IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2020, No. S1, pp. 60–87. DOI:10.13154/tosc.v2020.iS1.60-87

Xoodyak, a lightweight cryptographic scheme

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

[1] STMicroelectronics, Diegem, Belgium
michael-tosc@noekeon.org,gilles-tosc@noekeon.org,ronny.vankeer@st.com
[2] Radboud University, Nijmegen, The Netherlands
joan@cs.ru.nl,seth.hoffert@gmail.com

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.

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 variant when it is fed with a secret key [BDPA11b, MRV15, DMA17]. It is stateful and shares features with Markku Saarinen's Blinker [Saa14], Mike Hamburg's Strobe protocol framework [Ham17] and Trevor Perrin's Stateful Hash Objects (SHO) [Per18]. 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 [DHAK18a, DHAK18b]. The design approach of this 384-bit permutation is inspired by Keccak-p [BDPA11d, NIS15], while it is dimensioned like Gimli for efficiency on low-end processors [BKL+17]. 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 [BDP+16]. 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.

Received: 2019-12-10, Accepted: 2020-02-15, Published: 2020-06-22

Of interest in the realm of embedded devices, Xoodyak contains several built-in mechanisms that help protect against side-channel attacks.

- Following an idea by Taha and Schaumont [TS14], 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 [BDPA11b]. 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($\epsilon, \epsilon, \epsilon$) {initialization in hash mode}
Absorb($x$) {absorb string $x$}
$h \leftarrow$ 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) [NIS15].

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 [NIS16].

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($K, \epsilon, \epsilon$) {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($K, \epsilon, \epsilon$)
Absorb(nonce)
$C \leftarrow$ Encrypt($P$) {get ciphertext $C$}
```

And to decrypt ciphertext C, 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($K, \epsilon, \epsilon$)
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. Furthermore, the permutation can be computed in-place [DHAK18a, Section 4.1].
- 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[12] permutation. In Section 2.1, we briefly describe the Xoodoo[12] permutation, as its specifications can be found in [DHAK18a, Section 2] and are not repeated here. Then, in Section 2.2 we present the mode of operation, while in Section 2.3 we define Xoodyak and its associated security claim.

2.1 Xoodoo[12]

Xoodoo is a family of permutations parameterized by its number of rounds n_r and denoted Xoodoo[n_r]. In the case of Xoodyak, the number of rounds is fixed to 12. The width of Xoodoo, i.e., its input and output size, is $b = 384$ bits.

The specifications of Xoodoo can be found in [DHAK18a, Section 2].

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 (i.e., bits used for padding and domain separation), we require that $\max(R_\mathrm{hash}, R_\mathrm{kin}, R_\mathrm{kout}) + 2 \leq b'$, where $b' = b/8$ is the permutation width in bytes.

Upon initialization with $\textsc{Cyclist}(K, \mathrm{id}, \mathrm{counter})$, 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 $\mathrm{counter} \neq \epsilon$. In the former case, it ignores the initialization parameters. Note that, unlike Strobe [Ham17], 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.

Table 1: Symbols and strings appended to the process history.

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

Process history. Together, everything that influences the output of a Cyclist object, as returned by $\textsc{Squeeze}()$, $\textsc{SqueezeKey}()$ or as keystream produced by $\textsc{Encrypt}()$, is captured by the *process history*, see Definition 1 below.

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 (without ambiguity on their boundaries) and of all input calls (i.e., $\textsc{Absorb}()$, $\textsc{Encrypt}()$ and $\textsc{Decrypt}()$) so far. In addition, any two subsequent output calls (i.e., $\textsc{Squeeze}()$ and $\textsc{SqueezeKey}()$) generate strings from different domains.

To capture the idea that the output depends also on the operations performed, the process history contains symbols from the set $\mathcal{S}$ in Definition 1 below. For instance, a call to $\textsc{Absorb}(X)$ means the output will depend on the sequence $X \circ \textsc{Absorb}$, while a call to $\textsc{Encrypt}(P)$ will make the output depend on $P \circ \textsc{Crypt}$, with $\textsc{Absorb}, \textsc{Crypt} \in \mathcal{S}$.

As a side-effect of other design criteria, like minimizing the memory footprint, the state also depends on the number of blocks in the previous calls to $\textsc{Squeeze}()$ or $\textsc{SqueezeKey}()$ and on the previously processed plaintext or ciphertext blocks in $\textsc{Encrypt}()$ or $\textsc{Decrypt}()$. In particular, each symbol $\textsc{Block}$ in the process history represents an extra output block requested via $\textsc{Squeeze}()$ or $\textsc{SqueezeKey}()$, see Table 1.

Note that, when in keyed mode, the output naturally also depends on the secret key absorbed upon initialization, although the key is not part of the process history itself. This ensures that 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

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

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 1, 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 $\textsc{Encrypt}()$ or $\textsc{Decrypt}()$.

Inside Cyclist. The Cyclist mode of operation is defined in Algorithms 1 and 2. Here are some notes for a better understanding of the behavior of Cyclist.

- At the bottom level, Cyclist defines the DOWN() and UP() methods.
  - The DOWN() method absorbs one block of input, by bitwise adding it to the state, together with a *color* c_D in the last byte of the state. In this case, the color is a value that provides domain separation between the first block of ABSORB() (‘`01`‘ in hash mode or ‘`03`‘ in keyed mode), the first block of ABSORBKEY() (‘`02`‘) and any other block.
  - The UP() method aims at producing one block of output. It absorbs a color c_U, calls the underlying permutation f and returns the first bytes of the state. Here the color provides domain separation between a block of keystream (‘`80`‘), the first block of SQUEEZE() (‘`40`‘), the first block of SQUEEZEKEY() (‘`20`‘), a ratchet (‘`10`‘) and anything else.
  - The phase attribute keeps track of whether the last internal call was a UP() or a DOWN().
- At the next level, the ABSORBANY() method aims at absorbing input. It cuts the input string in blocks of given length (r parameter) and calls DOWN() and UP() appropriately. Similarly, SQUEEZEANY() provides output of a given length (ℓ parameter). Finally, the CRYPT() method alternates calls to UP() to produce keystream and to DOWN() to absorb the plaintext. The R_{absorb} attribute contains the block size when absorbing, which is R_{kin} in keyed mode or R_{hash} in hash mode. Similarly, the R_{squeeze} attribute contains the block size when squeezing, which is R_{kout} in keyed mode or R_{hash} in hash mode.
- At the top level, the public methods rely on ABSORBANY(), CRYPT() and SQUEEZEANY().

Algorithm 1 Definition of $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$

