Xoodoo cookbook

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

[1] STMicroelectronics
[2] Radboud University

Abstract. This document presents XOODOO, a 48-byte cryptographic permutation that allows very efficient symmetric crypto on a wide range of platforms and a suite of cryptographic functions built on top of it. The central function in this suite is XOOFFF, obtained by instantiating Farfalle with XOODOO. XOOFFF is what we call a *deck function* and can readily be used for MAC computation, stream encryption and key derivation. The suite includes two session authenticated encryption (SAE) modes: XOOFFF-SANE and XOOFFF-SANSE. Both are built on top of XOOFFF and differ in their robustness with respect to nonce misuse. Other members of the suite are a tweakable wide block cipher XOOFFF-WBC and authenticated encryption mode XOOFFF-WBC-AE, obtained by instantiating the Farfalle-WBC and Farfalle-WBC-AE constructions with XOOFFF. Finally, for lightweight applications, we define XOODYAK, a cryptographic scheme 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. This paper is a specification and security claim reference for the XOODOO suite. It is a *standing document*: over time, we may extend the XOODOO suite, and we will update it accordingly.

Date: 14 March 2019

Keywords: permutation-based crypto · Farfalle · duplex construction · dec functions · hashing · deck functions · authenticated encryption

1 Introduction

In [2] we presented new parallel modes of use of permutations for encryption, authentication, session authenticated encryption and wide block ciphers under the umbrella name Farfalle. We also proposed concrete instantiations called KRAVATTE by plugging in the KECCAK-$p[1600, n_\mathrm{r}]$ permutation with 6 rounds. All-over, KRAVATTE is very fast on a wide range of platforms but can hardly be called lightweight: it operates on a large state giving rise to considerable overhead on low-end CPUs and has for short inputs a relatively large overhead per byte.

It therefore makes sense to consider instantiating Farfalle with a smaller permutation, somewhere between 256 and 400 bits wide. Taking KECCAK-$p[400, n_\mathrm{r}]$ is problematic as it is defined in terms of operations on 16-bit *lanes*. The permutation Gimli [1] has the nice feature that it has a state of 384 bits and a round function that lends itself to low-end 32-bit CPUs but also vectorization and dedicated hardware. Unfortunately, its propagation properties are less than what could be expected. For constructing a Farfalle instance with 128-bit security strength one would have to take a relatively high number of rounds.

For that reason we took the initiative to design a permutation with the same width and objectives as Gimli, but with more favorable propagation properties. We called the result XOODOO and it can be seen as a porting of the KECCAK-p design approach to a Gimli-shaped state. In this document we specify this permutation and a suite of cryptographic functions built as modes of Xoodoo. This suite covers the symmetric-key crypto functions and we expect it to be very efficient on a wide range of CPUs and in dedicated hardware while having a comfortable safety margin. This makes the Xoodoo suite very competitive to, e.g., block cipher based crypto.

Xoofff is the central instance of the suite, covering keyed symmetric crypto functions. For more lightweight applications, we add Xoodyak to the suite, a compact and versatile cryptographic object that is suitable for most symmetric-key functions, including hashing, pseudo-random bit generation, authentication, encryption and authenticated encryption.

This document does not include extensive design rationale or analysis, nor does it provide performance benchmarks. For this, please see [8]. Its purpose is to expose the Xoodoo cipher suite to the cryptographic research community and security practicioners by serving as specification and security claim reference. At this stage the security claims serve as a challenge for cryptanalists and only a distant promise of security strength for users. This promise will gain in credibility as over time third-party cryptanalysis and public scrutiny accumulate.

Additionally, a reference implementation in C++ of Xoodoo and the members of the Xoodoo suite is available in [13] and optimized implementations in C and assembler in the *eXtended (or* Xoodoo *and)* Keccak *Code Package* (XKCP) [23].

Over time, we may add more cryptographic functions to the Xoodoo suite. When that is the case, we will update this document accordingly. In ISO/IEC terminology this would be called a *standing document*.

1.1 Notation

The set of all bit strings is denoted $\mathbb{Z}_2^*$ and ϵ is the empty string. The size in bits of the string X is denoted $|X|$. 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. Similarly, the set of all sequences containing at least one string is denoted $(\mathbb{Z}_2^*)^+$.

1.2 Deck functions

The central function in the Xoodoo suite is a Xoodoo-based Farfalle instance called Xoofff. Most other members of the suite are built as modes on top of Xoofff. Some of these modes are specified in [2] while others are introduced in this document. We follow the same naming conventions as in [2]. The name of the mode has two parts: a prefix indicating the underlying primitive type and a suffix referring to the target functionality. In instantiations with a particular primitive, we replace the prefix by the name of the primitive.

In our paper on Farfalle [2] we introduced the concept of a keyed cryptographic function with an extendable input and able to return an output of arbitrary length. In lack of a better name, we called these pseudorandom functions (PRF). We called the primitive type in our modes Farfalle as they needed support for sequences of strings as input and a specific incremental property, present in Farfalle instances: computing $F(Y \circ X)$ costs only the processing of Y if $F(X)$ was previously computed. Clearly, Farfalle is not the only way to build functions with such properties and we now think it would be better to decouple the input-output *signature* of the function (PRF with incremental sequence of strings input) from the implementation (Farfalle).

