CAESAR submission: Ketje v1

Designed and submitted by:

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

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

Version **1.1**
March 25, 2014

[1]STMicroelectronics
[2]NXP Semiconductors

Contents

KETJE is a set of two authenticated encryption functions with support for message associated data. They are aimed at memory-constrained devices and strongly rely on nonce uniqueness for security. In the architecture of our proposal we adopt a layered approach, with KETJE specified as instantiations of a mode built on top of a construction that calls a permutation as cryptographic primitive.

KETJE builds on round-reduced versions of KECCAK-$f[400]$ and KECCAK-$f[200]$ [4]. The construction calling these permutations is MONKEYDUPLEX, a variant of the duplex construction [3]. Its most important new feature is that it supports different types of calls that invoke the permutation with a different number of rounds. The performance of the resulting scheme can be optimized by reducing the number of rounds quite aggressively, at the cost of requiring nonce uniqueness for its security against key retrieval. This restricts MONKEYDUPLEX to use cases where nonce uniqueness can be guaranteed. This includes scenarios where replay attacks are a concern.

The mode that runs on top of MONKEYDUPLEX is called MONKEYWRAP, similar and functionally equivalent to SPONGEWRAP [3]. The main differences with the latter are that it is built on top of MONKEYDUPLEX instead of duplex and that it uses two bits per block for domain separation instead of a single one.

KETJE's rate is set to only 8 % of the permutation width in order to achieve state-of-the-art security strength despite the relatively small state size of 200 (and 400) bits. This is compensated by performing only a single round of KECCAK-f in the majority of calls. In this respect, KETJE is somewhat similar to RADIOGATÚN [1].

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

1 Definitions

1.1 Notation

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

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

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

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

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

1.2 Of bits and bytes

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

$(0,1,1,0,0,1,0,1,0,0,1,1,1,1,1,1)$ can also be written as the sequence $(0,1,1,0,0,1,0,1)$ $(0,0,1,1,1,1,1,1)$ or `0xA6` `0xFC`.

1.3 Padding rules

We use two different padding rules:

- The *simple padding*, denoted $\text{pad10}^*[r](|M|)$, returns a bit string 10^q with $q = (-|M| - 1) \bmod r$. When r is divisible by 8 and M is a sequence of bytes, then $\text{pad10}^*[r](|M|)$ returns the byte string `0x01` `0x00`$^{(q-7)/8}$.
- The *multi-rate padding*, denoted $\text{pad10}^*1[r](|M|)$, returns a bitstring 10^q1 with $q = (-|M| - 2) \bmod r$ [3]. When r is divisible by 8 and M is a sequence of bytes, then $\text{pad10}^*1[r](|M|)$ returns the byte string `0x01` `0x00`$^{(q-14)/8}$ `0x80`.

1.4 Key pack

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

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

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

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

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

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

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

2 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-$p[b, n_\text{r}]$ consists in the application of the *last* n_r rounds of Keccak-$f[b]$. When $n_\text{r} = 12 + 2\ell$, Keccak-$p[b, n_\text{r}] =$ Keccak-$f[b]$.

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

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

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

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

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

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

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

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

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

3 The MONKEYDUPLEX construction

The MONKEYDUPLEX construction is a toolbox aimed at building stream ciphers and authenticated encryption schemes. It uses a permutation f with a tunable number of rounds. We denote the instance of f with n_r rounds $f[n_r]$. The MONKEYDUPLEX construction takes four parameters that determine its efficiency and security strength.

Similar to the duplex construction [3], the MONKEYDUPLEX is stateful and accepts calls taking a string as input and returning a string as output. This output string depends on all inputs received so far. Unlike duplex, MONKEYDUPLEX supports two types of calls that are different in the number of rounds of f executed between input and output.

We call an instance of the MONKEYDUPLEX construction a *MONKEYDUPLEX object* and denote it as D in our descriptions. We prefix the calls made to a specific MONKEYDUPLEX object D by its name D and a dot.