```
Instantiation: cyclist ← $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}](K, \text{id}, \text{counter})$
    Phase and state: $(\textsc{phase}, s) \leftarrow (\text{up}, \text{‘00‘}^{b'})$
    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 $\textsc{AbsorbKey}(K, \text{id}, \text{counter})$

Interface: $\textsc{Absorb}(X)$
    $\textsc{AbsorbAny}(X, R_{\text{absorb}}, \text{‘03‘ (absorb)})$

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

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

Interface: $Y \leftarrow \textsc{Squeeze}(\ell)$
    return $\textsc{SqueezeAny}(\ell, \text{‘40‘ (squeeze)})$

Interface: $Y \leftarrow \textsc{SqueezeKey}(\ell)$, with $\textsc{mode} = \text{keyed}$
    return $\textsc{SqueezeAny}(\ell, \text{‘20‘ (key)})$

Interface: $\textsc{Ratchet}()$, with $\textsc{mode} = \text{keyed}$
    $\textsc{AbsorbAny}(\textsc{SqueezeAny}(\ell_{\text{ratchet}}, \text{‘10‘ (ratchet)}), R_{\text{absorb}}, \text{‘00‘})$
```

Algorithm 2 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 $\textsc{phase} \neq$ 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 $|K \,||\, \text{id}| \leq R_{\text{kin}} - 1$
  $(\textsc{mode}, R_{\text{absorb}}, R_{\text{squeeze}}) \leftarrow (\text{keyed}, R_{\text{kin}}, R_{\text{kout}})$
  $\textsc{AbsorbAny}(K \,||\, \text{id} \,||\, \text{enc}_8(|\text{id}|), R_{\text{absorb}}, \text{‘02‘ (key)})$
  if counter not empty then $\textsc{AbsorbAny}(\text{counter}, 1, \text{‘00‘})$

Internal interface: $O \leftarrow \textsc{Crypt}(I, \textsc{decrypt})$
  for all blocks $I_i$ in $\textsc{Split}(I, R_{\text{kout}})$ do
    $O_i \leftarrow I_i \oplus \textsc{Up}(|I_i|, \text{‘80‘ (crypt)}$ if first block else ‘00‘$)$
    $P_i \leftarrow O_i$ if $\textsc{decrypt}$ 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
    $\textsc{Down}(\epsilon, \text{‘00‘})$
    $Y \leftarrow Y \,||\, \textsc{Up}(\min(\ell - |Y|, R_{\text{squeeze}}), \text{‘00‘})$
  return $Y$

Internal interface: $\textsc{Down}(X_i, c_D)$
  $(\textsc{phase}, s) \leftarrow (\text{down}, s \oplus (X_i \,||\, \text{‘01‘} \,||\, \text{‘00‘}^* \,||\, (c_D \,\&\, \text{‘01‘}$ if $\textsc{mode} = \text{hash}$ else $c_D)))$

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

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 [MRH04], as well as properties that cannot be modeled as a single-stage game [RSS11].*

This means that Xoodyak hashing has essentially the same claimed security as, e.g., SHAKE128 [NIS15].

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* $\text{Cyclist}(K_i, \cdot, \cdot)$ *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 full-state keyed duplex [DMA17], 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 (i.e., the first* R_{kout} *bytes of the state) and for which the adversary gets a subsequent output block. In particular, this counts the number of blocks processed by* $\text{Decrypt}(\cdot)$ *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* $\text{Ratchet}()$ *followed by* $\text{Squeeze}(\ell)$ *or* $\text{SqueezeKey}(\ell)$ *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* $\text{Absorb}()$ *follows a call to* $\text{Squeeze}(\ell)$ *or to* $\text{SqueezeKey}(\ell)$ *with* $\ell \neq 0$.
- $q_{\text{iv}} \leq u$ *is the maximum number of keys that are used with the same* id*, i.e.,*

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

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, as an extendable output function (XOF) or most generally as a doubly extendable cryptographic (or *dec*) function. Alternatively, one can start Xoodyak in keyed mode and, e.g., to use it as a doubly-extendable cryptographic keyed (or *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 a 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:

```
Cyclist($\epsilon, \epsilon, \epsilon$)
Absorb($x$)
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*

By using consecutive calls to Absorb(), Xoodyak can hash not just one string, but potentially a sequence of strings. For instance, to compute a n-byte digest over the sequence $x_3 \circ x_2 \circ x_1$, one does the following:

```
Cyclist($\epsilon, \epsilon, \epsilon$)
Absorb($x_1$)
Absorb($x_2$)
Absorb($x_3$)
Squeeze($n$)
```

The output depends on the sequence as such and not just on the concatenation of the different strings. In this respect Xoodyak is therefore similar to TupleHash [NIS16].

Most generally, Xoodyak enjoys incrementality properties on both its input and its output: Appending a string to the input sequence costs only the processing of the new string, and requesting more output bits costs only the production of these new bits. Hence, we say that Xoodyak implements a dec function [DHAK18b].

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 Squeeze(n) || Squeeze(m) and 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 Squeeze(n) followed by SqueezeMore(m) is equivalent to calling Squeeze($n + m$) in the first place.

3.2 Keyed mode

In keyed mode, XOODYAK can naturally implement a deck function [DHAK18b], although we focus instead on duplex-based ways to perform authentication and (authenticated) encryption [BDPA11b].

To use XOODYAK as a keyed object, one starts it with $\text{CYCLIST}(K, \text{id}, \text{counter})$ 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 just 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 [Bih02]. Claim 2 reflects this in the term $\frac{q_{\text{iv}}N}{2^\kappa} \leq \frac{N}{2^{\kappa-\log_2 u}}$ as $q_{\text{iv}} \leq u$.

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 $\frac{q_{\text{iv}}N}{2^\kappa}$ becomes
$$\frac{q_{\text{iv}}N}{2^{128+\log_2 u}} \leq \frac{N}{2^{128}} \ .$$
- We make the key identifier id globally unique among the u keys and therefore ensure that $q_{\text{iv}} = 1$. 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 $\text{CYCLIST}()$ 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 [TS14]. 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 $\text{ABSORB}(\text{nonce})$ just after $\text{CYCLIST}(K, \text{id}, \epsilon)$.

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 $q_{\text{iv}} = 1$ 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, $\epsilon$)
Absorb(nonce)
Absorb($A$)
$C \leftarrow$ Encrypt($P$)
$T \leftarrow$ Squeeze($t$)
return $(C, T)$
```

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

```
Cyclist($K$, id, $\epsilon$)
Absorb(nonce)
Absorb($A$)
$P \leftarrow$ Decrypt($C$)
$T' \leftarrow$ Squeeze($t$)
if $T = T'$ then
  return $P$
else
  return $\perp$
```

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, $\epsilon$)
Absorb(nonce)
Absorb($A_1$)
$C_1 \leftarrow$ Encrypt($P_1$)
```

```
$T_1 \leftarrow \textsc{Squeeze}(t)$
  $\Rightarrow$ output $(C_1, T_1)$ and wait for next message
$\textsc{Absorb}(A_2)$
$C_2 \leftarrow \textsc{Encrypt}(P_2)$
$T_2 \leftarrow \textsc{Squeeze}(t)$
  $\Rightarrow$ 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() to mitigate the impact of recovering the internal state, e.g., after a side channel attack.

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.

