Xoodyak, a lightweight cryptographic scheme

Joan Daemen[2], Seth Hoffert, Michaël Peeters[1], Gilles Van Assche[1] and Ronny Van Keer[1]

[1] STMicroelectronics
[2] Radboud University

Abstract. In this paper, we present Xoodyak, a cryptographic primitive that can be used for hashing, encryption, MAC computation and authenticated encryption. Essentially, it is a duplex object extended with an interface that allows absorbing strings of arbitrary length, their encryption and squeezing output of arbitrary length. It inherently hashes the history of all operations in its state, allowing to derive its resistance against generic attacks from that of the full-state keyed duplex. Internally, it uses the Xoodoo[12] permutation that, with its width of 48 bytes, allows for very compact implementations. The choice of 12 rounds justifies a security claim in the hermetic philosophy: It implies that there are no shortcut attacks with higher success probability than generic attacks. The claimed security strength is 128 bits. We illustrate the versatility of Xoodyak by describing a number of use cases, including the ones requested by NIST in the lightweight competition. For those use cases, we translate the relatively detailed security claim that we make for Xoodyak into simple ones.

Version of Xoodyak: v1

Version of this document: v1.1 (March 29, 2019)

Keywords: lightweight cryptography · permutation-based cryptography · sponge construction · duplex construction · authenticated encryption · hashing

1 Introduction

Xoodyak is a versatile cryptographic object that is suitable for most symmetric-key functions, including hashing, pseudo-random bit generation, authentication, encryption and authenticated encryption. It is based on the duplex construction, and in particular on its full-state (FSKD) variant when it is fed with a secret key [3, 12]. It is stateful and shares features with Markku Saarinen's Blinker [19], Mike Hamburg's Strobe protocol framework [13] and Trevor Perrin's Stateful Hash Objects (SHO) [17]. In practice, Xoodyak is straightforward to use and its implementation can be shared for many different use cases.

Internally, Xoodyak makes use of the Xoodoo permutation [11, 10]. The design approach of this 384-bit permutation is inspired by Keccak-*p* [5, 15], while it is dimensioned like Gimli for efficiency on low-end processors [1]. The structure consists of three planes of 128 bits each, which interact per 3-bit columns through mixing and nonlinear operations, and which otherwise move as three independent rigid objects. Its round function lends itself nicely to low-end 32-bit processors as well as to compact dedicated hardware.

The mode of operation on top of Xoodoo is called *Cyclist*, as a lightweight counterpart to Keyak's Motorist mode [6]. It is simpler than Motorist, mainly thanks to the absence of parallel variants. Another important difference is that Cyclist is not limited to authenticated encryption, but rather offers fine-grained services, à la Strobe, and supports hashing.

Xoodyak contains several built-in mechanisms that help protect against side-channel attacks.

- Following an idea by Taha and Schaumont [23], Cyclist can absorb the session counter that serves as nonce in chunks of a few bits. This counters differential power analysis (DPA) by limiting the degrees of freedom of an attacker when exploiting a selection function, see Section 3.2.2.
- Another mechanism consists in replacing the incrementation of a counter with a key derivation mechanism: After using a secret key, a derived key is produced and saved for the next invocation of Xoodyak. The key then becomes a moving target for the attacker, see Section 3.2.6.
- Then, to mitigate the impact of recovering the internal state, e.g., after a side channel attack, the Cyclist mode offers a ratchet mechanism, similar to the "forget" call in [3]. This mechanism offers forward secrecy and prevents the attacker from recovering the secret key prior to the application of the ratchet, see Section 3.2.5.
- Finally, the Xoodoo round function lends itself to efficient masking countermeasures against differential power analysis and similar attacks.

1.1 Notation

The set of all bit strings is denoted $\mathbb{Z}_2^*$ and ϵ is the empty string. Xoodyak works with bytes and in the sequel we assume that all strings have a length that is multiple of 8 bits. The length in bytes of a string X is denoted $|X|$, which is equal to its bit length divided by 8.

We denote a sequence of m strings $X^{(0)}$ to $X^{(m-1)}$ as $X^{(m-1)} \circ \cdots \circ X^{(1)} \circ X^{(0)}$. The set of all sequences of strings is denoted $(\mathbb{Z}_2^*)^*$ and $\emptyset$ is the sequence containing no strings at all.

We denote with $\text{enc}_8(x)$ a byte whose value is the integer $x \in \mathbb{Z}_{256}$.

1.2 Usage overview

Xoodyak is a stateful object. It offers two modes: the hash and the keyed modes, one of which is selected upon initialization.

1.2.1 Hash mode

In hash mode, it can absorb input strings and squeeze digests at will. Absorb(X) absorbs an input string X, while Squeeze(ℓ) produces an ℓ-byte output depending on the data absorbed so far. The simplest case goes as follows:

```
Cyclist(ϵ, ϵ, ϵ) {initialization in hash mode}
Absorb(x) {absorb string x}
h ← Squeeze(n) {get n bytes of output}
```

Here, h gets a n-byte digest of x, where n can be chosen by the user. Xoodyak offers 128-bit security against any attack, unless easier on a random oracle. To get 128-bit collision resistance, we need to set $n \geq 32$ bytes (256 bits), while for a matching level of (second) preimage resistance, it is required to have $n \geq 16$ bytes (128 bits). This is similar to the SHAKE128 extendable output function (XOF) [15].

More complicated cases are possible, for instance:

CYCLIST$(\epsilon, \epsilon, \epsilon)$
ABSORB(x)
ABSORB(y)
$h_1 \leftarrow$ SQUEEZE(n_1)
ABSORB(z)
$h_2 \leftarrow$ SQUEEZE(n_2)

Here, h_1 is a digest over the two-string sequence $y \circ x$ and h_2 is a digest over $z \circ y \circ x$. The digest is over the sequence of strings and not just their concatenation. In this aspect, we here have a function that is similar to and has the same security level as TupleHash128 [16].

1.2.2 Keyed mode

In keyed mode, XOODYAK can do stream encryption, message authentication code (MAC) computation and authenticated encryption.

As a first example, the following sequence produces a tag (a.k.a. MAC) on a message M:

CYCLIST(K, ϵ, ϵ) {initialization in keyed mode with key K}
ABSORB(M) {absorb message M}
$T \leftarrow$ SQUEEZE(t) {get tag T}

The last line produces a t-byte tag, where t can be specified by the application. A typical tag length would be $t = 16$ bytes (128 bits).

Then, encryption is done in a stream cipher-like way, hence it requires a nonce. The obvious way to do encryption would be do call SQUEEZE() and use the output as a keystream. ENCRYPT(P) works similarly, but it also absorbs P block per block as it is being encrypted. This offers an advantage in case of nonce misuse, as the leakage is limited to one block when the two plaintexts start to differ. Hence, to encrypt plaintext P under a given nonce, we can run the following sequence:

CYCLIST(K, ϵ, ϵ)
ABSORB(nonce)
$C \leftarrow$ ENCRYPT(P) {get ciphertext C}

And to decrypt, we simply replace the last line with:

$P \leftarrow$ DECRYPT(C) {get plaintext P}

Finally, authenticated encryption can be achieved by combining the previous sequences. For instance, to encrypt plaintext P under a given nonce and associated data A, we proceed as follows:

CYCLIST(K, ϵ, ϵ)
ABSORB(nonce)
ABSORB(A) {absorb associated data A}
$C \leftarrow$ ENCRYPT(P)
$T \leftarrow$ SQUEEZE(t) {get tag T}

We attach a fairly precise, yet involved, security claim to XOODYAK in keyed mode. In addition, we provide clear corollaries with the resulting security strength for specific use cases. Here are two examples, which both assume a single secret key made of $\kappa = 128$ uniformly and independently distributed bits.

- First, let us take the MAC computation at the beginning of this section. It does not enforce the use of a nonce, hence an adversary gets more power in exploiting adaptive queries. Yet, this authentication scheme can resist against an adversary with up to 2^{128} computational complexity and up to 2^{64} data complexity (measured in blocks).
- Then, we discuss the last example of this section, namely the authenticated encryption scheme. We assume an application that correctly implements nonces and that does not release unverified decrypted ciphertexts. The use of nonces makes Xoodyak resist against even stronger adversaries. Our claim implies that this nonce-based authenticated encryption scheme can resist against an adversary with up to 2^{128} computational complexity and up to 2^{160} data complexity. Furthermore, the key size κ can be increased up to about 180 bits and the computational complexity limit follows 2^{κ}, still with a data complexity of 2^{160}.

1.3 Advantages and limitations

The advantages of Xoodyak are the following.

