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 **2.0**
August 29, 2015

[1]STMicroelectronics
[2]Radboud University

Contents

This document specifies KEYAK v2, a parameterized permutation-based authenticated encryption scheme with support for associated data and sessions. As for KEYAK v1, its underlying permutation is KECCAK-p. It is however based on a new mode, called Motorist, which is more efficient than the modes underlying KEYAK v1 and relies on some recently published insights for its provable generic security. KEYAK v1 consisted of four named instances. For KEYAK v2, we now formulate a generic definition and added one named instance. 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.

Due to time restrictions, some sections in this document are not as complete as we would like and will soon publish an improved version. However, the specification of Motorist and KEYAK and the security claims of the latter will not change.

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 [11]. 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 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.

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 [11] 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 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 know 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 absorbing rate, the state width minus the last part containing the fragment offsets. When being called to inject, it receives a handle 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. It is the responsibility of the Engine to ensures that the SUV and message sequence can be reconstructed from the sponge input to each Piston objects 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 $\text{enc}_8(x)$ 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.doSomething, popular in object oriented programming. Concretely, a byte stream is a string of bytes that supports the following functions, similarly to a queue:

- $z \leftarrow X$.pullByte removes the first byte of stream X and assigns it to z;
- X.pushByte(z) appends byte z to the end of the stream X;
- X.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;
- Clear X: 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 pushByte() and an output byte stream does not have to support 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 Piston.Inject(X). When a byte stream figures as the parameter in a function call, it should be seen as a *handle* to the byte stream being passed. The object supporting the called function can use this handle 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.

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.

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.Crypt$(I, O, \omega,$ unwrapFlag$)$ 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 unwrapFlag indicates whether it is encryption (False) or decryption (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 parameter ω specifies the index in the state from where the plaintext fragment must be injected. The fragment 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.

The function Piston.Inject$(X,$ cryptingFlag$)$ injects metadata taken from input stream X. The Boolean parameter indicates whether the current input block already has a plaintext fragment (True) or not (False). If so, Piston starts injecting from index R_s, otherwise it starts from index 0. 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.

The function Piston.Spark(eomFlag, ℓ) applies the underlying permutation f to the state. Before it does that, it codes in the data element EOM whether this is the last input block of a message (or of string injected collectively) as indicated by eomFlag and, if so, the number ℓ of bytes of the state after the application of f that are reserved as tag.

Finally, the function Piston.GetTag(T, ℓ) writes the first ℓ bytes of the state to output byte stream T, to be used as a tag or chaining value.

One could specify 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. The only thing that is essential is that if an input block contains a plaintext fragment, this must be announced before injecting metadata.

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 $R_s \leq R_a \leq \frac{f.b-32}{8}$

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

Interface: Piston.Crypt$(I, O, \omega, \textsc{unwrapFlag})$ with I, O input and output byte streams
while $(I.\textsc{hasMore} = \textsc{True})$ AND $(\omega < R_s)$ **do**
  $x \leftarrow I.\textsc{pullByte}$
  $O.\textsc{pushByte}(s[\omega] \oplus x)$
  if $\textsc{unwrapFlag} = \textsc{True}$ **then**
    $s[\omega] \leftarrow x$
  else
    $s[\omega] \leftarrow s[\omega] \oplus x$
  $\omega \leftarrow \omega + 1$
s[Crypt End] $\leftarrow s$[Crypt End] $\oplus \text{enc}_8(\omega)$

Interface: Piston.Inject$(X, \textsc{cryptingFlag})$ with X an input byte stream
if $\textsc{cryptingFlag} = \textsc{True}$ **then**
  $\omega \leftarrow R_s$
else
  $\omega \leftarrow 0$
s[Inject Start] $\leftarrow s$[Inject Start] $\oplus \text{enc}_8(\omega)$
while $(X.\textsc{hasMore} = \textsc{True})$ AND $(\omega < R_a)$ **do**
  $s[\omega] \leftarrow s[\omega] \oplus X.\textsc{pullByte}$
  $\omega \leftarrow \omega + 1$
s[Inject End] $\leftarrow s$[Inject End] $\oplus \text{enc}_8(\omega)$

