with the same interface,*

by any adversary $\mathcal{A}$ *that respects* σ_0 *being a nonce per key and that must provide* σ_i *before observing* Z_{i-1}*, is upper bounded by:*

$$\frac{q_{\text{iv}}N}{2^{|K|}} + \frac{2\nu(2^r, M, 2^c)(N+1)}{2^c} + \frac{Mq}{2^{c+|K|}} + \frac{3M^2}{2^b}\,.$$

3.2.2 Selection of capacity and key length

When using FSKD with some given permutation f in a mode, one must make choices for $|K|$ and c. We assume the goal is to achieve a certain security strength s. Often it is meaningful to impose an upper limit to the data complexity: $M \leq 2^d$. We call d the *data limit exponent*. A cryptosystem that achieves security strength $s = 128$ up to $d = 96$ offers a solid level of security for the foreseeable future. Note that the absence of a limitation on the data complexity corresponds to $d = s$.

First we discuss the key length $|K|$. Here the treatment is the same for both unrestricted and restricted cases. For an attack targeting a single key, q_{iv} equals 1 and the keylength must simply satisfy $|K| \geq s$. For attacks targeting multiple keys, q_{iv} is upper bounded by that number of keys and the keylength must satisfy $|K| \geq s + \log_2(\#K)$ with $\#K$ the number of keys under attack. Imposing that σ_0 is a global nonce, q_{iv} equals 1 even for multi-target attacks and we can again take $|K| \geq s$.

As for the choice of the capacity c, this differs for the two cases. For the unrestricted adversary, both terms $MN/2^c$ and $M^2/2^{c+1}$ impose that $c \geq s + d$. In the absence of a limitation on the data complexity ($d = s$) we get the classical birthday bound $c \geq 2s$.

For the restricted adversary, we must take a closer look at $\nu(2^r, 2^d, 2^c)$. As a matter of fact, it is interesting to investigate when $\nu(2^r, 2^d, 2^c) = 1$, the lowest possible value. In that case the term in the bound with 2^c in denominator reduces to $(N+1)/2^{c-1}$ imposing only $c \geq s + 1$. The implication of $\nu(2^r, 2^d, 2^c) = 1$ is that if we sample 2^d values randomly from a domain of size 2^r, the probability of a collision, given by $2^{2d-(r+1)}$ (see, e.g., [2]), must be smaller than 2^{-c}. This yields the following condition for c: $c \leq (b+1)/2 - d$. This gives an equation of what can be achieved with a b-bit permutation if we wish to have a security strength of $c - 1$ in the nominal case: $s + d \leq (b-1)/2$. An 800-bit permutation provides a comfortable 400 bits to be distributed among the security strength and the data limit exponent. Even a 400-bit permutation would be work in this regime by taking $c = 136, r = 264$ to yield the security strength $s = 128$ with limitation $M \leq 2^{70}$. In the absence of a data limit ($d = s$) we can reach a security strength s equal to $c - 1$ for $s \leq (b-1)/4$.

3.3 Decodability of Motorist

Lemma 1. *For any sequence of queries Q to a Motorist instance that does not result in an error, and knowing when a knot occurs, the SUV and the full sequence of messages can be unambiguously recovered from the input block sequences to its Piston objects.*

Proof. (sketch) As specified in the Engine.WRAP() interface, the Engine will make exactly one single inject call and at most one crypt call in between spark calls. Moreover, at the end of processing a message, an SUV or a knot operation, it will indicate this in the spark call and retrieve tags. So, in each input block, each Piston sets its four fragment offsets to the correct values. As explained in Section 1.4, the EOM allows delimiting the last input blocks containing SUV, the last input block containing message input and the last input blocks containing chaining values. In combination with EOM for the previous input block, the offset Crypt End allows determining the plaintext fragments in an an input block. Metadata, SUV or chaining value fragments can be determined with offsets Inject Start and Inject End. Once all fragments are identified, the SUV, plaintext, metadata and chaining values of messages can be reconstructed by simply concatenating the fragments. □

3.4 Security of Motorist

From Lemma 1 it follows that if the SUV is unique per session, the tags and key streams are hard to distinguish from random strings. The generic privacy and authenticity security hence depend on the advantage of distinguishing the output of the Motorist from that of a random oracle. Here we can make use of Theorem 1 and Theorem 2.