- It is compact: It only requires a 48-byte state and some input and output pointers. The underlying duplex construction allows for bytes that arrive to be immediately integrated into the state without the need of a message queue.
- It foresees protections against side-channel attacks.
  - It offers leakage resilience. During a session, the secret key is a moving target, as there is no fixed key. In between sessions, it foresees a mechanism to roll keys.
  - If the same key must be used many times, one can easily add protection against implementation attacks. The degree-2 round function of Xoodoo makes masking and threshold schemes relatively cheap.
- Its specifications are short and simple, while supporting all symmetric crypto operations with a security strength of 128 bits.
- Its mode offers great flexibility and can be adapted to the specific needs of an application. For instance, it supports sessions and intermediate tags in authenticated encryption in a transparent way. Intermediate tags allow reducing the buffer at the receiving end to store the plaintext before checking the tag.
- It considers the security against multi-target attacks in the design.
- It relies on a strong and efficient permutation.
  - Xoodoo is based on the same principles as Keccak-p and hence its propagation properties are well understood.
  - Xoodoo has an exceptionally good security strength build-up per operation count. This is visible in the diffusion properties and trail bounds.
- In case of misuse (i.e., nonce misuse or release of unverified decrypted ciphertexts), the key cannot be retrieved by cryptanalysis. Authentication does not rely on a nonce.

It has the following limitations:

- It is inherently serial at construction level.
- It does stream encryption so accidental nonce re-use may result in a leakage of up to 24 bytes of plaintext.

2 Specifications

Xoodyak is an instance of the Cyclist mode of operation on top of the Xoodoo permutation. We start with the definition of the permutation in Section 2.1. Then in Section 2.2 we present the mode of operation. And finally, in Section 2.3, we define Xoodyak and its associated security claim.

2.1 The Xoodoo permutation

Xoodoo is a family of permutations parameterized by its number of rounds n_r and denoted Xoodoo$[n_\mathrm{r}]$.

Xoodoo has a classical iterated structure: It iteratively applies a round function to a state. The state consists of 3 equally sized horizontal *planes*, each one consisting of 4 parallel 32-bit *lanes*. Similarly, the state can be seen as a set of 128 *columns* of 3 bits, arranged in a 4×32 array. The planes are indexed by y, with plane $y = 0$ at the bottom and plane $y = 2$ at the top. Within a lane, we index bits with z. The lanes within a plane are indexed by x, so the position of a lane in the state is determined by the two coordinates (x, y). The bits of the state are indexed by (x, y, z) and the columns by (x, z). *Sheets* are the arrays of three lanes on top of each other and they are indexed by x. The Xoodoo state is illustrated in Figure 1.

The permutation consists of the iteration of a round function $\mathrm{R_i}$ that has 5 steps: a mixing layer θ, a plane shifting ρ_west, the addition of round constants ι, a non-linear layer χ and another plane shifting ρ_east.

We specify Xoodoo in Algorithm 1, completely in terms of operations on planes and use thereby the notational conventions we specify in Table 1. We illustrate the step mappings in a series of figures: the χ operation in Figure 2, the θ operation in Figure 3, the ρ_east and ρ_west operations in Figure 4.

The round constants C_i are planes with a single non-zero lane at $x = 0$, denoted as c_i. We specify the value of this lane for indices -11 to 0 in Table 2 and refer to Appendix A for the specification of the round constants for any index.

Finally, in many applications the state must be specified as a 384-bit string s with the bits indexed by i. The mapping from the three-dimensional indexing (x, y, z) and i is given by $i = z + 32(x + 4y)$.

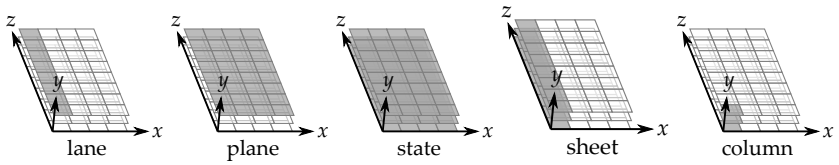

Figure 1: Toy version of the Xoodoo state, with lanes reduced to 8 bits, and different parts of the state highlighted.

Table 1: Notational conventions

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in (x, z) to position $(x + t, z + v)$
$\overline{A_y}$	Bitwise complement of plane A_y
$A_y + A_{y'}$	Bitwise sum (XOR) of planes A_y and $A_{y'}$
$A_y \cdot A_{y'}$	Bitwise product (AND) of planes A_y and $A_{y'}$

Algorithm 1 Definition of Xoodoo$[n_\mathrm{r}]$ with n_r the number of rounds

Parameters: Number of rounds n_r
for Round index i from $1 - n_\mathrm{r}$ to 0 **do**
$A = \mathrm{R_i}(A)$

Here $\mathrm{R_i}$ is specified by the following sequence of steps:

θ :
$P \leftarrow A_0 + A_1 + A_2$
$E \leftarrow P \lll (1, 5) + P \lll (1, 14)$
$A_y \leftarrow A_y + E$ for $y \in \{0, 1, 2\}$
ρ_west :
$A_1 \leftarrow A_1 \lll (1, 0)$
$A_2 \leftarrow A_2 \lll (0, 11)$
ι :
$A_0 \leftarrow A_0 + \mathrm{C}_i$
χ :
$B_0 \leftarrow \overline{A_1} \cdot A_2$
$B_1 \leftarrow \overline{A_2} \cdot A_0$
$B_2 \leftarrow \overline{A_0} \cdot A_1$
$A_y \leftarrow A_y + B_y$ for $y \in \{0, 1, 2\}$
ρ_east :
$A_1 \leftarrow A_1 \lll (0, 1)$
$A_2 \leftarrow A_2 \lll (2, 8)$

Table 2: The round constants c_i with $-11 \leq i \leq 0$, in hexadecimal notation (the least significant bit is at $z = 0$).

i	c_i	i	c_i	i	c_i	i	c_i
-11	0x00000058	-8	0x000000D0	-5	0x00000060	-2	0x000000F0
-10	0x00000038	-7	0x00000120	-4	0x0000002C	-1	0x000001A0
-9	0x000003C0	-6	0x00000014	-3	0x00000380	0	0x00000012

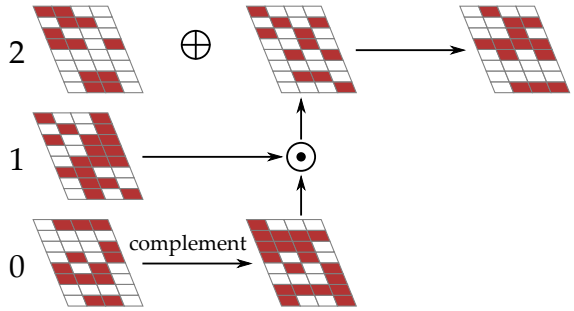

Figure 2: Effect of χ on one plane.

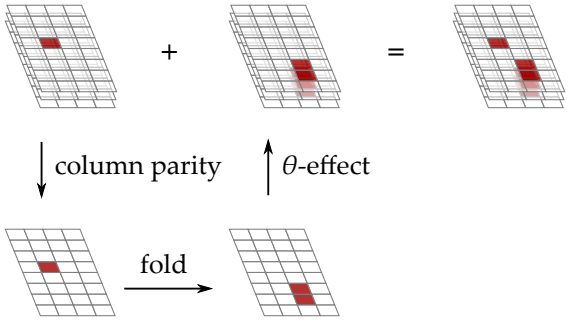

Figure 3: Effect of θ on a single-bit state.

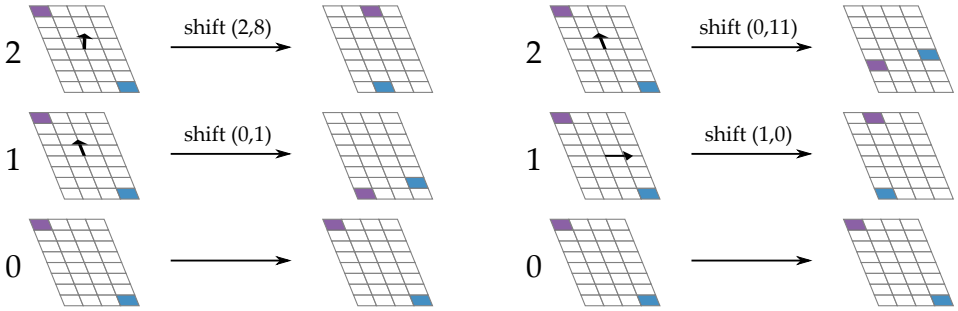

Figure 4: Illustration of ρ_{east} (left) and ρ_{west} (right).