Interface: Piston.Spark$(\textsc{eomFlag}, \ell)$ with $\ell \leq R_s$ and $\ell = 0$ if $\textsc{eomFlag} = \textsc{False}$
if $\textsc{eomFlag} = \textsc{True}$ **then**
  if $\ell = 0$ **then**
    s[EOM] $\leftarrow s$[EOM] $\oplus \text{enc}_8(255)$
  else
    s[EOM] $\leftarrow s$[EOM] $\oplus \text{enc}_8(\ell)$
else
  s[EOM] $\leftarrow s$[EOM] $\oplus \text{enc}_8(0)$ (i.e., do nothing)
$s \leftarrow f(s)$

Interface: Piston.GetTag(T, ℓ) with T an output byte stream
if $\ell > R_s$ **then**
  return error
for $i \leftarrow 0$ to $\ell - 1$ **do**
  $T.\textsc{pushByte}(s[i])$

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. For each piston, Engine remembers in the attribute E_t how much output was used as tag or chaining value, so as to pass this to Piston.Crypt() and avoid reusing the bits as key stream. Engine also maintains a state machine via the attribute PHASE to govern the sequence of function calls supported and thereby to enforce consistency.

The phase mainly indicates how the Π input blocks are being constructed in the Π Piston objects:

FRESH They are empty.

CRYPTED They have a plaintext fragment and more plaintext is coming.

ENDOFCRYPT They have a plaintext fragment and no more plaintext is coming.

ENDOFMESSAGE They have their fragments ready and the message has been fully injected.

The application of f on the Π states is centralized in the Engine.Spark(EOMFLAG, ℓ) internal function. This function takes as input a flag telling whether the message (or the string injected collectively) is finished and how many bits to reserve for a tag or for a chaining value. This last parameter, ℓ, is in fact a vector, allowing one to take a different number of bits in each Piston, and these numbers are stored in E_t.

The function Engine.Crypt($I, O,$ UNWRAPFLAG) dispatches the input I to the Π Piston objects and collects the corresponding Π output in O. Each Piston object takes a fragment from I, so the Pistons process in total up to ΠR_s bytes. The phase switches to CRYPTED, or to ENDOFCRYPT if the input stream is exhausted. The UNWRAPFLAG is as for Piston.Crypt().

The function Engine.Inject(A) dispatches the metadata A to the Π Piston objects. Each Piston object takes a fragment from A, so the Pistons process in total up to $\Pi(R_a - R_s)$ bytes (if Engine.Crypt() was called before) or ΠR_a bytes (otherwise). If both the input and the metadata streams are exhausted, it switches the phase to ENDOFMESSAGE and delays the application of f until the call to Engine.GetTags(). Otherwise, it calls Engine.Spark() to perform f on all Π Piston objects and switches the phase back to FRESH.

The function Engine.GetTags(T, ℓ) calls Piston.Spark() on all Π Piston objects and collects the corresponding tags into the output stream T. It then switches the phase back to FRESH. The parameter ℓ is as in Engine.Spark().

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.

After the whole stream X is processed, the phase is switched to ENDOFMESSAGE.

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*

Algorithm 2 Definition of ENGINE[Π, Pistons]

Require: $1 \leq \Pi \leq 255$
Require: Pistons is an array of Π pistons

Instantiation: Engine ← ENGINE[Π, Pistons]
Phase of the Engine: Engine.PHASE ← FRESH
Output bytes reserved for tag: Engine.$E_t \leftarrow 0^\Pi \in \mathbb{N}^\Pi$

Internal interface: Engine.Spark(EOMFLAG, ℓ) with $\ell \in \mathbb{N}^\Pi$
for $i \leftarrow 0$ to $\Pi - 1$ **do** Pistons[i].Spark(EOMFLAG, $\ell[i]$)
$E_t \leftarrow \ell$

Interface: Engine.Crypt(I, O, UNWRAPFLAG) with I, O input and output byte streams
if PHASE $\neq$ FRESH **then return** error
for $i \leftarrow 0$ to $\Pi - 1$ **do** Pistons[i].Crypt($I, O, E_t[i]$, UNWRAPFLAG)
if I.HASMORE **then**
  PHASE ← CRYPTED
else
  PHASE ← ENDOFCRYPT