The MONKEYDUPLEX$[f, r, n_{\text{start}}, n_{\text{step}}, n_{\text{stride}}]$ construction works as follows:

- A MONKEYDUPLEX instance D has a state of b bits, where b is the width of the underlying permutation. A MONKEYDUPLEX instance can be started with a call D.start(I), where the string I can be almost full width. This initializes the state by setting it to the input string I, extended to b bits with multi-rate padding. Subsequently, it applies $f[n_{\text{start}}]$ to it.

- With calls to D.step(σ, ℓ) and D.stride(σ, ℓ) one can inject a bit string σ of up to $r - 2$ bits. After the bits are injected, either $f[n_{\text{step}}]$ or $f[n_{\text{stride}}]$ is applied to the state and the first ℓ bits of the state are extracted, with $\ell \leq r$. These interfaces are similar but serve different purposes. Both aim at providing resistance against state retrieval, but in addition, D.stride() also aims at providing resistance against output forgery. Hence this requires that $n_{\text{step}} < n_{\text{stride}}$. Typically we also have $n_{\text{stride}} < n_{\text{start}}$.

The MonkeyDuplex construction is illustrated in Figure 1.

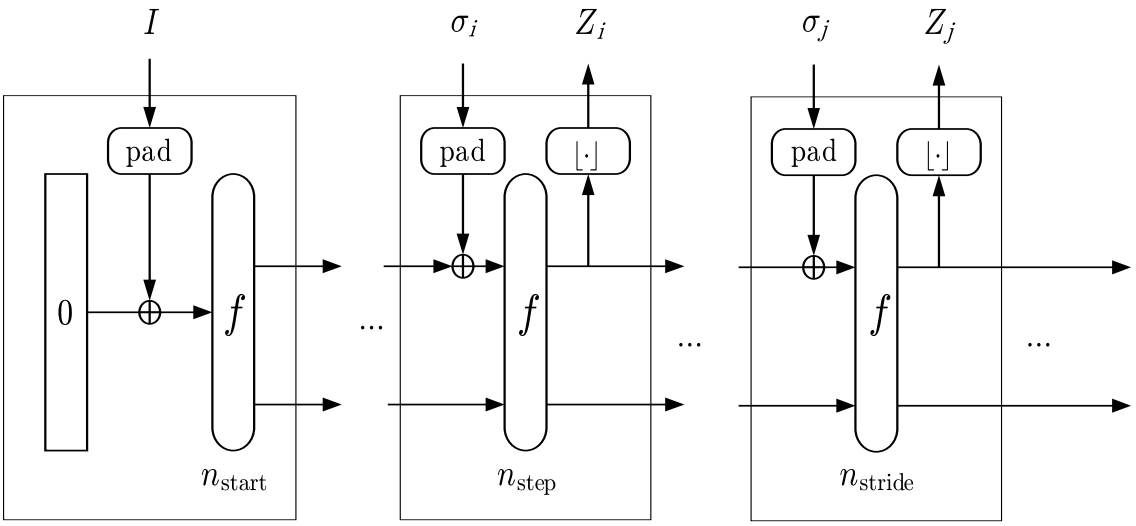

Figure 1: The MonkeyDuplex construction

We originally proposed the concept of the MonkeyDuplex construction in [5]. We slightly modified the definition since then due to new insights.

3.1 Specification

In this section we formally specify MonkeyDuplex with pseudo-code in Algorithm 1.

3.2 Rationale

MonkeyDuplex is meant to be used in a keyed mode. During its start-up it shall be loaded with I containing a secret key and a nonce and during operation an attacker shall not have access to the inner state.

The values of r, n_{start}, n_{step} and n_{stride} are meant to be tuned to target some given security strength s, possibly assuming the data complexity is below some (large) value. We relate the security strength s to the complexity of state reconstruction, to the highest differential probability (DP) of differentials over the permutation $f[n_{\text{start}}]$, and the highest DP of differentials between MonkeyDuplex input and output across stride calls. We now list three informal security claims that express the criteria underlying the choice of r, n_{start}, n_{step} and n_{stride}.