2.2 The Cyclist mode of operation

The Cyclist mode of operation relies on a cryptographic permutation and yields a stateful object to which the user can make calls. It is parameterized by the permutation f, by the block sizes R_{hash}, R_{kin} and R_{kout}, and by the ratchet size ℓ_{ratchet}, all in bytes. R_{hash}, R_{kin} and R_{kout} specify the block sizes of the hash and of the input and output in keyed modes, respectively. As Cyclist uses up to 2 bytes for frame bits, we require that $\max(R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}) + 2 \leq b'$, where b' is the permutation width in bytes.

Upon initialization with $\textsc{Cyclist}(K, \text{id}, \text{counter})$, the Cyclist object starts either in *hash mode* if $K = \epsilon$ or in *keyed mode* otherwise. In the latter case, the object takes the secret key K together with its (optional) identifier id, then absorbs a counter in a trickled way if counter $\neq \epsilon$. In the former case, it ignores the initialization parameters. Note that, unlike Strobe, there is no way to switch from hash to keyed mode, although we might extend Cyclist this way in the future.

The available functions depend on the mode the object is started in: The functions Absorb() and Squeeze() can be called in both hash and keyed modes, whereas the functions Encrypt(), Decrypt(), SqueezeKey() and Ratchet() are restricted to the keyed mode. The purpose of each function is as follows:

- $\textsc{Absorb}(X)$ absorbs an input string X;
- $C \leftarrow \textsc{Encrypt}(P)$ enciphers P into C and absorbs P;
- $P \leftarrow \textsc{Decrypt}(C)$ deciphers C into P and absorbs P;
- $Y \leftarrow \textsc{Squeeze}(\ell)$ produces an ℓ-byte output that depends on the data absorbed so far;
- $Y \leftarrow \textsc{SqueezeKey}(\ell)$ works like $Y \leftarrow \textsc{Squeeze}(\ell)$ but in a different domain, for the purpose of generating a derived key;
- Ratchet() transforms the state in an irreversible way to ensure forward secrecy.

The state of a Cyclist object will depend on the sequence of calls to it and on its inputs. More precisely, the intention is that any output depends on the sequence of all input strings and of all input calls (i.e., Absorb(), Encrypt() and Decrypt()) so far, and that any two subsequent output calls (i.e., Squeeze() and SqueezeKey()) generate strings from different domains. It does not only depend on the concatenation of input strings, but also on their boundaries without ambiguity. For instance, a call to $\textsc{Absorb}(X)$ means the output will depend on $X \circ \textsc{Absorb}$, while a call to $\textsc{Encrypt}(P)$ will make the output depend also on $P \circ \textsc{Crypt}$. However, some dependency comes as a side-effect of other design criteria, like minimizing the memory footprint. As a result, the state also depends on the number of blocks in the previous calls to Squeeze() and the previously processed plaintext blocks in Encrypt() or Decrypt().

Together, everything that influences the output of a Cyclist object, as returned by Squeeze(), SqueezeKey() or as keystream produced by Encrypt(), is captured by the *process history*, see Definition 1 below. When in keyed mode, the output also depends on the secret key absorbed upon initialization, although the key is not part of the process history itself. This ensures the security claim can be properly expressed in an indistinguishability setting where the adversary has full control on the process history but not on the secret key, see Claim 2.

Definition 1. The *process history* (or *history* for short) is a sequence of strings and symbols in $(\mathbb{Z}_2^* \cup \mathcal{S})^*$, with

$$\mathcal{S} = \{\textsc{Absorb}, \textsc{AbsorbKey}, \textsc{Crypt}, \textsc{Squeeze}, \textsc{SqueezeKey}, \textsc{Block}, \textsc{Ratchet}\}.$$

Table 3: Symbols and strings appended to the process history.

Hash mode:	
Absorb(X)	$X \circ$ Absorb
Squeeze(ℓ) after another Squeeze()	$\textsc{Block}^{n_{\text{hash}}(\ell)} \circ$ Squeeze
Squeeze(ℓ) (otherwise)	$\textsc{Block}^{n_{\text{hash}}(\ell)}$
Keyed mode:	
Cyclist(K, id, counter)	counter $\circ$ id $\circ$ AbsorbKey
Absorb(X)	$X \circ$ Absorb
$C \leftarrow$ Encrypt(P)	$P \circ$ Crypt
$P \leftarrow$ Decrypt(C)	$P \circ$ Crypt
Squeeze(ℓ)	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ$ Squeeze
SqueezeKey(ℓ)	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ$ SqueezeKey
Ratchet()	Ratchet

At initialization of the Cyclist object, the history is initialized to $\emptyset$. Then, each call to the Cyclist object appends symbols and strings according to Table 3, where

$$n_{\text{hash}}(\ell) = \max\left(0, \left\lceil \frac{\ell}{R_{\text{hash}}} \right\rceil - 1\right) \quad \text{and} \quad n_{\text{kout}}(\ell) = \max\left(0, \left\lceil \frac{\ell}{R_{\text{kout}}} \right\rceil - 1\right).$$

In addition, the process history is updated with the R_{kout}-byte blocks of plaintext as they are processed by Encrypt() or Decrypt().

The Cyclist mode of operation is defined in Algorithms 2 and 3.

Algorithm 2 Definition of $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$

Instantiation: cyclist $\leftarrow \textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}](K, \text{id}, \text{counter})$
  Phase and state: $(\textsc{phase}, s) \leftarrow (\text{up}, \text{`}\texttt{00}\text{'}^{b'})$
  Mode and absorb rate: $(\textsc{mode}, R_{\text{absorb}}, R_{\text{squeeze}}) \leftarrow (\text{hash}, R_{\text{hash}}, R_{\text{hash}})$
  if K not empty **then** AbsorbKey(K, id, counter)

Interface: Absorb(X)
  AbsorbAny(X, R_{absorb}, `03` (absorb))

Interface: $C \leftarrow$ Encrypt(P), with mode = keyed
  return Crypt(P, false)

Interface: $P \leftarrow$ Decrypt(C), with mode = keyed
  return Crypt(C, true)

Interface: $Y \leftarrow$ Squeeze(ℓ)
  return SqueezeAny(ℓ, `40` (squeeze))

Interface: $Y \leftarrow$ SqueezeKey(ℓ), with mode = keyed
  return SqueezeAny(ℓ, `20` (key))

Interface: Ratchet(), with mode = keyed
  AbsorbAny(SqueezeAny(ℓ_{ratchet}, `10` (ratchet)), R_{absorb}, `00`)

Algorithm 3 Internal interfaces of $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$

Internal interface: $\textsc{AbsorbAny}(X, r, c_D)$
  for all blocks X_i in $\textsc{Split}(X, r)$ **do**
    if PHASE $\neq$ up **then** $\textsc{Up}(0, \text{`00`})$
    $\textsc{Down}(X_i, c_D$ **if** first block **else** `00`)

Internal interface: $\textsc{AbsorbKey}(K, \text{id}, \text{counter})$, with $|K \mid\mid \text{id}| \leq R_{\text{kin}} - 1$
  $(\text{MODE}, R_{\text{absorb}}, R_{\text{squeeze}}) \leftarrow (\text{keyed}, R_{\text{kin}}, R_{\text{kout}})$
  $\textsc{AbsorbAny}(K \mid\mid \text{id} \mid\mid \text{enc}_8(|\text{id}|), R_{\text{absorb}}, \text{`02`}\ (\text{key}))$
  if counter not empty **then** $\textsc{AbsorbAny}(\text{counter}, 1, \text{`00`})$

Internal interface: $O \leftarrow \textsc{Crypt}(I, \text{DECRYPT})$
  for all blocks I_i in $\textsc{Split}(I, R_{\text{kout}})$ **do**
    $O_i \leftarrow I_i \oplus \textsc{Up}(|I_i|, \text{`80`}\ (\text{crypt})$ **if** first block **else** `00`)
    $P_i \leftarrow O_i$ **if** DECRYPT **else** I_i
    $\textsc{Down}(P_i, \text{`00`})$
  return $||_i O_i$

Internal interface: $Y \leftarrow \textsc{SqueezeAny}(\ell, c_U)$
  $Y \leftarrow \textsc{Up}(\min(\ell, R_{\text{squeeze}}), c_U)$
  while $|Y| < \ell$ **do**
    $\textsc{Down}(\epsilon, \text{`00`})$
    $Y \leftarrow Y \mid\mid \textsc{Up}(\min(\ell - |Y|, R_{\text{squeeze}}), \text{`00`})$
  return Y

Internal interface: $\textsc{Down}(X_i, c_D)$
  $(\text{PHASE}, s) \leftarrow (\text{down}, s \oplus (X_i \mid\mid \text{`01`} \mid\mid \text{`00`}^* \mid\mid c_D \ \&\ \text{`01`}$ **if** MODE $=$ hash **else** $c_D))$

Internal interface: $Y_i \leftarrow \textsc{Up}(|Y_i|, c_U)$
  $(\text{PHASE}, s) \leftarrow (\text{up}, f(s$ **if** MODE $=$ hash **else** $s \oplus (\text{`00`}^* \mid\mid c_U)))$
  return $s[0] \mid\mid s[1] \mid\mid \ \ldots \ \mid\mid s[|Y_i| - 1]$

Internal interface: $[X_i] \leftarrow \textsc{Split}(X, n)$
  if X is empty **then return** array with a single empty string $[\epsilon]$
  return array $[X_i]$, with $X = ||_i X_i$ and $|X_i| = n$ except possibly the last block.