Interface: Engine.Inject(A) with A an input byte stream
if PHASE $\notin$ {FRESH, CRYPTED, ENDOFCRYPT} **then return** error
CRYPTINGFLAG ← (PHASE $\in$ {CRYPTED, ENDOFCRYPT})
for $i \leftarrow 0$ to $\Pi - 1$ **do** Pistons[i].Inject(A, CRYPTINGFLAG)
if (PHASE = CRYPTED) OR (A.HASMORE = TRUE) **then**
  Engine.Spark(FALSE, 0^Π)
  PHASE ← FRESH
else
  PHASE ← ENDOFMESSAGE

Interface: Engine.GetTags(T, ℓ) with T an output byte stream and $\ell \in \mathbb{N}^\Pi$
if PHASE $\neq$ ENDOFMESSAGE **then return** error
Engine.Spark(TRUE, ℓ)
for $i \leftarrow 0$ to $\Pi - 1$ **do** Pistons[i].GetTag($T, \ell[i]$)
PHASE ← FRESH

Interface: Engine.InjectCollective(X, DIVERSIFYFLAG) with X an input byte stream
if PHASE $\neq$ FRESH **then return** error
Let $X_t[i]$ be an array of Π local byte streams, initially empty
while X.HASMORE = TRUE **do**
  $x \leftarrow X$.PULLBYTE
  for $i \leftarrow 0$ to $\Pi - 1$ **do** $X_t[i]$.PUSHBYTE(x)
if DIVERSIFYFLAG = TRUE **then**
  for $i \leftarrow 0$ to $\Pi - 1$ **do** $X_t[i]$.PUSHBYTE($\text{enc}_8(\Pi)$)
  for $i \leftarrow 0$ to $\Pi - 1$ **do** $X_t[i]$.PUSHBYTE($\text{enc}_8(i)$)
while ($X_t[0]$.HASMORE = TRUE) **do**
  for $i \leftarrow 0$ to $\Pi - 1$ **do** Pistons[i].Inject($X_t[i]$, FALSE)
  if ($X_t[0]$.HASMORE = TRUE) **then** Engine.Spark(FALSE, 0^Π)
PHASE ← ENDOFMESSAGE

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 strength: 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 c bits (for the inner part) or 32 bits (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.

To make a tag 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 function Motorist.StartEngine(SUV, TAGFLAG, T, UNWRAPFLAG, FORGETFLAG) starts 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 UNWRAPFLAG = 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, UNWRAPFLAG, FORGETFLAG) wraps a message or unwraps a cryptogram.

- To wrap, the function must be called with UNWRAPFLAG = 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 UNWRAPFLAG = 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 processing the input and the metadata. When the input stream is exhausted, it continues processing any remaining metadata. Note that when called with empty input and metadata streams, performs a call to Engine.Inject() to ensure that the Engine object enters the ENDOFMESSAGE phase. 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, 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. 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.

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:

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

Algorithm 3 Definition of MOTORIST$[f, \Pi, W, c, \tau]$.

Require: Π is the number of pistons, with $1 \leq \Pi \leq 255$
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 $\lceil \frac{c}{W} \rceil \leq \left\lfloor \frac{f.b - \max(c,32)}{W} \right\rfloor$
Require: τ is the tag length in bits, a multiple of W and $\tau \leq f.b - \max(c, 32)$

Instantiation: Motorist $\leftarrow$ MOTORIST$[f, \Pi, W, c, \tau]$
Squeezing byte rate: $R_s \leftarrow \frac{W}{8} \left\lfloor \frac{f.b - \max(c,32)}{W} \right\rfloor$
Absorbing byte rate: $R_a \leftarrow \frac{W}{8} \left\lfloor \frac{f.b - 32}{W} \right\rfloor$
Chaining value length: Motorist.$c' \leftarrow W \lceil \frac{c}{W} \rceil$
for $i \leftarrow 0$ to $\Pi - 1$ **do** Motorist.Pistons$[i] \leftarrow$ PISTON$[f, R_s, R_a]$
Motorist.Engine $\leftarrow$ ENGINE$[\Pi$, Motorist.Pistons$]$
Motorist.PHASE $\leftarrow$ READY

Interface: RES $\leftarrow$ Motorist.StartEngine$(SUV$, TAGFLAG, T, UNWRAPFLAG, FORGETFLAG$)$
if PHASE $\neq$ READY **then return** error
Engine.InjectCollective$(SUV$, TRUE$)$
if FORGETFLAG $=$ TRUE **then** Motorist.MakeKnot$()$
RES $\leftarrow$ Motorist.HandleTag$($TAGFLAG, T, UNWRAPFLAG$)$
if RES $=$ TRUE **then** PHASE $\leftarrow$ RIDING
return RES