Claim 1 (Solitary state retrieval hardness)**.** *For an attacker that can adaptively make D.step() and D.stride() calls to a single MonkeyDuplex instance with unknown inner state, there shall be no algorithm that succeeds in reconstructing its inner state with success probability above $N2^{-s}$.*

Algorithm 1 The MonkeyDuplex$[f, r, n_{\text{start}}, n_{\text{step}}, n_{\text{stride}}]$ construction

Require: $2 < r < b$, $n_{\text{step}} < n_{\text{stride}}$
Require: $s \in \mathbb{Z}_2^b$ (maintained across calls)

Interface: D.start(I) with $I \in \mathbb{Z}_2^{\leq b-2}$
$s = I||\text{pad10}^*1[b](|I|)$
$s = f[n_{\text{start}}](s)$

Interface: $Z = D$.step(σ, ℓ) with $\ell \leq r$, $\sigma \in \mathbb{Z}_2^{\leq r-2}$ and $Z \in \mathbb{Z}_2^{\ell}$
$P = \sigma||\text{pad10}^*1[r](|\sigma|)$
$s = s \oplus (P||0^{b-r})$
$s = f[n_{\text{step}}](s)$
return $\lfloor s \rfloor_{\ell}$

Interface: $Z = D$.stride(σ, ℓ) with $\ell \leq r$, $\sigma \in \mathbb{Z}_2^{\leq r-2}$ and $Z \in \mathbb{Z}_2^{\ell}$
$P = \sigma||\text{pad10}^*1[r](|\sigma|)$
$s = s \oplus (P||0^{b-r})$
$s = f[n_{\text{stride}}](s)$
return $\lfloor s \rfloor_{\ell}$

Here N is the computational complexity of the attack, where a computational effort equal to that of a single call to $f[n_{step}]$ is the unit and where the computation of the calls to the MonkeyDuplex instance under attack is included.

This claim simply expresses that it shall be hard to retrieve the state for an attacker who is limited to a single instance. By single-instance we understand that the attacker does not have access to multiple MonkeyDuplex instances started with the same input I. The single-instance limitation practically excludes the use of differential cryptanalysis for solitary state retrieval.

The requirement of solitary state retrieval hardness affects the choice of r and n_{step}. Two classical approaches to reconstruct the inner state are:

- State guessing: we take a MonkeyDuplex object, assume a value for the inner state, apply the known input sequence and check whether the simulated output sequence corresponds with that of the instance under attack. We repeat this procedure until we have success. The success probability depends on a feature of the available input-output sequence called the *multiplicity* m [2] that is limited by the data complexity of the attack. The success probability is slightly above $mN2^{r-b} = mN2^{-c}$. This implies we have to take $s < c - \log_2 m$. Note that this is a generic method.

- Equation solving: we express the round function as a set of *round equations*. We use these to form a set of equations in the inner state with known inputs and outputs covering a number of MonkeyDuplex calls. The known outputs and inputs should be large enough to give a unique solution for the inner state. Then we try to solve this set of equations. The success probability of this attack as a function of its computational complexity is hard to estimate and strongly depends on the method used and the nature of the round function of the cipher. However, it is clear that the number of equations and unknowns is proportional to the total number of rounds covered and the capacity c. This is a non-generic method.

In general, for a given round function f with width b, the designer has some freedom in the selection of the parameters r and n_{step}. Increasing the rate r reduces the resistance with respect to both attacks. In state guessing, it decreases the number $b - r$ of unknown state bits to be guessed. In equation solving, for equal n_{step} value, it reduces the total number of round equations and unknowns. Similarly, decreasing n_{step} also reduces the total number of equations and unknowns. In this context we can derive from n_{step}, the width b and the rate r a quantity that characterizes the difficulty of equation solving: the *unicity number*.