2.3 Xoodyak and its claimed security

We instantiate Xoodyak in Definition 2 and attach to it security Claims 1 and 2.

Definition 2. Xoodyak is $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$ with

- $f = \textsc{Xoodoo}[12]$ of width 48 bytes (or $b = 384$ bits)
- $R_{\text{hash}} = 16$ bytes
- $R_{\text{kin}} = 44$ bytes
- $R_{\text{kout}} = 24$ bytes
- $\ell_{\text{ratchet}} = 16$ bytes

Claim 1. *The success probability of any attack on* Xoodyak *in hash mode shall not be higher than the sum of that for a random oracle and $N^2/2^{255}$, with N the attack complexity*

in calls to XOODOO[12] *or its inverse. We exclude from the claim weaknesses due to the mere fact that the function can be described compactly and can be efficiently executed, e.g., the so-called random oracle implementation impossibility [14], as well as properties that cannot be modeled as a single-stage game [18].*

This means that XOODYAK hashing has essentially the same claimed security as, e.g., SHAKE128 [15].

Claim 2. *Let* $\mathbf{K} = (K_0, \ldots, K_{u-1})$ *be an array of* u *secret keys, each uniformly and independently chosen from* $\mathbb{Z}_2^\kappa$ *with* $\kappa \leq 256$ *and* κ *a multiple of* 8*. Then, the advantage of distinguishing the array of* XOODYAK *objects after initialization with* CYCLIST$(K_i, \cdot, \cdot)$ *with* $i \in \mathbb{Z}_u$ *from an array of random oracles* $\mathcal{RO}(i, h)$*, where* $h \in (\mathbb{Z}_2^* \cup \mathcal{S})^*$ *is a process history, is at most*

$$\frac{q_{\text{iv}} N + \binom{u}{2}}{2^\kappa} + \frac{N}{2^{184}} + \frac{(L+\Omega)N + \binom{L+\Omega+1}{2}}{2^{192}} + \frac{M^2}{2^{382}} + \frac{Mq}{2^{\min(192+\kappa, 384)}} \,. \tag{1}$$

Here we follow the notation of the generic security bound of the FSKD [12], namely:

- N *is the* computational complexity *expressed in the (computationally equivalent) number of executions of* XOODOO[12].
- M *is the* online *or* data complexity *expressed in the total number of input and output blocks processed by* XOODYAK.
- $q \leq M$ *is the total number of initializations in keyed mode.*
- $\Omega \leq M$ *is the number of blocks, in keyed mode, that overwrite the outer state and for which the adversary gets a subsequent output block. In particular, this counts the number of blocks processed by* DECRYPT$(\cdot)$ *for which the adversary can also get the corresponding key stream value or other subsequent output (e.g., in the case of the release of unverified decrypted ciphertext in authenticated encryption). And it also counts the number of calls to* RATCHET() *followed by* SQUEEZE(ℓ) *or* SQUEEZEKEY(ℓ) *with* $\ell \neq 0$.
- $L \leq M$ *is the number of blocks, in keyed mode, for which the adversary knows the value of the outer state from a previous query and can choose the input block value (e.g., in the case of authentication without a nonce, or of authenticated encryption with nonce repetition). This includes the number of times a call to* ABSORB() *follows a call to* SQUEEZE(ℓ) *or to* SQUEEZEKEY(ℓ) *with* $\ell \neq 0$.
- $q_{\text{iv}} \leq u$ *is the maximum number of keys that are used with the same* id*, i.e.,*

$$q_{\text{iv}} = \max_{\text{id}} |\{(i, \text{id}) \mid \text{CYCLIST}(K_i, \text{id}, \cdot) \textit{ is called at least once}\}| \,.$$

Claims 1 and 2 ensure XOODYAK has 128 bits of security both in hash and keyed modes (assuming $\kappa \geq 128$). Regarding the data complexity, it depends on the values of q, Ω and L, for which we will see concrete examples in Section 3. Given that they are bounded by M, XOODYAK resists to a data complexity of up to 2^{64} blocks, as the probability in Eq. (1) is negligible as long as $N \ll 2^{128}$ and $M \ll 2^{64}$. In the particular case of $L + \Omega = 0$, it resists even higher data complexities, as the probability remains negligible also when $M \ll 2^{160}$.

The parameter q_{iv} relates to the possible security degradations in the case of multi-target attacks, as an exhaustive key search would erode security by $\log_2 q_{\text{iv}} \leq \log_2 u$ bits in this case. However, when the protocol ensures $q_{\text{iv}} = 1$, there is no degradation and the security remains at $\min(128, \kappa)$ bits even in the case of multi-target attacks.

A rationale for the security claim is given in Section 4.

3 Using Xoodyak

Xoodyak, as a Cyclist object, can be started in hash mode and therefore used as a hash function. Alternatively, one can start Xoodyak in keyed mode and, e.g., to use it as a deck function or for duplex-like session authenticated encryption. In this section, we cover use cases in this order, first in hash mode, then in keyed mode, then some combination of both.

3.1 Hash mode

As already mentioned, Xoodyak can be used as a hash function. More generally, it can serve as an extendable output function (XOF), the generalization of a hash function with arbitrary output length. To get a n-byte digest of some input x, one can use Xoodyak as follows:

$\textsc{Cyclist}(\epsilon, \epsilon, \epsilon)$
$\textsc{Absorb}(x)$
$\textsc{Squeeze}(n)$

This sequence is the nominal sequence for using Xoodyak as a XOF. Its security is summarized in the following Corollary.

Corollary 1. *Assume that* Xoodyak *satisfies Claim 1. Then, this hash function has the following security strength levels, with n the output size in bytes:*

collision resistance	$\min(8n/2, 128)$ *bits*
preimage and second preimage resistance	$\min(8n, 128)$ *bits*
m-target preimage resistance	$\min(8n - \log m, 128)$ *bits*

Xoodyak can also naturally implement a dec function and process a sequence of strings. Here the output depends on the sequence as such and not just on the concatenation of the different strings and, in this sense it is similar to TupleHash [16]. To compute a n-byte digest over the sequence $x_3 \circ x_2 \circ x_1$, one does:

$\textsc{Cyclist}(\epsilon, \epsilon, \epsilon)$
$\textsc{Absorb}(x_1)$
$\textsc{Absorb}(x_2)$
$\textsc{Absorb}(x_3)$
$\textsc{Squeeze}(n)$

A XOF can be implemented in a stateful manner and can come with an interface that allows for requesting more output bits. This is the so-called extendable output feature, and for Cyclist this is provided quite naturally by the Squeeze() function. Here, some care must be taken for interoperability: For supporting use cases such as the one in Section 3.2.4, Cyclist considers squeezing calls as being in distinct domains. This means a Cyclist objects with some given history, the $n + m$ bytes returned by $\textsc{Squeeze}(n) \,||\, \textsc{Squeeze}(m)$ and $\textsc{Squeeze}(n + m)$ will be the same in the first n bytes and differ in the last m bytes. If an extendable output is required without this feature, an interface can be built to allow incremental squeeze calls. For instance, an interface SqueezeMore() would behave such that calling $\textsc{Squeeze}(n)$ followed by $\textsc{SqueezeMore}(m)$ is equivalent to calling $\textsc{Squeeze}(n + m)$ in the first place.

3.2 Keyed mode

In keyed mode, Xoodyak can naturally implement a deck function, although we focus instead on duplex-based ways to perform authentication and (authenticated) encryption.

To use Xoodyak as a keyed object, one starts it with Cyclist$(K, \mathrm{id}, \mathrm{counter})$ where K is a secret key with a fixed length of κ bits. We first show how to use the id and counter parameters, to counteract multi-target attacks and to handle the nonce, then discuss various kinds of authenticated encryption use cases.