Interface: RES $\leftarrow$ Motorist.Wrap$(I, O, A, T$, UNWRAPFLAG, FORGETFLAG$)$
if PHASE $\neq$ RIDING **then return** error
if $(I$.HASMORE $=$ FALSE$)$ AND $(A$.HASMORE $=$ FALSE$)$ **then**
  Engine.Inject(A)
while I.HASMORE $=$ TRUE **do**
  Engine.Crypt$(I, O,$ UNWRAPFLAG$)$
  Engine.Inject(A)
while A.HASMORE $=$ TRUE **do**
  Engine.Inject(A)
if $(\Pi > 1)$ OR $($FORGETFLAG $=$ TRUE$)$ **then** Motorist.MakeKnot$()$
RES $=$ Motorist.HandleTag$($TRUE, T, UNWRAPFLAG$)$
if RES $=$ FALSE **then** Clear O
return RES

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

Internal interface: RES $\leftarrow$ Motorist.HandleTag$($TAGFLAG, T, UNWRAPFLAG$)$
Let T' be a local byte stream, initially empty
if TAGFLAG $=$ FALSE **then**
  Engine.GetTags$(T', 0^{\Pi})$
else
  Engine.GetTags$(T', [\tau/8, 0^{\Pi-1}])$
  if UNWRAPFLAG $=$ FALSE **then**
    Copy T' into T
  else if $T' \neq T$ **then**
    PHASE $\leftarrow$ FAILED
    return FALSE
return TRUE

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

The additions and multiplications between the terms are in GF(2). With the exception of the value of the round constants $\mathrm{RC}[i_\mathrm{r}]$, 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 $\mathrm{RC}[i_\mathrm{r}][x,y,z]$ 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.

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 $\mathrm{pad10^*}[r](|M|)$. 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 $\mathrm{pad10^*}[r](|M|)$ returns the byte string `0x01` `0x00`$^{(q-7)/8}$.

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

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

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

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

2.3 Generic definition of KEYAK

KEYAK makes use of 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 keypack$(K, \ell_\text{k})||N$ with $\ell_\text{k} = \frac{W}{8}\left\lceil\frac{c+9}{W}\right\rceil$ and no limitation on the length of N.

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:

RIVER KEYAK $b = 800, \Pi = 1$

LAKE KEYAK $b = 1600, \Pi = 1$ (primary recommendation)

SEA KEYAK $b = 1600, \Pi = 2$

OCEAN KEYAK $b = 1600, \Pi = 4$

LUNAR KEYAK $b = 1600, \Pi = 8$

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. These lengths are chosen so that keypack$(K, \ell_\text{k})||N$ and the two bytes of diversification all fit in exactly one block.

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

Our security claims for KEYAK are summarized in Table 1, where the security strength is indicated with the logarithm base 2 of the expected attack cost and the unit is the execution of the underlying permutation, and where $|T|$ is the tag size (i.e., $|T| = \tau$, unless

	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: Security claims for KEYAK

truncated). In our named instance we target security strength 128 bits by taking $c = 256$, $\tau = 128$ and $|K| \geq 128$.

Although we make no claims above security strength $c/2$, we are aware that in most cases a security strength close to c can be achieved, see Section 3.2.

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. 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.

The security claim in Table 1 assume KEYAK implementations that respect the nonce requirement on the data element N (mapping to public message number in CAESAR terminology) and upon unwrapping only release plaintext if the cryptogram has a valid tag. Not respecting these requirements results in a degradation of security and hence we strongly advise implementers and users to respect the nonce requirement on N at all times and never release unverified plaintext. The security degradation is the following.

A nonce-violation on N in general breaks the confidentiality of the plaintext. It leaks the bitwise difference between the plaintext messages encrypted under the same N. In the case of a single, accidental, nonce violation, the situation is just a little bit better than with a stream cipher, as the leakage is limited to the first block where the input messages start to differ. Due to the way Motorist works, the subsequent blocks will not lose confidentiality. Release of unverified plaintext 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. Nonce violation and release of unverified plaintext have no consequences for integrity and do not put the key in danger for KEYAK.