Definition 1. *In a mode calling an iterative primitive f, the unicity number $n_{unicity}$ is the number of rounds of f that separate the first and last bits of a sequence of output bits long enough to fully determine its internal state.*

The unicity number of MonkeyDuplex is given by:

$$n_{\text{unicity}} = \left\lceil \frac{b-r}{r} \right\rceil n_{\text{step}} \ .$$

This formula can be explained as follows. The size of the state is b bits so we need b output bits. After seeing r bits of output from a call to D.step() or D.stride(), we need $b - r$ more bits of output. This takes $\left\lceil \frac{b-r}{r} \right\rceil$ additional calls to D.step(), each one taking n_{step} rounds (note that $n_{\text{step}} < n_{\text{stride}}$) .

The security grows and efficiency decreases with increasing n_{unicity}. Informally, n_{unicity} is the number of rounds covered by any system of equations that can be used to recover the state. The idea is for a given round function to estimate a safe value of n_{unicity}, and use that to determine n_{step} and the rate r. We give an evaluation of n_{unicity} for our submission in Section 5.3.

Our second claim concerns resistance against output forgery. A typical call sequence in that case is D.start(I), D.step(), ..., D.step(), followed by D.stride(), D.step(), ... We will bound the resistance against an attacker trying to manipulate the output after the call to D.stride(). In our description, we index this call with 0, previous calls with negative numbers and further calls with positive numbers. We implicitly exclude the case where the attacker could extract the state before this call, since otherwise she would succeed with probability 1.

Claim 2 (Solitary output forgery hardness)**.** *Consider an attacker that is given the inputs σ_i and outputs Z_i with $-m < i < n$ of any sequence of calls D.step() and D.stride() of a single MonkeyDuplex instance with unknown inner state and let the call with index $i = 0$ be a call to D.stride(). For such an attacker the success probability per attempt of any algorithm that succeeds in constructing another input sequence σ'_i and a corresponding partial output sequence Z'_j for $0 \leq j < n$ that would be consistent with the given instance shall be below* $\max(2^{-rn}, 2^{-s})$.

This claim expresses that modifying the input to a single instance impacts its output in a way that makes it hard to predict. In fact, the success probability of correctly predicting from a difference applied at the input what is the difference at the output after a call to D.stride() should not be better than pure chance, and this up to a length that matches the security strength s. Output forgery hardness affects the choice of r and n_{stride}. It induces a cost whenever protection against output forgery is a requirement. This is the case for tags in authenticated encryption.

An approach to make a forgery would be to find a differential of type $(\sigma'_{-m}, \ldots, \sigma'_0)$ to $(Z'_0, \ldots, Z'_{n-1})$ with DP above $\max(2^{-rn}, 2^{-s})$. Subsequently, the attacker just modifies the observed input sequence by the adding σ'_i and assumes that the forged output is the observed output sequence with Z'_j added to it.

Claim 3 (Strong instance separation)**.** *An attacker that can start multiple* MonkeyDuplex *instances, with the only restriction that no two objects may have the same value I in D.*start(I)*, but possibly the same key, must not have an advantage in performing state retrieval or output forgery over the cases of solitary state retrieval or solitary output forgery.*

Strong instance separation affects the choice of n_{start}. Applying $f[n_{\text{start}}]$ to I shall destroy all structures an attacker can apply in choosing I that would allow her to attack multiple instances better than single ones. The value of n_{start} basically determines the fixed set-up cost for starting a new MonkeyDuplex object and as such strongly affects key agility.

Note that no security claims are made against an attacker that can start multiple MonkeyDuplex instances with the same input I. As a matter of fact, this may allow an attacker to retrieve the inner state, fully breaking it. Hence it is essential that I is unique for each use of MonkeyDuplex. Or, stated otherwise, the uniqueness of I is as important as its secrecy.

In short, an attacker that can start multiple MonkeyDuplex instances with the same input I can inject differences in the input to calls to D.step$(-, r)$ and observe differences in its response. There are only n_{step} rounds between input and output and this typically is a very small number. The combination of input and output differences impose restrictions on bits of the inner part of the state in the form of simple equations and an attacker may collect enough of such equations to reconstruct the full state.

3.3 Applications

The MonkeyDuplex construction can be used in several use cases, typically implemented as a mode on top of the MonkeyDuplex construction. Here we give an informal discussion on the three most important use cases we have in mind.