3.2.1 Two ways to counteract multi-target attacks

The id is an optional key identifier. It offers one of two ways to counteract multi-target attacks.

In a multi-target attack, the adversary is not interested in breaking a specific device or key, but in breaking any device or key from a (possibly large) set. If there are u keys in a system, the security can degrade by up to $\log_2 u$ bits in such a case [8]. Claim 2 reflects this in the term $\frac{q_{\mathrm{iv}}N}{2^\kappa} \leq \frac{N}{2^{\kappa-\log_2 u}}$ as $q_{\mathrm{iv}} \leq u$.

Let us assume that we wish to target a security strength level of 128 bits including multi-target attacks. Xoodyak can achieve this in two ways.

- We extend the length of the secret key. By setting $\kappa = 128 + \log_2 u$, then the term $\frac{q_{\mathrm{iv}}N}{2^\kappa}$ becomes
$$\frac{q_{\mathrm{iv}}N}{2^{128+\log_2 u}} \leq \frac{N}{2^{128}} \ .$$
- We make the key identifier id globally unique among the u keys and therefore ensure that $q_{\mathrm{iv}} = 1$. Then, there is no degradation for exhaustive key search in a multi-target setting, and the key size can be equal to the target security strength level, so $\kappa = 128$ in this example.

3.2.2 Three ways to handle the nonce

The counter parameter of Cyclist() is a data element in the form of a byte string that can be incremented. It is absorbed in a trickled way, one digit at a time, so as to limit the number of power traces an attacker can take with distinct inputs [23]. At the upper level, the user or protocol designer fixes a basis $2 \leq B \leq 256$ and assumes that the counter is a string in $\mathbb{Z}_B^*$. A possible way to go through all the possible strings in $\mathbb{Z}_B^*$ is as follows. First, the counter is initialized to the empty string. Then, as the counter is incremented, it takes all the possible strings in $\mathbb{Z}_B^1$, then all the possible strings in $\mathbb{Z}_B^2$, and so on.

The counter shall be absorbed starting with the most significant digits. This allows caching the state after absorbing part of the counter as the first digits absorbed will change the least often. The smaller the value B, the smaller the number of possible inputs at each iteration of the permutation, so the better protection against power analysis attacks and variants.

This method of absorbing a nonce, as a counter absorbed in a trickled way, is desired in situations where protection against power analysis attacks matter. Otherwise, the nonce can be absorbed at once with Absorb(nonce) just after Cyclist$(K, \mathrm{id}, \epsilon)$.

Finally, a third method consists in integrating the nonce with the id parameter. If id is a global nonce, i.e., it is unique among all the keys used in the system, this also ensures $q_{\mathrm{iv}} = 1$ as explained above.

3.2.3 Authenticated encryption

We propose using Xoodyak for authenticated encryption as follows. To encrypt a plaintext P under a given nonce and associated data A under key K with identifier id, and to

get a tag of $t = 16$ bytes, we make the following sequence of calls:

```
Cyclist(K, id, ϵ)
Absorb(nonce)
Absorb(A)
C ← Encrypt(P)
T ← Squeeze(t)
return (C, T)
```

To decrypt (C, T), we proceed similarly:

```
Cyclist(K, id, ϵ)
Absorb(nonce)
Absorb(A)
P ← Decrypt(C)
T′ ← Squeeze(t)
if T = T′ then
  return P
else
  return ⊥
```

If the nonce is not repeated and if the decryption does not leak unverified decrypted ciphertexts, then we have $L = \Omega = 0$ here, see Claim 2. The resulting simplified security claim is given in the following corollary.

Corollary 2. *Assume that (1)* Xoodyak *satisfies Claim 2; (2) this authenticated encryption scheme is fed with a single κ-bit key with $\kappa \leq 192$; (3) it is implemented such that the nonce is not repeated and the decryption does not leak unverified decrypted ciphertexts. Then, it can be distinguished from an ideal scheme with an advantage whose dominating terms are:*

$$\frac{N}{2^\kappa} + \frac{N}{2^{184}} + \frac{M^2}{2^{192+\kappa}} \ .$$

This translates into the following security strength levels assuming a t-byte tag (the complexities are in bits):

	computation	*data*
plaintext confidentiality	$\min(184, \kappa, 8t)$	$96 + \kappa/2$
plaintext integrity	$\min(184, \kappa, 8t)$	$96 + \kappa/2$
associated data integrity	$\min(184, \kappa, 8t)$	$96 + \kappa/2$

3.2.4 Session authenticated encryption

Session authenticated encryption works on a sequence of messages and the tag authenticates the complete sequence of messages received so far. Starting from the sequence in Section 3.2.3, we add the support for messages (A_i, P_i), where A_i, P_i or both can be empty.

```
Cyclist(K, id, ϵ)
Absorb(nonce)
Absorb(A_1)
C_1 ← Encrypt(P_1)
T_1 ← Squeeze(t)
  ⇒ output (C_1, T_1) and wait for next message
Absorb(A_2)
```

$C_2 \leftarrow \textsc{Encrypt}(P_2)$
$T_2 \leftarrow \textsc{Squeeze}(t)$
  $\Rightarrow$ output (C_2, T_2) and wait for next message
$\textsc{Absorb}(A_3)$
$T_3 \leftarrow \textsc{Squeeze}(t)$
  $\Rightarrow$ output T_3 and wait for next message
$C_4 \leftarrow \textsc{Encrypt}(P_4)$
$T_4 \leftarrow \textsc{Squeeze}(t)$
  $\Rightarrow$ output (C_4, T_4) and wait for next message
$T_5 \leftarrow \textsc{Squeeze}(t)$
  $\Rightarrow$ output T_5 and wait for next message

In this example, T_2 authenticates $(A_2, P_2) \circ (A_1, P_1)$. The third message has no plaintext, the fourth message has no associated data, and the fifth message is empty. In such a sequence, the convention is that the call to Squeeze() ends a message. Since it appears in the processing history, there is no ambiguity on the boundaries of the messages even if some of the elements (or both) are empty.

The use of empty messages may be clearer in the case of a session shared by two (or more) communicating devices, where each device takes a turn. A device may have nothing to say and so skips its turn by just producing a tag.

To relate to Claim 2, we have to determine L by counting the number of invocations to Absorb() that follow Squeeze(). If the nonce is not repeated and if the decryption does not leak unverified decrypted ciphertexts, we have $L = T - q$, with T the number of messages processed (or tags produced), and $\Omega = 0$.

3.2.5 Ratchet

At any time in keyed mode, the user can call Ratchet(). This causes part of the state to be overwritten with zeroes, thereby making it computationally infeasible to compute the state value before the call to Ratchet().

In an authenticated encryption scheme, the call to Ratchet() can be typically inserted either just before producing the tag or just after. The advantage of calling it just before the tag is that it is most efficient: It requires only one extra call to the permutation f. An advantage of calling it just after the tag is that its processing can be done asynchronously, while the ciphertext is being transmitted and it waits for the next message. Unless Ratchet() is the last call, the number of calls to it must be counted in Ω.

$\textsc{Cyclist}(K, \mathrm{id}, \epsilon)$
$\textsc{Absorb}(\mathrm{nonce})$
$\textsc{Absorb}(A)$
$C \leftarrow \textsc{Encrypt}(P)$
$\textsc{Ratchet}()$ {either here ...}
$T \leftarrow \textsc{Squeeze}(t)$
$\textsc{Ratchet}()$ {... or here}

3.2.6 Rolling subkeys

As an alternative to using a long-term secret key together with its associated nonce that is incremented at each use, Cyclist offers a mechanism to derive a subkey via the SqueezeKey() call. On an encrypting device, one can therefore replace the process of incrementing and storing the updated nonce at each use of the long-term secret key with the process of updating a rolling subkey:

$K_1 \leftarrow K$ and $i \leftarrow 1$
while necessary **do**
  Initialize a new XOODYAK instance with CYCLIST$(K_i, \epsilon, \epsilon)$
  $K_{i+1} \leftarrow$ SQUEEZEKEY(ℓ_{sub}) {and store K_{i+1} by overwriting K_i}
  RATCHET() {optional}
  ABSORB(A_i)
  $C_i \leftarrow$ ENCRYPT(P_i)
  $T_i \leftarrow$ SQUEEZE(t)
    $\Rightarrow$ output (C_i, T_i) and wait for next message
  $i \leftarrow i + 1$

Here ℓ_{sub} should be chosen large enough to avoid collisions, say $\ell_{\text{sub}} = 32$ bytes (256 bits). Assuming that there are no collisions in the subkeys, $L = 0$ and Ω is the number of calls to RATCHET().