We decided to introduce the name *deck function* for a keyed function that takes a sequence of input strings and returns a pseudorandom string of arbitrary length and that can be computed incrementally. Here *deck* stands for *Doubly-Extendable Cryptographic Keyed* function.

Definition 1 ([8, 22])**.** A *deck function* takes as input a secret key K and a sequence of an arbitrary number of strings $X^{(m-1)} \circ \cdots \circ X^{(0)} \in (\mathbb{Z}_2^*)^+$, produces a potentially infinite string of bits and takes from it the range starting from a specified offset $q \in \mathbb{N}$ and for a specified length $n \in \mathbb{N}$. We denote this as

$$Z = 0^n + F_K\left(X^{(m-1)} \circ \cdots \circ X^{(0)}\right) \ll q\ .$$

A deck function should allow efficient incremental computing. In particular, by keeping state after computing an output for input sequence $X = X^{(m-1)} \circ \cdots \circ X^{(0)}$, computing an output for $Y^{(n-1)} \circ \cdots \circ Y^{(0)} \circ X$ should have a cost independent of X. In addition, by keeping state after compting $0^n + F_K\left(X^{(m-1)} \circ \cdots \circ X^{(0)}\right) \ll q$, computing $0^m + F_K\left(X^{(m-1)} \circ \cdots \circ X^{(0)}\right) \ll (q+n)$ should have a cost independent of n or q.

As such, we will indicate the modes we define this document and that we will instantiate with Xoofff by the prefix Deck. The modes on top of Farfalle specified in [2] may as well be renamed by replacing the prefix Farfalle by Deck. To avoid confusion, we will not do that.

1.3 Dec functions

Naturally, we can define the unkeyed equivalent of deck functions, namely *dec functions*, where *dec* stands for *Doubly-Extendable Cryptographic* function [22]. In short, a dec function is an extendable output function (XOF) that accepts sequences of strings as input and that supports incremental queries efficiently.

Definition 2 ([22])**.** A *dec function* takes as input a sequence of an arbitrary number of strings $X^{(m-1)} \circ \cdots \circ X^{(0)} \in (\mathbb{Z}_2^*)^+$, produces a potentially infinite string of bits and takes from it the range starting from a specified offset $q \in \mathbb{N}$ and for a specified length $n \in \mathbb{N}$. We denote this as

$$Z = 0^n + H\left(X^{(m-1)} \circ \cdots \circ X^{(0)}\right) \ll q\ .$$

Like a deck function, a dec function should allow efficient incremental computing.

1.4 Session authenticated encryption

In many use cases where one wishes confidentiality, authentication is required too and it makes sense to offer a scheme that provides both: an *authenticated encryption scheme.* Doing this with a deck function is simple: one enciphers the plaintext by adding to it the output of a deck function applied to a nonce and computes a tag on the ciphertext (and possibly metadata) also using the deck function.

Often, one does not only want to protect a single message, but rather a *session* where multiple messages are exchanged, such as in the Transport Layer Security (TLS) protocol [10] or the Secure Shell (SSH) protocol [24]. Examples of session authenticated encryption schemes are Keyak [5], Ketje [4] and Kravatte-SAE [2]. They require only a nonce at the startup of the session and each tag authenticates all messages already sent in the session.

We consider authenticated encryption of a message as a process that takes as input metadata A and plaintext P and that returns a cryptogram C and a tag T. We denote this operation by the term *wrapping* and the reverse operation of taking metadata A, a cryptogram C and a tag T and returning the plaintext P if the tag T is correct by the term *unwrapping.* We further consider the process of authenticating and encrypting a sequence of messages $(\overline{A,P}) = (A^{(1)}, P^{(1)}, A^{(2)}, \ldots, A^{(n)}, P^{(n)})$ in such a way that the authenticity is guaranteed not only on each (A, P) pair but also on the sequence received so far. This is further formalized in [3, Section 2.1].

We use the abbreviation SAE to indicate session authenticated encryption in general. The generic term SAE should not be confused with Farfalle-SAE, that is a particular SAE mode of a deck function specified in [2].

1.5 Overview

We specify the core of all functions in the Xoodoo suite, the Xoodoo$[n_\mathrm{r}]$ family of permutations, in Section 2. We depict in Figure 1 all suite members and their relations. They are the following:

- The Xoofff deck function, specified in Section 3. We obtain this deck function by instantiating Farfalle with Xoodoo and suitable rolling functions and make a security claim.
- The Xoofff-SANE SAE scheme, specified in Section 4. We obtain this by defining an SAE mode of deck functions called Deck-SANE and instantiate it with Xoofff. Xoofff-SANE relies on user-provided nonces for confidentiality.
- The Xoofff-SANSE SAE scheme, specified in Section 5. We obtain this by defining an SAE mode of deck functions called Deck-SANSE and instantiate it with Xoofff. Xoofff-SANSE is more robust against nonce misuse and realizes this by using the SIV mechanism.
- The Xoofff-WBC wide block cipher, specified in Section 6. We obtain this by instantiating Farfalle-WBC with Xoofff and Xoofffie, a variant of Xoofff whose purpose is solely to provide differential uniformity. We give security claims for Xoofff-WBC and a dedicated claim for Xoofffie. Finally, we define the Xoofff-WBC-AE authenticated encryption scheme by applying Farfalle-WBC-AE on top of Xoofff-WBC.
- The Xoodyak scheme, specified in Section 7. We obtain this by instantiating Cyclist with Xoodoo.