```
$\textsc{Cyclist}(K, \text{id}, \epsilon)$
$\textsc{Absorb}(\text{nonce})$
$\textsc{Absorb}(A)$
$C \leftarrow \textsc{Encrypt}(P)$
$\textsc{Ratchet}()$ {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($\ell_{\text{sub}}$) {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:

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

CYCLIST($K_D, \epsilon, \epsilon$)
ABSORB(nonce)
ABSORB($A$)
$C \leftarrow$ ENCRYPT($P$)
$T \leftarrow$ SQUEEZE($t$)
return ($C, T$)
```

Note that if $\ell \leq R_{\text{hash}}$, an implementation can efficiently chain $K_D \leftarrow$ SQUEEZE(ℓ) and the subsequent reinitialization CYCLIST(K_D, ϵ, ϵ). Since K_D is located in the outer part (i.e., the first R_{hash} bytes) 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” [BDPA11d]. Putting aside definitional issues, the hermetic sponge strategy described in [BDPA11d] targets the absence of distinguishers on the permutation 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 generic 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 expressed on top of the duplex construction [BDPA11b] with simple padding as recalled in Algorithm 3. The state is initialized to the all-zero string like Cyclist, and a duplexing call corresponds to a sequence of DOWN() and UP().

The rate is set to $r = 8R_{\text{hash}} + 2$ bits, which amounts to the input block size plus two additional bits that can be potentially controlled by the attacker, namely the padding bit and one color bit (i.e., c_D & ‘01‘) via DOWN() in Algorithm 2. Note that, in the duplex construction, the input σ and its padding affect only the first r bits, whereas Cyclist’s color bit is in the last byte of the state. This could be formalized by permuting the bit positions before and after f, i.e., by running the duplex construction on $f' = \pi^{-1} \circ f \circ \pi$ with π a bit transposition such that all the input bits are in the first r positions, and this would not affect the generic security.

Algorithm 3 The duplex construction DUPLEX$[f, \text{pad}10^*, r]$

```
**Instantiation:** duplex ← DUPLEX$[f, \text{pad}10^*, r]$
  State: $s \leftarrow 0^b$

**Interface:** $Z \leftarrow \text{duplexing}(\sigma, \ell)$ with $\ell \leq r$, $\sigma \in \bigcup_{n=0}^{r-1} \mathbb{Z}_2^n$, and $Z \in \mathbb{Z}_2^\ell$
  $s \leftarrow s \oplus (\sigma || 10^*)$
  $s \leftarrow f(s)$
  **return** $\lfloor s \rfloor_\ell$
```

Using the Duplexing-Sponge Lemma [BDPA11b], the generic security of the duplex construction can be reduced to that of the sponge construction with a capacity of $c = b - r$, so $c = 254$ bits in the case of XOODYAK. Consequently, we make a flat sponge claim [BDPA11a] 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 full-state keyed duplex (KD), as recalled in Algorithm 4. Here the rate is $r = 8R_{\text{kout}}$ and $c = b - r$, so $c = 192$ bits in the case of XOODYAK. A crucial property of the KD 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, as depicted in Figure 1. 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 KD. A similar comment applies to ABSORBANY(), possibly except the first iteration.

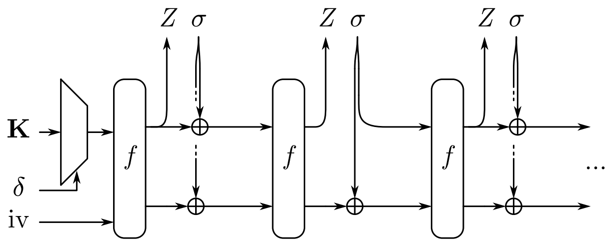

Figure 1: The full-state keyed duplex construction.

Algorithm 4 Full-state keyed duplex construction $\mathrm{KD}^f_{\mathbf{K}=(K_0,\ldots,K_{u-1})}$ with $\forall i, K_i \in \mathbb{Z}_2^\kappa$

```
Interface: $Z \leftarrow$ KD.Init($K_\delta$, iv, $\sigma$, flag) with $\delta \in \mathbb{Z}_u$, iv $\in \mathbb{Z}_2^{b-\kappa}$, $\sigma \in \mathbb{Z}_2^b$, flag $\in$ {true, false}, and $Z \in \mathbb{Z}_2^r$
    $s \leftarrow f(K_\delta || \mathrm{iv})$
    $Z \leftarrow$ the first $r$ bits of $s$
    if flag = true then $\sigma \leftarrow \sigma \oplus (Z || 0^*)$
    $s \leftarrow s \oplus \sigma$
    return $Z$