Using Cyclist this way offers resilience against side channel attacks, as the long-term key is not exposed any more and can even be discarded as soon as the first subkey is derived. The key to attack becomes a moving target, just like the state in session authenticated encryption.

3.2.7 Nonce reuse and release of unverified decrypted ciphertext

The authenticated encryption schemes presented in this section assume that the nonce is unique per session, namely that the value is used only once per secret key. It also assumes that an implementation returns only an error when receiving an invalid cryptogram and in particular does not release the decrypted ciphertext if the tag is invalid. If these two assumptions are satisfied, we refer to this as the *nominal case*; otherwise, we call it the *misuse case*.

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

A nonce violation in general breaks confidentiality of part of the plaintext. In particular, two sessions that have the same key and the same process history (i.e., the same K, id, counter and the same sequence of associated data, plaintexts) will result in the same output (ciphertext, tag). We call such a pair of sessions in-sync. Clearly, in-sync sessions leak equality of inputs and hence also plaintexts. As soon as in-sync sessions get different input blocks, they lose synchronicity. If these input blocks are plaintext blocks, the corresponding ciphertext blocks leak the bitwise difference of the corresponding plaintext blocks (of $R_{\text{kout}} = 24$ bytes). We call this the *nonce-misuse leakage*.

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

Nonce violation and release of unverified decrypted ciphertext have no consequences for integrity and do not put the key in danger for XOODYAK. This is formalized in Corollary 3.

Corollary 3. *Assume that (1)* XOODYAK *satisfies Claim 2; (2) this authenticated encryption scheme is fed with a single κ-bit key with $\kappa \leq 192$. Then, except for nonce-misuse leakage and keystream harvesting, it can be distinguished from an ideal scheme with an advantage whose dominating terms are:*

$$\frac{N}{2^{\kappa}} + \frac{N}{2^{184}} + \frac{MN + M^2}{2^{192}} \ .$$

This translates into the following security strength levels assuming a t-byte tag (the complexities are in bits):

	computation	*data*
plaintext confidentiality (nominal case)	$\min(128, \kappa, 8t)$	64
plaintext confidentiality (misuse case)	-	-
plaintext integrity	$\min(128, \kappa, 8t)$	64
associated data integrity	$\min(128, \kappa, 8t)$	64

3.3 Authenticated encryption with a common secret

A key exchange protocol, such as Diffie-Hellman or variant, results in a common secret that usually requires further derivation before being used as a symmetric secret key. To do this with a Cyclist object, we can use an object in hash mode, process the common secret, and use the derived key in a new object that we start in keyed mode. For example:

$\textsc{Cyclist}(\epsilon, \epsilon, \epsilon)$
$\textsc{Absorb}$(ID of the chosen protocol)
$\textsc{Absorb}(K_A)$ {Alice's public key}
$\textsc{Absorb}(K_B)$ {Bob's public key}
$\textsc{Absorb}(K_{AB})$ {Their common secret produced with Diffie-Hellman}
$K_D \leftarrow \textsc{Squeeze}(\ell)$

$\textsc{Cyclist}(K_D, \epsilon, \epsilon)$
$\textsc{Absorb}$(nonce)
$\textsc{Absorb}(A)$
$C \leftarrow \textsc{Encrypt}(P)$
$T \leftarrow \textsc{Squeeze}(t)$
return (C, T)

Note that if $\ell \leq R_{\text{hash}}$, an implementation can efficiently chain $K_D \leftarrow \textsc{Squeeze}(\ell)$ and the subsequent reinitialization $\textsc{Cyclist}(K_D, \epsilon, \epsilon)$. Since K_D is located in the outer part of the state, it needs only to set the rest of the state to the appropriate value before calling f.

Note also that if at least one of the public key pairs is ephemeral, the common secret K_{AB} is used only once and no nonce is needed.

4 Design rationale

In this section, we give the design rationale of Xoodyak. First, we give the general strategy. Then, we report on the generic security of the Cyclist mode and relate it to Xoodyak's security claim. Finally, we highlight the properties of the Xoodoo[12] permutation.

4.1 Design strategy

Xoodyak connects a mode of operation, namely Cyclist, to a permutation, namely Xoodoo[12]. The design strategy is *hermetic* in the following sense: We chose the number of rounds in Xoodoo such that the best attacks on Xoodyak are (claimed to be) the generic attacks on the Cyclist mode. This is visible in the security claims Claim 1 and 2, as they replicate the best known security bounds of the sponge and keyed duplex constructions. In contrast, a non-hermetic strategy would keep some buffer between the claimed security level and the generic attacks.

Note that the strategy behind Xoodyak differs from the so-called "hermetic sponge strategy" [5]. Putting aside definitional issues, the hermetic sponge strategy described in [5] targets the absence of distinguishers in an absolute sense, whereas we only consider

the security of the resulting function Xoodyak. Hence we do not claim that Xoodoo[12] is free of distinguishers, only that it is strong enough when plugged in Cyclist.

4.2 Generic security and the security claim

We now give more details to relate the security of the sponge and keyed duplex constructions to Xoodyak's security claim.

4.2.1 Xoodyak in hash mode

In hash mode, Cyclist can be reduced to the sponge construction with a capacity of $c = b - 8R_{\text{hash}} - 2$, so $c = 254$ bits in the case of Xoodyak. In addition to the contents of the input blocks of R_{hash} bytes prepared by Absorb(), variable frame bits can be added at only two additional positions (see Down() in Algorithm 3), hence accounting for a reduction of 2 bits in the capacity. We then make a flat sponge claim [2] with claimed capacity equal to c, hence accounting for a success probability of $\frac{N^2}{2^{c+1}} = \frac{N^2}{2^{255}}$ in Claim 1.

4.2.2 Xoodyak in keyed mode

When in keyed mode, Cyclist can be rephrased in terms of calls to the FSKD with $c = b - 8R_{\text{kout}}$, so $c = 192$ bits in the case of Xoodyak. A crucial property of the FSKD is that each duplexing call starts with applying the permutation f, then generates a block of output and finally adds an input block to the outer state. In the language of Cyclist, a duplexing call translates into a sequence of Up() followed by Down(X). This cycle is exactly an iteration of Encrypt(), where the plaintext block is given before the corresponding keystream block is output, so an iteration of Encrypt() directly translates to a call to FSKD. A similar comment applies to AbsorbAny(), possibly except the first iteration.

However, a call to SqueezeAny() always ends with Up(), without knowing what the next input will be. To simulate this and properly remap it to the FSKD setting [12], we can say that in that case the Cyclist object gets its output block by making a duplexing call with an arbitrary input block. When the actual input block becomes known, it restarts the whole FSKD object with the same queries, but this time with the correct input block. This is like re-doing a query with the same path and is accounted for in L each time it happens.

The different terms of Claim 2's Eq. (1) stem from the security bound in the FSKD paper [12], which we now detail.

- $\frac{(L+\Omega)N}{2^c}$ and $\frac{\binom{L+\Omega+1}{2}}{2^c}$ are present as is in Eq. (1).
- $\frac{2\nu_{r,c}^{2(M-L)}(N+1)}{2^c}$ is upper bounded as $\frac{2b(N+1)}{4\times 2^c}$ since it is shown in [12] that $2^{(r-c)/2} < M \leq 2^{r-1}$ implies $\nu_{r,c}^{2(M-L)} \leq \nu_{r,c}^{2M} \leq b/4$. When then bound $\frac{2b(N+1)}{4\times 2^c} = \frac{384(N+1)}{2^{c+1}} \leq \frac{N}{2^{c-8}}$.
- $\frac{(M-q-L)q}{2^b-q} + \frac{M(M-L-1)}{2^b}$ is not greater than $\frac{4M^2}{2^b}$ for $q \leq 2^{b-1}$, and $\frac{4M^2}{2^b} = \frac{M^2}{2^{382}}$ here.
- $H_{\min(\mathcal{D}_{\text{K}})} = H_{\text{coll}}(\mathcal{D}_{\text{K}}) = \kappa$ in our setting, so $\frac{(M-q-L)q}{2^{H_{\min(\mathcal{D}_{\text{K}})}+\min(c,b-k)}}$ is upper bounded as $\frac{Mq}{2^{\min(c+\kappa,b)}}$, $\frac{q_{\text{iv}}N}{2^{H_{\min(\mathcal{D}_{\text{K}})}}}$ as $\frac{uN}{2^\kappa}$ (since $q_{\text{iv}} \leq u$), and $\frac{\binom{u}{2}}{2^{H_{\text{coll}}(\mathcal{D}_{\text{K}})}} = \frac{\binom{u}{2}}{2^\kappa}$.