The first and simplest use case is that of a synchronous stream cipher. The key and nonce are concatenated to form I and a new instance can be started by D.start(I). From then on, key stream can be generated ad libitum by making D.step$(-, r)$ calls. Output forgery does not apply and hence no D.stride$(-, r)$ is needed.

The second use case is that of a reseedable pseudorandom bit sequence generator [2]. Initial seed material is concatenated with a nonce to form I and a new instance can be started by D.start(I). From then on pseudorandom bits can be generated ad libitum by making D.step(σ, r) calls, where σ may contain fresh seed material. Also here, typical use cases do not require protection against output forgery and hence no D.stride$(-, r)$ is needed.

The third use case, and the most relevant one in this document, is that of authenticated encryption. The key and nonce are concatenated to form I and a new instance can be started by D.start(I). From then on, messages with associated data can be wrapped or cryptograms with associated data and tags can be unwrapped. Encryption is done by bitwise addition with the output of D.step$()$ calls. The inner state depends on all the messages and associated data presented to the MonkeyDuplex instance since it was started and so will any output of a call to D.step$()$ or D.stride$()$. The (first part of the) tag is the output of a D.stride$()$ call, providing protection against tag forgery. Actually, this tag forgery concern is the reason we introduced the D.stride$()$ call in MonkeyDuplex, compared to the our draft proposal of MonkeyDuplex in [5].

4 The authenticated encryption mode MonkeyWrap

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

We further consider the process of authenticating and encrypting a sequence of header-body pairs $(\overline{A, B}) = (A^{(1)}, B^{(1)}, A^{(2)}, \ldots, A^{(n)}, B^{(n)})$ in such a way that the authenticity is guaranteed not only on each (A, B) pair but also on the sequence received so far. This is further formalized in [3, Section 2.1].

The authenticated encryption process is initialized by loading a key K and a nonce N.

We propose an authenticated encryption mode MonkeyWrap that is very similar to SpongeWrap [3]. The differences with SpongeWrap are the following:

1. MonkeyWrap is built on MonkeyDuplex rather than on duplex.
2. MonkeyWrap has a simpler way to load the key and a nonce in the initialization. It relies on uniqueness of the combination key and nonce for resistance against key retrieval.
3. MonkeyWrap makes different calls to MonkeyDuplex when transitioning to tag generation than in other cases.

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

The maximum key length of MonkeyWrap is only limited by the width of the underlying permutation and the coding of the key pack. Note that using a key of length $|K|$ does necessarily imply that the security strength of the instance is $|K|$.

4.1 Specification

MonkeyWrap is defined in Algorithm 2 and illustrated in Figure 2. For simplifying notation, we restrict the length of the key K to multiples of 8. In the algorithms we denote by A_i the block consisting of bits ρi to $\rho(i+1) - 1$ of A. The quantity ρ can be seen as the block length of the mode. The blocks of length up to ρ bits map to blocks of $r = \rho + 4$ inside MonkeyDuplex by the addition of two domain separation bits and subsequent application of multi-rate padding.

The number of blocks in A is denoted by $\|A\|$. All blocks of A have ρ bits except the last one, $A_{\|A\|-1}$. This one may have less bits but must be non-empty if A is not the empty string. If A is the empty string, it has a single block A_0 that is also the empty string. The same holds for B, C and T.

4.2 Rationale

After initialization, MonkeyWrap appends two frame bits to each input block providing domain separation resulting in protection against generic attacks. Its generic security is similar to the one of SpongeWrap [3].

Algorithm 2 The MonkeyWrap$(f, \rho, n_{\text{start}}, n_{\text{step}}, n_{\text{stride}})$ construction.