Interface: $Z =$ KD.Duplexing($\sigma$, flag) with $\sigma \in \mathbb{Z}_2^b$, flag $\in$ {true, false}, and $Z \in \mathbb{Z}_2^r$
    $s \leftarrow f(s)$
    $Z \leftarrow$ the first $r$ bits of $s$
    if flag = true then $\sigma \leftarrow \sigma \oplus (Z || 0^*)$
    $s \leftarrow s \oplus \sigma$
    return $Z$
```

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 KD setting [DMA17], 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 KD 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 KD paper [DMA17], 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 [DMA17] that $2^{(r-c)/2} < M \leq 2^{r-1}$ implies $\nu_{r,c}^{2(M-L)} \leq \nu_{r,c}^{2M} \leq b/4$. We 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.
- $H_{\min(\mathcal{D}_\mathrm{K})} = H_{\mathrm{coll}}(\mathcal{D}_\mathrm{K}) = \kappa$ in our setting, so $\frac{(M-q-L)q}{2^{H_{\min(\mathcal{D}_\mathrm{K})}+\min(c,b-k)}}$ is upper bounded as $\frac{Mq}{2^{\min(c+\kappa,b)}}$, $\frac{q_{\mathrm{iv}}N}{2^{H_{\min(\mathcal{D}_\mathrm{K})}}}$ as $\frac{uN}{2^\kappa}$ (since $q_{\mathrm{iv}} \leq u$), and $\frac{\binom{u}{2}}{2^{H_{\mathrm{coll}}(\mathcal{D}_\mathrm{K})}} = \frac{\binom{u}{2}}{2^\kappa}$.

4.2.3 Decodability

In this section, we show the following lemma, which means that any output of Cyclist unambiguously depends on the process history. For the keyed mode, in particular, this links the Cyclist mode to the full-state keyed duplex construction [DMA17].

Lemma 1. *From 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.*

Proof. First let us observe that any sequence of calls to the Cyclist object is translated internally into an alternating sequence of $\textsc{Down}(X_i, c_D)$ and $\textsc{Up}(|Y_i|, c_U)$ 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 2. 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 ($\ell > R_{\text{hash}}$). □

Table 2: Matching up and down sub-sequences with process history. Here $u()$ and $d()$ are shortcut notations for Up() and Down(), respectively.

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

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 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 [DHAK18a] 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 [Dae95], 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 [SD18]. 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 [WT85]. Like Keccak-p, it has so-called weak alignment [BDPA11c], 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 (see below). Note that the weight of a trail relates to its differential probability or its correlation, see [DHAK18a, Section 5.2] for more details.

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 [BDP+19]. 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 [SLG17]. Note that in hashing mode, Xoodyak has a much smaller rate (or block size), 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 is used for Keccak-p in Keyak [BDP+16], also relying on the full-state keyed duplex construction.

4.3.1 Extended trail analysis

Since the publication of Xoodoo, we have extended the trail analysis and improved the bounds. Table 3 shows the currently known lower bounds. In particular, we improved upon [DHAK18a, Table 7] for 4 and 5 rounds.

Let us first summarize the notions related to differential and linear trails. For more details, please refer to [DHAK18a, Sections 5.2 and 7].

We split the round function into the nonlinear layer χ and the sequence of linear mappings $\lambda = \rho_{\text{west}} \circ \theta \circ \rho_{\text{east}}$. An n-round differential trail is the concatenation of n round differentials (a_i, a_{i+1}), with a_i the difference at the output of χ after i rounds, and is fully specified by the sequence $(a_0, a_1, \ldots, a_n)$. We use a redundant representation of trails, where we also include the differences after the linear layer: $Q = (a_0, b_0, a_1, b_1 \ldots, b_{n-1}, a_n)$, with $b_i = \lambda(a_i)$. We define the restriction weight $\mathrm{w_r}(b_i \to a_{i+1})$ as $\mathrm{DP}(b_i \to a_{i+1}) = 2^{-\mathrm{w_r}(b_i \to a_{i+1})}$. As shown in [DHAK18a], $\mathrm{w_r}(b_i \to a_{i+1})$ is equal to 2 times the number of active columns in b_i or a_{i+1}, and this number is preserved through χ, so we can write $\mathrm{w_r}(b_i \to a_{i+1}) = \mathrm{w_r}(b_i) = \mathrm{w_r}(a_{i+1})$.

We define the weight of a trail Q as $\mathrm{w_r}(Q) = \sum_{i=0}^{b-1} \mathrm{w_r}(b_i) = \mathrm{w_r}(a_1) + \sum_{i=1}^{b-1} \mathrm{w_r}(b_i)$. Consequently, the evaluation of $\mathrm{w_r}(Q)$ do not require the value of a_0, b_0 or a_n. To bound the weight of trails, we can therefore restrict our attention to *differential trail cores*, as in [DA12]:

$$Q = a_1 \xrightarrow{\lambda} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda} b_2 \xrightarrow{\chi} a_3 \xrightarrow{\lambda} \ldots a_{n-1} \xrightarrow{\lambda} b_{n-1} \ .$$

For linear trails, the reasoning is similar, but differences are replaced with masks, the restriction weight is replaced with the correlation weight, and the operations are taken in reverse order and transposed [DHAK18a]. The correlation weight of an n-round linear trail $(a_0, b_0, a_1, b_1, \ldots, b_{n-1}, a_n)$ is given by $\mathrm{w_c}(a_1) + \sum_{i=1}^{n-1} \mathrm{w_c}(b_i)$, with $\mathrm{w_c}(u)$ denoting the correlation weight of a mask u. This is a *linear trail core*:

$$Q = a_1 \xrightarrow{\lambda^\top} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda^\top} b_2 \xrightarrow{\chi} a_3 \xrightarrow{\lambda^\top} \ldots a_{n-1} \xrightarrow{\lambda^\top} b_{n-1} \ .$$

Due to the high similarity of the trail search between differential and linear trails, we discuss them generically, and use the notation $\mathrm{w}(\cdot)$ to denote either $\mathrm{w_r}(\cdot)$ or $\mathrm{w_c}(\cdot)$.

- For 4 rounds, we exhaustively covered the space of differential and linear trail cores with weight $\mathrm{w}(a_1) + \mathrm{w}(b_1) + \mathrm{w}(b_2) + \mathrm{w}(b_3) \leq 72$. Since no such trail core was found and since all weights are even, a 4-round trail must have weight at least 74. We divided the search into two parts. First, we generated 2-round trail cores with weight $\mathrm{w}(a_1) + \mathrm{w}(b_1) \leq 38$ and extended them forward to 4 rounds up to weight

Table 3: 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	8	10	12
differential:	2	8	36	≥ 74	≥ 94	≥ 104	≥ 148	≥ 188	≥ 222
linear:	2	8	36	≥ 74	≥ 94	≥ 104	≥ 148	≥ 188	≥ 222

72. Second, when $\mathrm{w}(a_1) + \mathrm{w}(b_1) > 38$, we have $\mathrm{w}(b_2) + \mathrm{w}(b_3) = \mathrm{w}(a_3) + \mathrm{w}(b_3) \leq 32$. So we generated 2-round trail cores with weight $\mathrm{w}(a_3) + \mathrm{w}(b_3) \leq 32$ and extended them backwards to 4 rounds up to weight 72. The unbalance between the two parts compensates for the higher cost of the backward extension compared to the forward extension.

- For 5 rounds, we exhaustively covered the space of differential and linear trail cores with weight $\mathrm{w}(a_1) + \mathrm{w}(b_1) + \mathrm{w}(b_2) + \mathrm{w}(b_3) + \mathrm{w}(b_4) \leq 92$. Since no such trail core was found, a 5-round trail must have weight at least 94. We again divided the search into two parts. First, we generated 2-round trail cores with weight $\mathrm{w}(a_1)+\mathrm{w}(b_1) \leq 40$ and extended them forward to 5 rounds up to weight 92. Second, when $\mathrm{w}(a_1)+\mathrm{w}(b_1) > 40$, we have $\mathrm{w}(b_2) + \mathrm{w}(b_3) + \mathrm{w}(b_4) \leq 50$, and this coincides with the set of 3-round trail cores produced in [DHAK18a]. So we just took these trail cores and extended them backwards to 5 rounds up to weight 92.

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 is 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.

In [SG18], Song et al. mounted cube attacks on a Xoodoo-based authenticated encryption scheme following the same mode as Ketje. 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. 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.

In [ZLD+19], Zhou et al. mounted a conditional cube attack on Xoodyak with the permutation reduced to 6 rounds (out of the nominal 12). In the nonce-misuse setting, their attack could recover the 128-bit key in about 2^{44} operations with negligible memory costs.

From these two research papers, we can deduce that 12 rounds provide enough safety margin against this type of attacks.

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 (or block sizes) 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 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

The purpose of the NIST Lightweight Cryptography Standardization Process [DHP+19] is to get algorithms that are suitable for use in constrained environments. In this section, we discuss the implementation of Xoodyak in the light of this aspect and report on implementations on ARM Cortex-M0 and Cortex-M3. For more details on the implementation of the Xoodoo permutation, we refer to [DHAK18a, Section 4].

5.1 Lightweight?

It is a reasonable question to ask whether a scheme like Xoodyak, based on a 384-bit permutation, can be called lightweight.

Part of the answer comes from the use of a permutation-based construction like duplex. Even if, say, the AES-128 block cipher has only a data path of 128 bits, one must take into account an extra 128 bits for its key schedule, plus the memory needed for whatever mode of operation on top the block cipher. In contrast, all the use-cases of Xoodyak rely on the duplex construction having a state whose size is equal to the width of the permutation, and it needs almost no additional state for the modes on top. The Xoodoo permutation can be computed essentially in-place (see [DHAK18a, Section 4.1]), and the input and output are added to and extracted directly from the state, without the need of a message queue. Even if the interface allows the user to enter inputs and extract outputs incrementally, everything fits in the 384 bits of the permutation plus a couple of pointers for bookkeeping. In a different (i.e., non-lightweight) context, Yalla et al. showed the benefits of such a construction in terms of area in a hardware implementation [YHK15]. Not surprisingly, many candidates to the NIST Lightweight Cryptography Standardization Process adopted the same strategy [DHP+19].

Another aspect is the permutation itself and the choice of operations in it. Xoodoo has a lot of symmetry, making it possible to reuse parts of the circuit or the code. The operations are all simple, making use of logical XORs and ANDs, and we deliberately avoided modular additions with their long carry chains. Finally, Xoodyak comes with several features to protect against side-channel attacks, something that is otherwise very costly to achieve.