4.2.3 Decodability

In this section, we show that, given the sequence of b-bit blocks that are added to the state between each call to f, one can recover the process history of the Xoodyak object, together with the secret key if in keyed mode.

First let us observe that any sequence of calls to the Cyclist object is translated internally into an alternating sequence of Down(X_i, c_D) and Up$(|Y_i|, c_U)$ steps. The first step is the internal input step that takes a message block X_i, applies a simple reversible padding to it and injects the result into the state, complemented optionally by a *color byte* c_D, i.e., a byte that performs domain separation between the different operations. The second step is the internal output step, which first optionally injects a color byte c_U into state, applies the permutation f and then produces the requested number of bytes as output. Since the parts of the state that these two steps deal with are not overlapping, and since each input block X_i is padded in a reversible way, it is straightforward to extract from the b-bit block sequence the corresponding calls to Down() and Up() along with their parameters X_i, c_D and c_U. We ignore the output length parameter $|Y_i|$ that is not necessary for the decodability.

In general, each Cyclist call starts with a first colored step and continues with zero, one or several uncolored steps. One can use this property to easily detect where each call starts in the alternating sequence of Down() and Up() steps. There are a few exceptions to this color property that we detail now.

The most notable exception is in hash mode, where none of Squeeze() steps are colored. If there are Down() steps, these will have empty input strings. For the sake of decodability, we can then simply consider that these steps are part of the previous call.

There are also exceptions in keyed mode. In the case the phase is down, Absorb() will start with an uncolored Up() step. This case may occur for instance if Absorb() is called twice in a row. A similar yet more subtle situation occurs if Squeeze() is called after any call that terminates with a Up() step. In that case, Squeeze() starts with an implicit uncolored *void step*, i.e., a Down()-like step that has no effect on the state. The same situation occurs for Encrypt(), Decrypt() and Ratchet(). For all these exceptions, we can in fact either ignore the first uncolored step or consider that this step is part of the sequence attached to the previous call. Since each call to Cyclist is associated with a unique color, we can then use this color property to decode the alternating step sequence and extract the corresponding call parameters.

To summarize, the decodability of Cyclist works as follows. First, we convert the sequence of b-bit blocks that are added to the state into the corresponding sequence of step calls Down() and Up() along with their parameters. Working backward, we cut this sequence into sub-sequences, each starting with a colored step (or a void step) and followed by zero, one or more uncolored steps. We associate then each sub-sequence to corresponding call, reconstructing when necessary the message parameter from the concatenation of all block parameters extracted in the sub-sequence. This is illustrated in Table 4. In hash mode, we observe that although calls to Squeeze() are not meant to be decodable, some of them can still be decoded as a side-effect of the insertion of a void step (denoted $d()$) between two consecutive calls to Squeeze(), or due to empty down steps that appear in long Squeeze() calls ($\ell > R_{\text{hash}}$).

4.3 Choice of the permutation

The choice of the permutation was driven by the idea of sharing resources between hash and keyed modes. The size of the permutation is therefore determined mainly by the hash mode, as for a given security level, it requires more capacity than the keyed mode. Since 128-bit security is desired, we need to have a capacity of at least 256 bits to prevent collisions. The permutation should therefore be wider than 256 bits, but not too much

Table 4: Matching up and down sub-sequences with process history.

Hash mode:	
$[u(\cdot)]\ d(X_i, \text{`01`})\ (u(\cdot)\ d(X_i \text{ or } \epsilon, \cdot))^*\ [u(\cdot)]$	$\textsc{Block}^* \circ X = \|\|_i X_i \circ \textsc{Absorb}$
$d()\ u(\cdot)\ (d(\epsilon, \cdot)\ u(\cdot))^{n_{\text{hash}}(\ell)}$	$\textsc{Block}^{n_{\text{hash}}(\ell)} \circ \textsc{Squeeze}$
Keyed mode:	
$[u(\cdot)]\ d(X_i, \text{`02`})\ (u(\cdot)\ d(X_i, \cdot))^*$	$(\text{counter} \circ (K\|\|\text{id})) = \|\|_i X_i \circ \textsc{AbsorbKey}$
$[u(\cdot)]\ d(X_i, \text{`03`})\ (u(\cdot)\ d(X_i, \cdot))^*$	$X = \|\|_i X_i \circ \textsc{Absorb}$
$[d()]\ u(\text{`80`})\ d(X_i, \cdot)\ (u(\cdot)\ d(X_i, \cdot))^*$	$P = \|\|_i X_i \circ \textsc{Crypt}$
$[d()]\ u(\text{`40`})\ (d(\epsilon, \cdot)\ u(\cdot))^{n_{\text{kout}}(\ell)}$	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ \textsc{Squeeze}$
$[d()]\ u(\text{`20`})\ (d(\epsilon, \cdot)\ u(\cdot))^{n_{\text{kout}}(\ell)}$	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ \textsc{SqueezeKey}$
$[d()]\ u(\text{`10`})\ (d(X_i \text{ or } \epsilon, \cdot)\ u(\cdot))^*$	$\textsc{Ratchet}$

Table 5: The weight of the best differential and linear trails (or lower bounds) as a function of the number of rounds.

# rounds:	1	2	3	4	5	6
differential:	2	8	36	≥ 74	≥ 88	≥ 104
linear:	2	8	36	≥ 74	≥ 88	≥ 104

wider.

A possible candidate was Keccak-$p[400, n_r]$, as the permutation size leaves enough room for the input block. However, it uses operations on 16-bit lanes but 16-bit processors are not so common nowadays. Instead, the choice of Xoodoo was quite natural as it shares a lot of similarity with the Keccak-p family and works on 32-bit lanes. The entire state of 384 bits can be held in 12 registers of 32 bits, making it a nice fit with the low-end 32-bit devices.

For the design rationale of Xoodoo, we give here some highlights and refer to [10] for more details. Xoodoo operates on three planes of 128 bits each, which interact per 3-bit columns through mixing and nonlinear operations, and which otherwise move as three independent rigid objects. Its round function uses the five step mappings θ, ρ_{west}, ι, χ and ρ_{east}. The nonlinear layer χ is an instance of the transformation χ that was already described and analyzed in [9], and that operates on 3 bits in Xoodoo. It has algebraic degree 2, it is involutive and hence r rounds of Xoodoo or its inverse cannot have an algebraic degree higher than 2^r. The mixing layer θ is a column parity mixer [22]. As in both the parity plane computation in θ and in χ the state bits interact only within columns, the dispersion steps aim at dislocating the bits of the columns between every application of θ and of χ. For that reason, ρ_{east} and ρ_{west} shift the planes, treating them as rigid objects, between each χ and each θ step. Finally, the translation-invariance symmetry is destroyed by adding a round constants in the step ι.

The Xoodoo round function exhibits fast avalanche properties: It needs 3.5 rounds or 2 inverse rounds to satisfy the strict avalanche criterion [24]. Like Keccak-p, it has so-called weak alignment [4], where alignment characterizes the way differences or linear correlations propagate. The weak alignment has the advantage of making Xoodoo less susceptible to truncated differentials attacks or to trail clustering effects.

Finally, in terms of differential and linear cryptanalysis, Xoodoo has strong bounds on the weight of its trails. Note that the weight of a trail relates to its differential probability or its correlation, see [10, Section 5.2] for more details. Since the publication of Xoodoo in [10], we have extended the trail search and improved the bounds. Table 5 shows the currently known lower bounds.

The choice for the number of rounds, namely 12, comes for one part from our experience

in designing sponge-based hash functions and authenticated encryption schemes, and for another part from the similarity to KECCAK-p on which extensive cryptanalysis has been performed in the last ten years [7]. With 12 rounds, XOODOO[12] has strong avalanche, differential and linear propagation properties, even stronger than those of KECCAK-$p[400, n_r]$ in terms of differential and linear trails. Even if an attack can somehow skip 4 rounds, it is guaranteed that any 8-round trail, either differential or linear, has weight at least 148.

For hashing, the best collision or (second) preimage attack on KECCAK reaches only 5 or 6 rounds, depending on how many degrees of freedom are available [21]. Note that in hashing mode, XOODYAK has a much smaller rate, hence much less degrees of freedom, than the aforementioned KECCAK instances.

For keyed operations, we believe that XOODOO[12] is suitable to be plugged in the full-state keyed duplex construction, on which Cyclist relies. As a comparison, this is the same number of rounds that was used for KECCAK-p in KEYAK [6], also relying on the full-state keyed duplex construction.