Require: $0 < \rho \leq b - 4$
Require: $D = \text{MonkeyDuplex}[f, \rho + 4, n_{\text{start}}, n_{\text{step}}, n_{\text{stride}}]$

Interface: W.initialize(K, N) with $K \in \mathbb{Z}_2^{\leq b-18}$, $|K| \bmod 8 = 0$ and $N \in \mathbb{Z}_2^{\leq b-|K|-18}$,
D.start(keypack$(K, |K| + 16)||N$)

Interface: $(C, T) = W$.wrap(A, B, ℓ) with $A, B, C \in \mathbb{Z}_2^*$, $\ell \geq 0$, and $T \in \mathbb{Z}_2^\ell$
for $i = 0$ to $\|A\| - 2$ **do**
  D.step$(A_i||00, 0)$
$Z = D$.step$(A_{\|A\|-1}||01, |B_0|)$
$C_0 = B_0 \oplus Z$
for $i = 0$ to $\|B\| - 2$ **do**
  $Z = D$.step$(B_i||11, |B_{i+1}|)$
  $C_{i+1} = B_{i+1} \oplus Z$
$T = D$.stride$(B_{\|B\|-1}||10, \rho)$
while $|T| < \ell$ **do**
  $T = T||D$.step$(0, \rho)$
$T = \lfloor T \rfloor_\ell$
return (C, T)

Interface: $B = W$.unwrap(A, C, T) with $A, B, C, T \in \mathbb{Z}_2^*$
for $i = 0$ to $\|A\| - 2$ **do**
  D.step$(A_i||00, 0)$
$Z = D$.step$(A_{\|A\|-1}||01, |B_0|)$
$B_0 = C_0 \oplus Z$
for $i = 0$ to $\|C\| - 2$ **do**
  $Z = D$.step$(B_i||11, |C_{i+1}|)$
  $B_{i+1} = C_{i+1} \oplus Z$
$T' = D$.stride$(B_{\|C\|-1}||10, \rho)$
while $|T'| < |T|$ **do**
  $T' = T'||D$.step$(0, \rho)$
$T' = \lfloor T \rfloor_{|T|}$
if $T = T'$ **then**
  return B
else
  return error

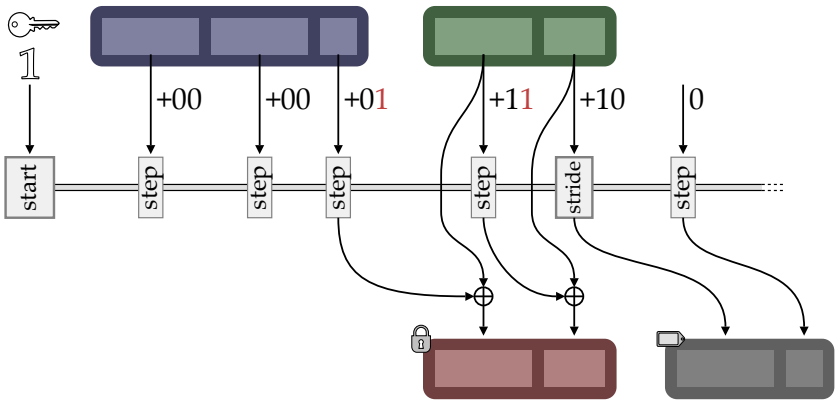

Figure 2: Wrapping a header and a body with MonkeyWrap

	KETJE JR	KETJE SR
plaintext confidentiality	$\min(96, \|K\|)$	$\min(128, \|K\|)$
plaintext integrity	$\min(96, \|K\|, \|T\|)$	$\min(128, \|K\|, \|T\|)$
associated data integrity	$\min(96, \|K\|, \|T\|)$	$\min(128, \|K\|, \|T\|)$
public message number integrity	$\min(96, \|K\|, \|T\|)$	$\min(128, \|K\|, \|T\|)$

Table 1: Security claims for KETJE. For KETJE JR we assume a maximum data complexity below 2^{87} bytes. The key length $|K|$ assumes the keys follow a uniform distribution. If not, $|K|$ shall be interpreted as the min-entropy of the key.

5 KETJE

In this section we specify KETJE, our submission to CAESAR and give a rationale for the choice of parameters.