3.4.1 Nominal case

Let us first consider the nominal case, where we also exclude that the adversary calls Motorist.MAKEKNOT() or that she inputs a tag as metadata—we will treat these aspects shortly. In wrap operations, the privacy security equals the bound in Theorem 2. In unwrap operations, the adversary can harvest key stream blocks if she succeeds in forging a cryptogram. S attempts of forging a tag have a probability of success $S/2^{\tau}$. Per forged cryptogram, she can harvest Π blocks of key stream. So we consider that a successful cryptogram forgery implies a break of privacy.

During a session, an adversary can take a tag and input it as metadata in the next wrap operation. This has the effect that c' bits of the outer state of the FSKD in Piston 0 are set to 0. This is also the case with a call to Motorist.MAKEKNOT(). Assuming that $c' < r$, there remain $r - c'$ bits in the outer part of the state that are not controlled by the adversary. This gives rise to an additional term $\nu(2^{r-c'}, H, 2^c)N2^{-c}$ with H the number of Motorist.MAKEKNOT() calls and tags output. If $\log_2(H) < r + 1 - (c + c')$, ν reduces to 1. This is always satisfied if $c < b/4$. Otherwise, we have to analyze it specifically, as we do for RIVER KEYAK in Section 3.5.

When $\Pi > 1$, we have to consider the tag consisting of output bits of the Piston with index 0. It depends on the output bits of the other Piston objects via the chaining values.

An adversary could try to build a forgery by means of a collision in such a chaining value. This would require a pair of query sequences Q and Q' that exhibit this collision. Due to the fact that Engine imposes synchronicity between Piston objects, the two colliding Piston FSKD inputs must have the same length, be initialized with the same SUV and have the same diversifiers to be usable for a forgery. It follows that any new attempt to generate a collision requires a new session. As for the success probability, each attempt at a collision in a chaining value requires an active attempt targeted at a specific chaining value in a specific session. The probability of success per attempt is hence given by $2^{-c'}$. As the tag length in Motorist is limited to c', the success probability is not larger than that of forging the tag directly.

We will now give bounds for Motorist restricted to large permutations, i.e., $s + d < b/2$, so that the regime $\nu(2^r, 2^d, 2^c) = 1$ applies.

Theorem 3. *The authenticated encryption mode Motorist defined in Section 1.6 with SUV fitting in a single input block and equal to $SUV = enc(K)||N$ for multiple keys K sampled randomly without replacement and N unique per key, satisfies the following security level against any single adversary $\mathcal{A}$ in the nominal case:*

$$\mathrm{Adv}^{\mathrm{priv}}_{\textsc{Motorist}[f,\Pi,W,c,\tau]}(\mathcal{A}) \leq \frac{q_{\mathrm{iv}}N}{2^{|K|}} + \frac{(\nu(2^{r-c'}, H, 2^c)+1)(N+1)}{2^c} + \frac{Mq}{2^{c+|K|}} + \frac{3M^2}{2^b} + \frac{S}{2^\tau}\,, \quad \textit{and}$$
$$\mathrm{Adv}^{\mathrm{auth}}_{\textsc{Motorist}[f,\Pi,W,c,\tau]}(\mathcal{A}) \leq \frac{q_{\mathrm{iv}}N}{2^{|K|}} + \frac{(\nu(2^{r-c'}, H, 2^c)+1)(N+1)}{2^c} + \frac{Mq}{2^{c+|K|}} + \frac{3M^2}{2^b} + \frac{S}{2^\tau}\,,$$

if $K \xleftarrow{\$} \mathbb{Z}_2^{|K|}$, with f a randomly chosen permutation and with S ($\leq M$) the total number of forged cryptograms the adversary attempts to unwrap and H ($\leq M$) the total number of calls to Motorist.$\textsc{MakeKnot}()$ *and tags output.*

3.4.2 Misuse case

In the misuse case there is a loss of security as described in Section 2.5.1 but it does not degenerate completely. We now give a bound for the remaining security level.

Theorem 4. *The authenticated encryption mode Motorist defined in Section 1.6 satisfies the following security level against any single adversary $\mathcal{A}$ in the misuse case (modulo nonce-misuse leakage and key stream harvesting):*

$$\mathrm{Adv}^{\mathrm{priv}}_{\textsc{Motorist}[f,\Pi,W,c,\tau]}(\mathcal{A}) \leq \frac{q_{\mathrm{iv}}N}{2^{|K|}} + \frac{MN}{2^c} + \frac{M^2}{2^{c+1}}\,, \quad \textit{and}$$
$$\mathrm{Adv}^{\mathrm{auth}}_{\textsc{Motorist}[f,\Pi,W,c,\tau]}(\mathcal{A}) \leq \frac{q_{\mathrm{iv}}N}{2^{|K|}} + \frac{MN}{2^c} + \frac{M^2}{2^{c+1}} + \frac{S}{2^\tau}\,,$$

if $K \xleftarrow{\$} \mathbb{Z}_2^{|K|}$, f is a randomly chosen permutation and with S ($\leq M$) the total number of forged cryptograms the adversary attempts to unwrap.