3 Security rationale

For its generic security, Motorist relies on the *full-state keyed duplex* (FSKD) construction. This construction differs from the proper duplex or (sponge) construction in that is 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 the full-state keyed duplex (FSKD) construction 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 1.

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

Require: $r < f.b$

Instantiation: FSKD $\leftarrow$ FSKD$[f, r]$
State: FSKD.$s \leftarrow 0^{f.b}$

Interface: $Z =$ FSKD.Init(K, σ_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(σ) 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 $\lfloor s \rfloor_r$

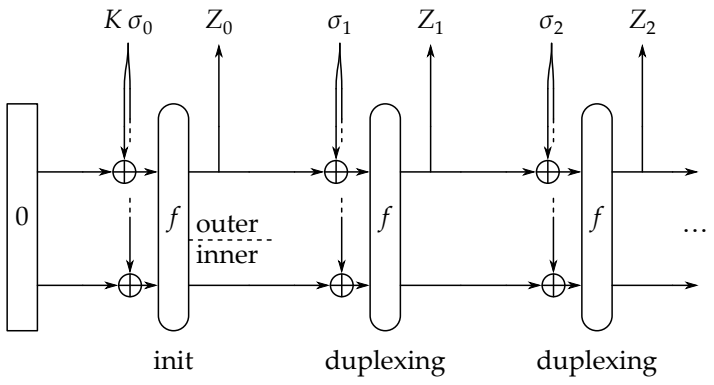

Figure 1: The full-state keyed duplex construction

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

3.2 Generic security of FSKD

The generic security of the FSKD construction was recently investigated by Mennink, Reyhanitabar and Vizár [11]. 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 budget of the adversary is composed of M, the data complexity (i.e., the total number of calls to the permutation f by the keyed object(s) under attack) and N the computational complexity (i.e., the total number of blocks queried to the permutation f or its inverse). The parameter $\mu \leq M$ is the total maximum multiplicity [1], whose value depends on the circumstances of an attack, as we discuss below.

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 [11]. By borrowing some techniques from [1], we can prove a more interesting bound. We state the new bound without proof in Theorem 1 below, and we will publish the proof in a separate paper.

For an adversary attacking one or more FSKD object(s), we define two kinds of multiplicities, namely the total maximum multiplicity μ and the maximum key multiplicity μ_K. They both depend on the set of queries that the adversary makes.

Definition 1 (Multiplicity)**.** *The* forward multiplicity *of a given outer value a, denoted by $\mu_{\text{fw}}(a)$, is the number of duplexing calls to FSKD where the outer part of the state is a before calling f. The* backward multiplicity *$\mu_{\text{bw}}(a)$ is the number of both duplexing and init calls to FSKD where the outer part of the state is a after calling f. The total maximum multiplicity μ is*

$$\mu = \max_a \mu_{\text{fw}}(a) + \max_a \mu_{\text{bw}}(a) \ .$$

The key multiplicity *of a given string x, denoted by $\mu_K(x)$, is the number of different init calls to FSKD with $\sigma_0 = x$. The maximum key multiplicity μ_K is*

$$\mu_K = \max_x \mu_K(x) \ .$$

Theorem 1 (Improved FSKD bound)**.** *The advantage of distinguishing FSKD$[f, r]$, initialized with a key K uniformly distributed over $\mathbb{Z}_2^{|K|}$ for $|K| \leq c$, from a random oracle, with f a random permutation, b its width, c the capacity, and $r = b - c$ the rate, is upper bounded by:*

$$(1 + \epsilon)\frac{M^2}{2^c} + \frac{\mu N}{2^c} + \frac{\mu_K N}{2^{|K|}} \ ,$$

with $\epsilon = 2^{-(r-2)} + 2^{-\min(|K|, r)}$ and where M is the data complexity (i.e., the total number of blocks fed to the full-state keyed duplex object), N the computational complexity (i.e., the total number of blocks queried to the permutation or its inverse), μ the total maximum multiplicity and μ_k the maximum key multiplicity.

In the typical case of $M < 2^{r/2}$, nonce-respecting adversaries will be confronted with a total maximum multiplicity $\mu = 2$ with overwhelming probability. An adversary that violates the nonce requirement and gets to start q sessions can increase the multiplicity to $q/2$.