5.1 Specification

We propose two concrete instances of MONKEYWRAP calling KECCAK-p. Our primary recommendation is called **KETJE SR**:

$$\text{KETJE SR} = \text{MONKEYWRAP}(\text{KECCAK-}p[400], \rho = 32, n_{\text{start}} = 12, n_{\text{step}} = 1, n_{\text{stride}} = 6)$$

KETJE SR supports keys K of variable length up to 382 bits and a nonce of length up to $382 - |K|$. For the targeted security strength we recommend a key length of 128 bits. Higher key lengths can be adopted as a possible countermeasure against multi-target attacks.

Our secondary recommendation is called **KETJE JR**:

$$\text{KETJE JR} = \text{MONKEYWRAP}(\text{KECCAK-}p[200], \rho = 16, n_{\text{start}} = 12, n_{\text{step}} = 1, n_{\text{stride}} = 6)$$

KETJE JR supports keys K of length up to 182 bits and a nonce of length up to $182 - |K|$. For the targeted security strength we recommend a key length of 96 bits. Higher key lengths can be adopted as a possible countermeasure against multi-target attacks.

For both proposals we remind the requirement of uniqueness of the combination of key and nonce. In case of maximum-length keys the nonce has length 0 and the key alone shall be unique.

In general, a KETJE object supports multiple calls to wrap or unwrap per initialization and the length of the tag can simply be adapted to the required tag forgery level of the target application.

5.2 Security goals

Table 1 specifies the security goals for KETJE. The security strength is indicated with the logarithm base 2 of the attack cost, where the unit is a single KECCAK-p round. For KETJE JR we assume the total data complexity is below 2^{87} bytes.

Users are required to use the public message number N as a nonce, i.e., the cipher may lose all integrity and confidentiality if the legitimate key holder uses the same public message number N to encrypt two different (plaintext, associated data) pairs under the same key K. The uniqueness of the nonce N is as critical for security as the secrecy of K.

In multi-target attacks against KETJE 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 KETJE 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 (or 96) bits by taking sufficiently long keys: $|K| \geq 128 + \log_2 n_{\max}$ (or $|K| \geq 96 + \log_2 n_{\max}$). An option that avoids erosion without increasing the length of keys is to impose universal nonce uniqueness. By this we mean that not only the combination (K, N) must be unique, but the nonce N for each KETJE instance must be unique. Many use cases actually allow this. For example, one can take as nonce the combination of the universally unique IDs of the two communicating devices and a strictly incrementing session counter.

5.3 Rationale

First of all, we chose KECCAK-p as the underlying permutations as we consider this a test case for KECCAK.

The basic philosophy behind the two KETJE proposals is to maximize the capacity by taking a small rate and compensate the loss of performance by reducing the number of KECCAK-p rounds in the step calls to a single one, i.e., $n_{\text{step}} = 1$. This has the advantage that the success probability of solitary state retrieval by state guessing is minimized. The expected workload of this generic attack is lower bounded by $2^{c-1}/M$ with M the total amount of data treated by the keyed KETJE instance(s) under attack [2].

We chose the concrete value of the rate by assuming a reasonable value for n_{unicity}. As a matter of fact, basing ourselves on third-party analysis of KECCAK and our own, we estimate that $n_{\text{unicity}} = 12$ gives a comfortable safety margin against solitary state retrieval with equation solving. This leads to $r \leq b/12$, matching nicely with the size of two lanes. So we took $\rho = 32$ in KETJE SR and $\rho = 16$ in KETJE JR.

We chose the number of rounds in a stride call, n_{stride}, equal to 6. In this decision we took into account the small rate. For an adversary that does not know the value of the inner state we think it will be infeasible to come up with an input difference pattern that will lead to an output difference after 6 KECCAK-p rounds with a differential probability (DP) that will break the solitary output forgery hardness.

We chose the number of rounds in a start call, n_{start}, equal to 12. Based on third-party cryptanalysis of KECCAK and our own, we believe this provides a comfortable safety margin against attacks applying differences or other structures at its input to have exploitable relations at its output.