4.4 Known attacks

At the time of writing, there are no known attacks on XOODYAK and therefore Claims 1 and 2 can plausibly be believed to hold.

XOODYAK is built on strong foundations and based on conservative design choices. There is a large number of research papers on the generic security of sponge and duplex-based modes, on KECCAK, KETJE, KEYAK and other permutation-based designs for hashing or authenticated encryption. These show the fairly wide understanding of the field around XOODYAK by the cryptographic community.

It is interesting to note that cube attacks were attempted on a XOODOO-based authenticated encryption scheme following the same mode as KETJE [20]. The authors succeed on the initialization phase reduced to 6 rounds of XOODOO. Despite that XOODYAK does not use the same mode as KETJE, there is nevertheless significant similarity between their initializations, and we can deduce from this research that 12 rounds provide enough safety margin against this type of attacks. Furthermore, the authors discuss the effects of switching from 5-bit to 3-bit χ between KECCAK-p and XOODOO, and argue that the narrower χ contributes to an increased resistance against cube-attack-like analysis.

4.5 Tunable parameters

XOODYAK does not have user-chosen parameters, as the security claims apply to the only defined instance of XOODYAK. In contrast, KECCAK has user-chosen parameters, namely the rate and capacity, for which the full range is covered by a security claim.

This said, should the need arise, we can already identify the parameters that could be modified to adapt XOODYAK's performance or security.

- The number of rounds of the permutation. Clearly, this is an essential parameter to protect against shortcut attacks. Reducing it can improve the performance but lower the safety margin. Should shortcut attacks be found, it can be increased to add safety margin.

- The different rates R_{hash}, R_{kin} and R_{kout}. In a hermetic approach, tuning the rates (hence the associated capacities) have an impact mainly on the generic security. Increasing such a rate would have a positive impact on performance and the expense of the generic, and therefore claimed, security levels. For instance, we have a lot of margin in terms of data complexity in the case $L = \Omega = 0$ (see Corollary 2), and in that case we could increase R_{kout} to, say, 28 bytes. In the other direction, we could also wish to increase the generic and claimed security levels by reducing R_{hash} or

Table 6: Performance figures of XOODYAK in cycles per byte.

	ARM Cortex-M0	ARM Cortex-M3
Hash mode		
ABSORB()	134.5	39.3
SQUEEZE()	136.2	40.6
Keyed mode		
ABSORB()	48.7	14.2
ENCRYPT()	91.2	27.1
DECRYPT()	91.3	27.4
SQUEEZE()	86.2	24.3

R_{kin}. Decreasing these rates may also be a way to counteract some shortcut attacks, but this idea is not in the spirit of a hermetic approach.

5 Implementation aspects

The implementation aspects of XOODYAK essentially rely on those of the underlying permutation XOODOO. We therefore refer to [10, Section 4] for more details.

In Table 6, we report on the performance of XOODYAK on ARM Cortex-M0 and -M3.

6 Submission to NIST Lightweight Cryptography Standardization Process

This document is part of the submission of XOODYAK to NIST Lightweight Cryptography Standardization Process.

- The algorithm submitted for authenticated encryption with associated data (AEAD) is the sequence in Section 3.2.3 executed with XOODYAK. By default, the key length is $\kappa = 128$ bits, there is no global key identifier (id $= \epsilon$), the nonce length is 128 bits and the tag length is 128 bits. The amount of data that can be processed by a key is only implied by the security claim.
- The algorithm submitted for hashing is the first sequence in Section 3.1 executed with XOODYAK. The default output length is $n = 32$ bytes, or otherwise freely chosen as in a XOF. The limit on the message size is only implied by the security claim.

These two algorithms share the same underlying XOODYAK algorithm and an implementation would naturally share the XOODOO permutation and several input-output operations for absorbing and squeezing data. These two algorithms are therefore paired to be evaluated jointly.

References

[1] D. J. Bernstein, S. Kölbl, S. Lucks, P. Maat Costa Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier, *Gimli : A cross-platform permutation*, Cryptographic Hardware and Embedded Systems - CHES 2017, Proceedings (W. Fischer and N. Homma, eds.), Lecture Notes in Computer Science, vol. 10529, Springer, 2017, pp. 299–320.

[2] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Cryptographic sponge functions*, January 2011, https://keccak.team/files/SpongeFunctions.pdf.

[3] ———, *Duplexing the sponge: Single-pass authenticated encryption and other applications*, Selected Areas in Cryptography - SAC 2011, Revised Selected Papers (A. Miri and S. Vaudenay, eds.), Lecture Notes in Computer Science, vol. 7118, Springer, 2011, pp. 320–337.

[4] ———, *On alignment in* Keccak, ECRYPT II Hash Workshop 2011, 2011.

[5] ———, *The* Keccak *reference*, January 2011, https://keccak.team/papers.html.

[6] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, *CAESAR submission:* Keyak *v2, document version 2.2*, September 2016, https://keccak.team/keyak.html.

[7] ———, Keccak *third-party cryptanalysis*, 2019, https://keccak.team/third_party.html.

[8] E. Biham, *How to decrypt or even substitute des-encrypted messages in* 2^{28} *steps*, Inf. Process. Lett. **84** (2002), no. 3, 117–124.

[9] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis, PhD thesis*, K.U.Leuven, 1995.

[10] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, *The design of Xoodoo and Xoofff*, IACR Trans. Symmetric Cryptol. **2018** (2018), no. 4, 1–38.

[11] ———, *Xoodoo cookbook*, IACR Cryptology ePrint Archive **2018** (2018), 767.

[12] J. Daemen, B. Mennink, and G. Van Assche, *Full-state keyed duplex with built-in multi-user support*, Advances in Cryptology - ASIACRYPT 2017, Proceedings, Part II (T. Takagi and T. Peyrin, eds.), Lecture Notes in Computer Science, vol. 10625, Springer, 2017, pp. 606–637.

[13] M. Hamburg, *The STROBE protocol framework*, Real World Crypto, 2017.

[14] U. Maurer, R. Renner, and C. Holenstein, *Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology*, Theory of Cryptography - TCC 2004 (M. Naor, ed.), Lecture Notes in Computer Science, no. 2951, Springer-Verlag, 2004, pp. 21–39.

[15] NIST, *Federal information processing standard 202, SHA-3 standard: Permutation-based hash and extendable-output functions*, August 2015, http://dx.doi.org/10.6028/NIST.FIPS.202.

[16] ———, *NIST special publication 800-185, SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash*, December 2016, https://doi.org/10.6028/NIST.SP.800-185.

[17] T. Perrin, *Stateful hash objects: API and constructions*, https://github.com/noiseprotocol/sho_spec/blob/master/output/sho.pdf, 2018.

[18] T. Ristenpart, H. Shacham, and T. Shrimpton, *Careful with composition: Limitations of the indifferentiability framework*, Eurocrypt 2011 (K. G. Paterson, ed.), Lecture Notes in Computer Science, vol. 6632, Springer, 2011, pp. 487–506.

[19] M.-J. O. Saarinen, *Beyond modes: Building a secure record protocol from a cryptographic sponge permutation*, Topics in Cryptology - CT-RSA 2014. Proceedings (J. Benaloh, ed.), Lecture Notes in Computer Science, vol. 8366, Springer, 2014, pp. 270–285.

[20] L. Song and J. Guo, *Cube-attack-like cryptanalysis of round-reduced Keccak using MILP*, IACR Trans. Symmetric Cryptol. **2018** (2018), no. 3, 182–214.

[21] L. Song, G. Liao, and J. Guo, *Non-full sbox linearization: Applications to collision attacks on round-reduced Keccak*, Advances in Cryptology - CRYPTO 2017 (J. Katz and H. Shacham, eds.), Lecture Notes in Computer Science, vol. 10402, Springer, 2017, pp. 428–451.

[22] K. Stoffelen and J. Daemen, *Column parity mixers*, IACR Trans. Symmetric Cryptol. **2018** (2018), no. 1, 126–159.

[23] M. M. I. Taha and P. Schaumont, *Side-channel countermeasure for SHA-3 at almost-zero area overhead*, 2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014, IEEE Computer Society, 2014, pp. 93–96.

[24] A. F. Webster and S. E. Tavares, *On the design of S-boxes*, Advances in Cryptology - CRYPTO '85, Proceedings (H. C. Williams, ed.), Lecture Notes in Computer Science, vol. 218, Springer, 1985, pp. 523–534.

A Constants for any number of rounds

See [10].