For an attack targeting a single key, the maximum key multiplicity equals 1. For attacks targeting multiple keys, it is upper bound by that number of keys. By imposing that σ_0 is a global nonce, one can limit the maximum key multiplicity to 1 also for multi-target attacks. Note that if σ_0 can be imposed as global nonce, Theorem 1 allows taking a key as short as the security strength.

Note that if for Motorist the SUV consists of the key followed by a nonce, together fitting in a single input block, σ_0 is synonymous to the nonce. Otherwise, application of Theorem 1 requires making the distinction between the first part of SUV and the remaining part. We conjecture that our proof for can be adapted to cover multi-block SUV and

yielding the same bound, and with μ_K defined in terms of the full SUV rather than just its part injected in the first block. For simplicity, we will assume a single-input-block SUV when discussing maximum key multiplicity.

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 unambuguously recovered from the input block sequences to its Piston objects.*

Proof. (sketch) By the finite state machine implemented in its PHASE, 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 as hard to distinguish from random strings as indicated in Theorem 1. This covers privacy.

For authenticity, 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 sponge 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, the chaining values have length $c' \geq c$ and they are FSKD outputs. A collision in such a chaining value can have two causes. Either there is an inner collision in FSKD. This would constitute however distinguishing it from a random oracle and is hence covered by the bound in Theorem 1. Or there is no inner collision but just an output collision. The probability of that happening is upper bounded by $q^2/2^{c+1}$, with q the total number of sessions started.

This gives the following bounds:

Theorem 2. *The authenticated encryption mode Motorist defined in Section 1.6 satisfies*

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

against any single adversary $\mathcal{A}$ if $K \xleftarrow{\$} \mathbb{Z}_2^{|K|}$, f is a randomly chosen permutation and with

- *M: the total number of calls to f by Motorist due to queries of the adversary;*
- *q: the total number of calls to* Motorist.StartEngine() *by the adversary;*
- *N: the total number of direct queries to f or its inverse by the adversary;*
- *$S \leq M$: the total number of forged tags the adversary submits;*
- *μ: the total maximum multiplicity;*
- *μ_k: the maximum key multiplicity.*

3.5 Security of Keyak

For the security of Keyak against generic attacks, we can simply apply Theorem 2. Note that for the estimation of the maximum key multiplicity μ_K, 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. Note that the generic security strength achieved for the named Keyak versions is higher than the 128 bits in the Keyak security claims (see Section 2.5) for nonce-respecting adversaries. Due to the fact that in any real-world attack we have $M < 2^{r/2}$, nonce-respecting adversaries will be confronted with a total maximum multiplicity $\mu = 2$ with overwhelming probability. So against nonce-respecting adversaries one can even attain 256 bits of security, assuming reasonable data complexity.

As for non-generic attacks, we believe that the permutations Keccak-$p[1600, n_r = 12]$ and Keccak-$p[800, n_r = 12]$ 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 [7, 10].

The most powerful attacks on modes using Keccak-p are the cube attacks in [8, 9]. The application of full-state absorbing gives the attacker more degrees of freedom than the ones exploited in those papers. We have studied these attacks to see whether they can be improved for Keyak and they can. Our preliminary findings are the following:

- A full state recovery attack on all named Keyak instances with the number of rounds reduced to 6. It requires $Y = 2^{20}$ sessions with a chosen (first of a session) message and identical nonces to recover 12 state bits. Obtaining the full inner part of the state requires repeating this 128 times (or 64 for River Keyak) and some additional queries, so it can be done in about 2^{27} single-message sessions.
- Similar attack on 7 rounds. It takes $Y = 2^{40}$ sessions to recover 24 state bits, requiring a total of about 2^{46} sessions.

Closer investigation may reveal that we can reduce the number of required sessions Y due to the fact that less bits have to be guessed for controlling the propagation of our structure. It is not clear to us whether the attack extends to 8 rounds. If so, it would require 2^{85} sessions and hence would pose no practical threat. Moreover, the probability that less bits must be guessed decreases with the number of rounds. We believe the 12 rounds we took still provide a comfortable safety margin. We plan to publish our attacks soon in a separate paper.

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.

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

[8] 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/`.

[9] ———, *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.

[10] 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/.

[11] 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/.

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 $\frac{S}{2^r}$, 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.