Note that the value of q_{iv} in our theorems above only cover the case for an SUV fitting a single input block. We conjecture that our proof can be generalized to cover multi-block SUV, yielding the same bound.

3.5 Security of Keyak

For the security of Keyak against generic attacks, we can simply apply Theorem 3. Note that for the estimation of the maximum q_{iv}, we must distinguish between the part of the

SUV injected in the same block as the key pack, and the remaining part. In case the SUV fits in a single block, this distinction evaporates.

For River Keyak we must estimate the value of $\nu(2^{r-c'}, H, 2^c)$. Limiting H to 2^{128} yields $\nu(2^{288}, 2^{128}, 2^{256})$. The number of balls in a bin has a Poisson distribution with $\lambda = 2^{128-288} = 2^{-160}$. The probability for any single urn to have less than n balls is approximately $1 - \frac{2^{-160n}}{n!}$ and the probability for all 2^{288} urns to have n balls or less is hence $(1 - \frac{2^{-160n}}{n!})^{2^{288}} \approx 1 - \frac{2^{288-160n}}{n!}$. Taking $n = 4$ yields $1 - \frac{2^{288-640}}{24}$ so the probability that $\mu = 3$ is about 2^{-356}. So we conclude $\nu(2^{288}, 2^{128}, 2^{256}) = 3$. (For the other Keyak instances, we have $c < b/4$ and thus $\nu(2^{r-c'}, H, 2^c) = 1$.)

As for non-generic attacks, we believe that the permutations Keccak-$p[1600, n_\text{r} = 12]$ and Keccak-$p[800, n_\text{r} = 12]$ do not have properties that could be exploited to mount attacks that would be more efficient than generic ones. Regarding the properties of underlying permutations, we refer to [2, 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 [5]. 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 [9, 12].

The most powerful attacks on modes using Keccak-p were the cube attacks in [10, 11], until recently an improved cube attack was published on the IACR ePrint archive [13]. It presents nonce-repeating attacks that recover the key from a 7-round version of Lake Keyak with 2^{42} input blocks and an 8-round version with 2^{74} input blocks. None of these attacks exploit the additional degrees that full-state absorbing gives. We have attempted extending the existing attacks by one round using these degrees of freedom prior to the appearance of [13]. However, this publication uses techniques similar to the ones we were studying, but without the need for full-state absorbing. Whether these attacks can still be extended to more rounds by exploiting full-state absorbing remains an open question.

3.6 Keyak variants with 256-bit security strength

Some users may wish to use an authentication encryption scheme in a consistent combination with cryptographic functions of 256-bit security strength. We feel that, as such, 256-bit security does not provide a practical and tangible security improvement over 128-bit security. A cipher that stands by its claim of 128-bit security provides enough protection against any adversary in the foreseeable future.

This being said, inspection of Theorem 3 reveals a generic security strength can be achieved of 256 bits for the nominal case by all our named Keyak instances except River Keyak. It suffices to increase the tag length to 256 bits. To achieve 256 bits of security strength against shortcut attacks, we recommend increasing the number of rounds in Keccak-p from 12 to 14.

4 Using Keyak in the context of CAESAR

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

4.1 Specification and security goals

The specifications can be found in Section 2 and the security goals in Section 2.5.

4.2 Security analysis and design rationale

The security analysis and design rationale can be found in Section 3.

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 $c = 256$.

KEYAK has the following security assurance features:

- Generic security of the Motorist 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 [4]. For the layers above, rationales are given in Section 3.

4.3 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.

4.4 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.

4.5 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.

4.6 CAESAR use cases

For all 5 named KEYAK instances, we primarily target Use Case 3: defense in depth. In the mail of the CAESAR secretary dated 16 July 2016 20:36:10, the following criteria were listed:

1. critical: authenticity despite nonce misuse
2. desirable: limited privacy damage from nonce misuse
3. desirable: authenticity despite release of unverified plaintexts
4. desirable: limited privacy damage from release of unverified plaintexts
5. desirable: robustness in more scenarios; e.g., huge amounts of data

We claim that KEYAK satisfies the five criteria.