Finally, we chose the security strength of KETJE SR to be 128 bits as this is a widespread standard and future-proof. In KETJE JR the width of the permutation is only 200 bits and aiming for a security strength of 128 bits would impose too much limitations on the online data complexity and limit the length of the nonce too much to our taste. So we reduced the security strength to 96 bits. Despite the fact that this is significantly below 128 bits, we believe attacks with computational complexity equivalent to 2^{96} calls to the KECCAK-$p[200, n_{\text{r}}]$ round function will remain overly expensive to deploy in practice for several decades to come.

6 Using KETJE in the context of CAESAR

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

6.1 Specification and security goals

In the context of CAESAR, performing an authenticated encryption with KETJE on a message M, associated data AD using a public message number N and a key K is done as follows. Create a KETJE object W and initialize it with the key and public message number. Subsequently wrap the associated data and message, asking for a tag with length equal to the target security strength s. This is W.initialize(K, N) followed by W.wrap(AD, M, s). The secret message number has length 0.

The security goals of KETJE are specified in Section 5.2.

6.2 Security analysis and design rationale

For the security analysis and design rationale of KETJE and its building blocks we refer to the sections that explain the rationale behind them: Section 5.3 for KETJE, Section 4.2 for MONKEYWRAP, and Section 3.2 for MONKEYDUPLEX.

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 KETJE JR and KETJE SR the capacity is $c = 180$ and $c = 364$, respectively.

KETJE has the following security assurance features:

- Generic security of the mode MONKEYWRAP.
- Security assurance from cryptanalysis of KECCAK. Note that thanks to the Matryoshka property, most analysis performed on versions of KECCAK-f transfers to those with smaller widths.

The designers have not hidden any weaknesses in this cipher or any of its components. We believe this to be impossible:

- KECCAK-f and its round-reduced versions: all design choices are documented and explained in [4]
- MONKEYDUPLEX: a rationale is given in Section 3.2.
- MONKEYWRAP: a rationale is given in Section 4.2.
- KETJE: a rationale is given in Section 5.3.

6.3 Features

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

- KETJE is lightweight in the sense that it has a small code and working memory footprint and requires a relatively small amount of computation, as illustrated in Table 2.
- The implementation of the round function can be re-used for other symmetric cryptographic primitives, such as hashing, which further reduces the footprint compared to a solution with distinct primitives.
- KETJE lends itself well to protections against side channel attacks, both in hardware and software. This is of particular importance in the context of constrained devices and smart cards.

- As a functional feature not present in most authenticated ciphers, KETJE 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.

A typical application of KETJE would be the so-called secure messaging with secure chips such as smart cards [6]. The session feature offers an easy and agile way to send sequences of commands as scripts, interactively or by batch, while preventing an attacker to insert, remove or swap commands in the script.

6.4 Intellectual property

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

6.5 Consent

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

feature		**KETJE JR**	**KETJE SR**
round function		KECCAK-$p[200, n_r]$	KECCAK-$p[400, n_r]$
state size		25 bytes	50 bytes
block size		2 bytes	4 bytes
processing	**unit**	**computational cost**	
initialization	per session	12 rounds	12 rounds
wrapping	per block	1 round	1 round
8-byte tag computation	per message	$6 + 3 \times 1 = 9$ rounds	$6 + 1 \times 1 = 7$ rounds

Table 2: KETJE lightweight implementation features

References

[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, RadioGatún, *a belt-and-mill hash function*, Second Cryptographic Hash Workshop, Santa Barbara, August 2006, `http://radiogatun.noekeon.org/`.

[2] ———, *Sponge-based pseudo-random number generators*, CHES (S. Mangard and F.-X. Standaert, eds.), Lecture Notes in Computer Science, vol. 6225, Springer, 2010, pp. 33–47.

[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] ———, *Permutation-based encryption, authentication and authenticated encryption*, Directions in Authenticated Ciphers, July 2012.

[6] ISO/IEC, *Identification cards – integrated circuit cards – part 4: Organization, security and commands for interchange*, 2005.

A Change log

A.1 From 1.0 to 1.1

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