5.2 Software implementation results

We implemented Xoodyak on ARM Cortex-M0 and -M3 processors. On Cortex-M0, the implementation of Xoodoo[12] executes one round in a loop. Being a compact processor, the focus was more on compactness than on speed. In contrast, the implementation on

Table 4: 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

Cortex-M3 unrolls the 12 rounds of the permutation, hence maximizing the speed by taking advantage of the implicit rotations in parallel with the other operations. Note that we could also unroll the implementation on Cortex-M0 and/or make a more compact implementation on Cortex-M3, but these give interesting extreme points. These implementations are available in the extended Keccak code package (XKCP) [VKC20].

We report on the computational performance of Xoodyak in Table 4. We compared this with some other candidates, namely, Ascon [DEMS19], Gimli [BKL+17] and Schwaemm & Esch based on the Sparkle permutation [BBdS+19]. Note that we did not run a benchmark of these functions, but instead based ourselves on the information published by the designers.

- For Ascon, the NIST submission document does not report speed on the same platform, but on Cortex-A7 (ARMv7) the authenticated encryption schemes Ascon-128 and Ascon-128a run at 57.2 and 41.2 cycles/byte, respectively. Assuming that a Cortex-A7 is at least as fast as a Cortex-M3, this shows that Xoodyak is faster than Ascon. The hash function Ascon-Hash has twice as many rounds as Ascon-128, so we expect it will be about twice slower as well. The difference in speed can be explained by the sizes of the permutations: The Ascon permutation is 320-bit wide, hence leaving less room than Xoodyak for the input and output blocks at a given capacity. However, as Ascon is defined on 64-bit words, we expect it to be faster than Xoodyak on 64-bit machines.

- The paper on Gimli reports on speed of the permutation only. As they use 128 bits of rate for both absorbing and squeezing, we can approximate the speed of the Gimli hash and authenticated encryption schemes by multiplying the number of cycles per byte by three. Doing so gives us 147 cycles/byte on Cortex-M0 and 63 cycles/byte on Cortex-M3, which are both slower than Xoodyak. Here the difference is likely due to the number of rounds, namely 24 for Gimli vs 12 for Xoodyak. Although Gimli has faster rounds, this does not compensate for the higher number of rounds.

- The Sparkle submission reports speed of Schwaemm and Esch on Cortex-M3. We compare with the instances that use the Sparkle384 permutation, with the same width as Xoodoo. For the Schwaemm256-128 authenticated encryption scheme, the speed is 46 cycles/byte on this platform, while for the Esch256 hash function, the authors report 66 cycles/byte. In both cases, Xoodyak is faster on this platform.

In terms of memory consumption, our implementation maintains a state of size the width of the permutation (48 bytes) together with 4 bytes to store the phase, the mode, R_{absorb} and R_{squeeze}, so 52 bytes in total.

Table 5: Code sizes (in bytes) of our implementations of Xoodyak and of the permutation only.

	Xoodyak	of which Xoodoo[12]	Notes
ARM Cortex-M0	3494	468	1 round in a loop
ARM Cortex-M3	4058	2388	12 rounds unrolled

Finally, we report in Table 5 the sizes of the code of these two implementations. We also report the code size taken by the permutation only. Naturally, the size of the unrolled permutation is significantly larger than the other one. As a comparison, the Sparkle384 permutation takes 484 bytes [BBdS+19] on Cortex-M3, which shows that Xoodoo's code size is competitive when it is not unrolled. Note that an unrolled implementation of the Gimli permutation takes 3950 bytes on Cortex-M3 [BBdS+19]. At the time of this writing, we are experimenting with code size optimizations and we expect to reach around 1200 bytes of code for the full Xoodyak with similar speeds.

6 Conclusion

In this paper, we have presented the Xoodyak lightweight cryptographic scheme that we submitted to the NIST Lightweight Cryptography Standardization Process [DHP+19]. We have shown through a variety of examples that, thanks to its duplex-based mode of operation Cyclist, it is a versatile primitive, covering use cases such as hashing, authentication and session-supporting authenticated encryption. Its features make it suitable for implementations that must protect against side-channel attacks.

Under the hood, the Xoodoo permutation has an excellent security strength build-up per operation count, and this is visible in the diffusion properties and trail bounds. In particular, we have extended the trail analysis in this paper and have improved the bounds for 4 and 5 rounds.

Acknowledgments

We thank Frederik Armknecht and anonymous reviewers of ToSC editorial board for their suggestions improving the editorial quality of this paper. Joan Daemen is supported by the European Research Council under the ERC advanced grant agreement under grant ERC-2017-ADG Nr. 788980 ESCADA.

References