Points 2 and 4 deserve some explanations. In case of nonce misuse or release of deciphered ciphertext, the *limited privacy damage* consists of the leading plaintext block equality, of the first differing plaintext block differences and of key stream block harvesting. The features of KEYAK allow the user to easily prevent the misuse cases.

- The session mechanism reduces the need for nonces. Often, exchanged messages can be naturally grouped in a session, such as in a network connection, a smartcard transaction or a chat application. In many protocols, the key is a one-time session key, in which case no nonce is needed at all. When it is instead a long-term key, the nonce is required only per session.
- KEYAK supports a variable-length nonce field N allowing users to put multiple data elements to reduce the risk of nonce repetition. For robustness, this may include elements related to the context of the session, e.g., date and time, identity of sender and identity of receiver, session number of communication, etc.
- Key stream block harvesting can be excluded by using the *tag on session setup* feature of Motorist. When starting up Motorist for unwrapping, one can set the tag flag, requiring the presence of a tag in the startup. Without this tag, Motorist will refuse to start up and hence not return so-called deciphered ciphertext. This tag is supposed to come from the wrapping Motorist object. Without it, the only thing an adversary can do to obtain key stream blocks from an unwrapping Motorist object is take a guess at this tag value.

For Point 5, we highlight the following robustness features of KEYAK.

- Even in the misuse case, an adversary cannot retrieve the internal state nor the key.

- Processing huge amounts of data does not result in security breakdown, even in the misuse case. In particular, the 2^{64} blocks birthday bound observed in AES-based modes does not play for any of the named Keyak instances.

- Keyak offers robustness w.r.t. side-channel attacks.

  - Motorist lends itself for protection against side channel attacks. As opposed to block cipher modes, there are no round keys being used during operation that can be attacked. The security is based on the secrecy of the evolving inner states of the Piston objects.
  - Motorist lends itself for protection against differential fault analysis in the nominal case. In wrapping, the unique nonce makes that it is very unlikely that differences due to faults after the starting phase can be exploited. In unwrapping, the fault will with high probability trigger a tag to be invalid.
  - Keccak-*p* lends itself to protection against side channels as it can easily be implemented in constant-time and is suitable for masking and threshold schemes.
  - The forget mechanism in Motorist provides forward secrecy. Even if a side-channel attack would reveal the entire state, the attacker cannot recover the state prior to the forget point. A fortiori, one cannot go back to the key.

For all 5 named Keyak instances except River Keyak, we also target Use Case 2: high-performance applications, as they are

- efficient on 64-bit CPUs and very efficient on dedicated hardware;
- efficient on 32-bit CPUs;
- constant-time when message length is constant.

Finally, for River Keyak we also address Use Case 1: lightweight applications, as it:

- fits into small hardware area and small code for 8-bit and 32-bit CPU;
- has a natural ability to protect against side-channel attacks;
- offers competitive hardware performance, including energy/bit;
- is relatively fast on 8-bit CPU;
- can be combined with a sponge-based hash function based on Keccak-*p*[800].

References

[1] E. Andreeva, J. Daemen, B. Mennink, and G. Van Assche, *Security of keyed sponge constructions using a modular proof approach*, Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers (Gregor Leander, ed.), Lecture Notes in Computer Science, vol. 9054, Springer, 2015, pp. 364–384.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Cryptographic sponge functions*, January 2011, `http://sponge.noekeon.org/`.

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

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

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

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

[7] ———, KECCAKTOOLS *software*, September 2015, `https://github.com/gvanas/KeccakTools`.

[8] ———, KECCAK *code package*, June 2016, `https://github.com/gvanas/KeccakCodePackage`.

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

[10] I. Dinur, P. Morawiecki, J. Pieprzyk, M. Srebrny, and M. Straus, *Practical complexity cube attacks on round-reduced keccak sponge function*, Cryptology ePrint Archive, Report 2014/259, 2014, `http://eprint.iacr.org/`.

[11] ———, *Cube attacks and cube-attack-like cryptanalysis on the round-reduced keccak sponge function*, Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I (E. Oswald and M. Fischlin, eds.), Lecture Notes in Computer Science, vol. 9056, Springer, 2015, pp. 733–761.

[12] 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/`.

[13] Senyang Huang, Meiqin Wang, Xiaoyun Wang, and Jingyuan Zhao, *Conditional cube attack on reduced-round keccak sponge function*, Cryptology ePrint Archive, Report 2016/790, 2016, `http://eprint.iacr.org/2016/790`.

[14] B. Mennink, R. Reyhanitabar, and D. Vizár, *Security of full-state keyed and duplex sponge: Applications to authenticated encryption*, Cryptology ePrint Archive, Report 2015/541, 2015, `http://eprint.iacr.org/`.

Acknowledgments

We acknowledge the following people that have helped us in improving and better understanding KEYAK: Seth Hoffert for bringing our attention to what an adversary can do with long tags; Itai Dinur, Paweł Morawiecki, Josef Pieprzyk, Marian Srebrny, Michał Straus for their cube attacks and Senyang Huang, Meiqin Wang, Xiaoyun Wang, Jingyuan Zhao for improving those; Bart Mennink, Reza Reyhanitabar and Damian Vizár for their work on the generic security of full-state keyed duplex; Monika Seidlová for her investigations of higher-order differential attacks on KECCAK-p; Jos Wetzels and Wouter Bokslag for their work on KEYAK hardware implementation and finally anonymous CAESAR committee members for their second round comments.

A Change log

A.1 From 1.0 to 1.1

Only Section 4.3 (“Features”) changed to include a brief comparison with AES-GCM.

A.2 From 1.1 to 1.2

The main change is the correction of the expressions for the advantage of forging ciphertext-tag pairs in two theorems.

In both cases a term 2^{-t} that was there before has been replaced by $\frac{S}{2^t}$, with t is the tag length and S the number of submitted tags. This term expresses the probability of tag forging by pure chance, in the former case in a single attempt and in the latter case in S attempts. In the new expression we assume the adversary gets one forgery attempt for each submitted tag, while the old expression carried the implication that only a single tag forging attempt is considered. We thank Bart Mennink for bringing this error to our attention.

We also added a section with a reference to the available implementations.

A.3 From 1.2 to 2.0

The mode underlying Keyak has been completely re-factored and so has the document. Keyak remains an authenticated encryption scheme supporting sessions, based on 12-round Keccak-p permutations and the named instances still have security strength 128 bits. We turned Keyak into a parameterized authenticated encryption scheme, supporting a wide range of parameters. The named instances, to which we added one named Lunar Keyak, are defined by fixing parameters in the general Keyak scheme.

A.4 From 2.0 to 2.1

We added Figures 1–2 in the original text, and the new Section 1.7 with further illustrations and examples (Figures 3–11).

We added Section 2.6 on implementations.

No change has been made to any of the algorithms. The Motorist mode, the Keyak functions and their security claims remain unchanged.

A.5 From 2.1 to 2.2

The changes are:

- In Motorist, the tag length is now limited to the capacity length. This does not affect the Keyak functions.
- The definitions of Piston, Engine and Motorist have been simplified to ease understanding by the reader. In particular, Piston and Engine have slightly different behaviours and interfaces. There are however no difference at the Motorist interface and its behaviour remains fully identical to the previous version. This means that there is no change in the test vectors for Keyak and that there is no impact on existing optimized implementations. It is however suggested to update reference implementations that follow closely the Piston and Engine algorithms to match the current description. The most important changes are summarized below.
  - In Motorist.Wrap(), the loops and calls to Engine.Crypt() and Engine.Inject() are now replaced by a single loop that calls the merged interface Engine.Wrap() until both streams are exhausted.
  - In Engine, the E_t attribute is now stored in each Piston object, which now stores separate crypt (ω_C) and inject (ω_I) offsets. These offsets are updated by Piston.

  - In Engine, the state machine attribute PHASE is removed. Engine now relies on Motorist for the consistency of the operation sequence.
  - The Engine.CRYPT() and Engine.INJECT() interfaces are now merged into a single Engine.WRAP() interface.
  - The flags CRYPTINGFLAG and EOMFLAG at Piston interface are removed. These are now managed internally by Piston.
  - Piston.SPARK() now only applies f to the Piston state.
  - The setting of offset EOM is moved to Piston.GETTAG(), which now also applies f to the Piston state before extracting the tag bytes and updating the crypt offset.

- Sections 2.5 and 3 have been restructured and updated with new results on generic security and with the latest third-party cryptanalysis.
- We added Section 4.6 as required for the CAESAR competition.

The KEYAK functions and their security claims remain unchanged.