[BBdS+19] C. Beierle, A. Biryukov, L. Cardoso dos Santos, J. Großschädl, L. Perrin, A. Udovenko, V. Velichkov, and Q. Wang. Sparkle (Schwaemm and Esch). Submission to NIST Lightweight Cryptography Standardization Process (round 2), March 2019. https://www.cryptolux.org/index.php/Sparkle.

[BDP+16] 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.

[BDP+19] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. Keccak third-party cryptanalysis, 2019. https://keccak.team/third_party.html.

[BDPA11a] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions, January 2011. https://keccak.team/papers.html.

[BDPA11b] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In A. Miri and S. Vaudenay, editors, *Selected Areas in Cryptography - SAC 2011, Revised Selected Papers*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.

[BDPA11c] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On alignment in Keccak. In *ECRYPT II Hash Workshop*, 2011.

[BDPA11d] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, January 2011. https://keccak.team/papers.html.

[Bih02] E. Biham. How to decrypt or even substitute des-encrypted messages in 2^{28} steps. *Inf. Process. Lett.*, 84(3):117–124, 2002.

[BKL$^+$17] 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. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[DA12] J. Daemen and G. Van Assche. Differential propagation analysis of Keccak. In A. Canteaut, editor, *Fast Software Encryption (FSE) 2012. Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*, pages 422–441. Springer, 2012.

[Dae95] J. Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis, PhD Thesis.* K.U.Leuven, 1995.

[DEMS19] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer. Ascon v1.2. Submission to NIST Lightweight Cryptography Standardization Process (round 2), March 2019. https://ascon.iaik.tugraz.at/.

[DHAK18a] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. The design of Xoodoo and Xoofff. *IACR Trans. Symmetric Cryptol.*, 2018(4):1–38, 2018.

[DHAK18b] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. Xoodoo cookbook. *IACR Cryptology ePrint Archive*, 2018:767, 2018.

[DHP$^+$19] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. Xoodyak, a lightweight cryptographic scheme. Submission to NIST Lightweight Cryptography Standardization Process (round 2), March 2019. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Xoodyak-spec-round2.pdf.

[DMA17] J. Daemen, B. Mennink, and G. Van Assche. Full-state keyed duplex with built-in multi-user support. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 606–637. Springer, 2017.

[Ham17] M. Hamburg. The STROBE protocol framework. In *Real World Crypto*, 2017.

[MRH04] U. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In M. Naor, editor, *Theory of Cryptography - TCC 2004*, number 2951 in Lecture Notes in Computer Science, pages 21–39. Springer-Verlag, 2004.

[MRV15] B. Mennink, R. Reyhanitabar, and D. Vizár. Security of full-state keyed sponge and duplex: Applications to authenticated encryption. In T. Iwata and J. H. Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015, New Zealand, 2015*, volume 9453 of *LNCS*, pages 465–489. Springer, 2015.

[NIS15] 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`.

[NIS16] NIST. 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`.

[Per18] T. Perrin. Stateful hash objects: API and constructions. `https://github.com/noiseprotocol/sho_spec/blob/master/output/sho.pdf`, 2018.

[RSS11] T. Ristenpart, H. Shacham, and T. Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In K. G. Paterson, editor, *Eurocrypt 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2011.

[Saa14] M.-J. O. Saarinen. Beyond modes: Building a secure record protocol from a cryptographic sponge permutation. In J. Benaloh, editor, *Topics in Cryptology - CT-RSA 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 270–285. Springer, 2014.

[SD18] K. Stoffelen and J. Daemen. Column parity mixers. *IACR Trans. Symmetric Cryptol.*, 2018(1):126–159, 2018.

[SG18] L. Song and J. Guo. Cube-attack-like cryptanalysis of round-reduced Keccak using MILP. *IACR Trans. Symmetric Cryptol.*, 2018(3):182–214, 2018.

[SLG17] L. Song, G. Liao, and J. Guo. Non-full sbox linearization: Applications to collision attacks on round-reduced Keccak. In J. Katz and H. Shacham, editors, *Advances in Cryptology - CRYPTO 2017*, volume 10402 of *Lecture Notes in Computer Science*, pages 428–451. Springer, 2017.

[TS14] M. M. I. Taha and P. Schaumont. Side-channel countermeasure for SHA-3 at almost-zero area overhead. In *2014 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2014*, pages 93–96. IEEE Computer Society, 2014.

[VKC20] G. Van Assche, R. Van Keer, and Contributors. Extended Keccak code package, February 2020. `https://github.com/XKCP/XKCP`.

[WT85] A. F. Webster and S. E. Tavares. On the design of S-boxes. In H. C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 523–534. Springer, 1985.

[YHK15] P. Yalla, E. Homsirikamol, and J.-P. Kaps. Comparison of multi-purpose cores of Keccak and AES. In *Design, Automation Test in Europe DATE 2015*, pages 585–588. ACM, March 2015.

[ZLD^{+}19] H. Zhou, Z. Li, X. Dong, K. Jia, and W. Meier. Practical key-recovery attacks on round-reduced Ketje Jr, Xoodoo-AE and Xoodyak. *IACR Cryptology ePrint Archive*, 2019:447, 2019.