We make no security claims for the Xoofff-SANE and Xoofff-SANSE SAE schemes as their claimed security follows immediately from the security claim of Xoofff. Similarly, the claimed security of Xoofff-WBC-AE follows directly from the security claim of Xoofff-WBC. For Xoodoo$[n_\mathrm{r}]$ we also do not make security claims as it is not a cryptographic function per se, just a building block.

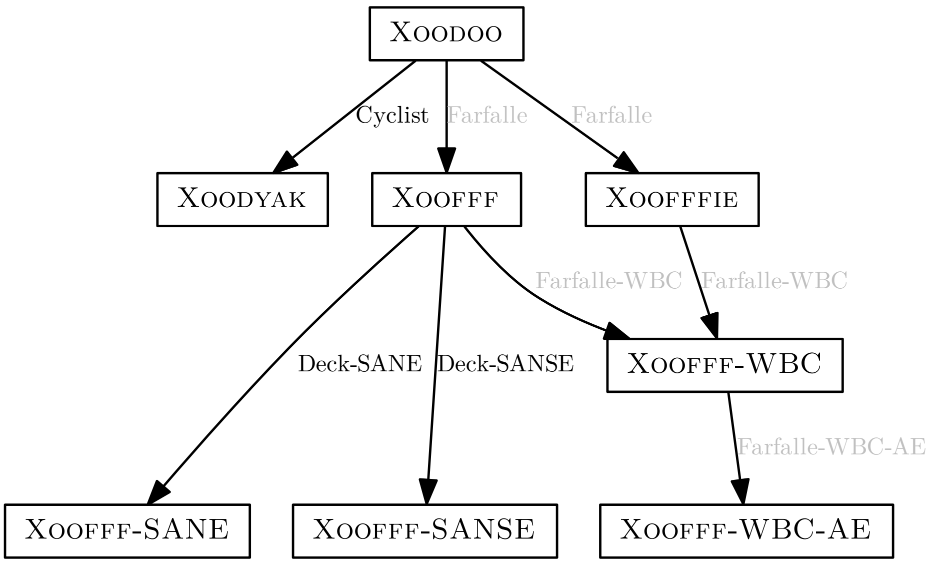

Figure 1: Overview of the Xoodoo suite, with the schemes in boxes and the modes indicated on the edges. All schemes and the modes in black print are specified in this document, the modes in grey are defined in [2].

2 Xoodoo

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 2.

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 3, the θ operation in Figure 4, the ρ_east and ρ_west operations in Figure 5.

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 $i = z + 32(x + 4y)$.

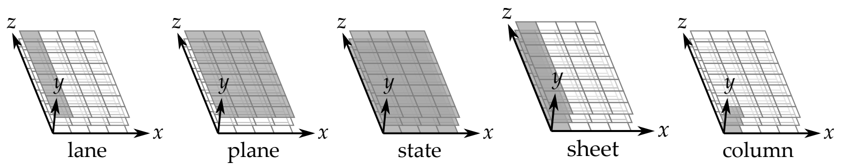

Figure 2: Toy version of the Xoodoo state, with lanes reduced to 8 bits, and different parts of the state highlighted.

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$[n_r]$ with n_r the number of rounds

```
**Parameters:** Number of rounds $n_r$
**for** Round index $i$ from $1 - n_r$ to 0 **do**
  $A = R_i(A)$

Here $R_i$ is specified by the following sequence of steps:
    $\theta$ :
        $P \leftarrow A_0 + A_1 + A_2$
        $E \leftarrow P \lll (1, 5) + P \lll (1, 14)$
        $A_y \leftarrow A_y + E$ for $y \in \{0, 1, 2\}$
    $\rho_{\text{west}}$ :
        $A_1 \leftarrow A_1 \lll (1, 0)$
        $A_2 \leftarrow A_2 \lll (0, 11)$
    $\iota$ :
        $A_0 \leftarrow A_0 + C_i$
    $\chi$ :
        $B_0 \leftarrow \overline{A_1} \cdot A_2$
        $B_1 \leftarrow \overline{A_2} \cdot A_0$
        $B_2 \leftarrow \overline{A_0} \cdot A_1$
        $A_y \leftarrow A_y + B_y$ for $y \in \{0, 1, 2\}$
    $\rho_{\text{east}}$ :
        $A_1 \leftarrow A_1 \lll (0, 1)$
        $A_2 \leftarrow A_2 \lll (2, 8)$
```

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

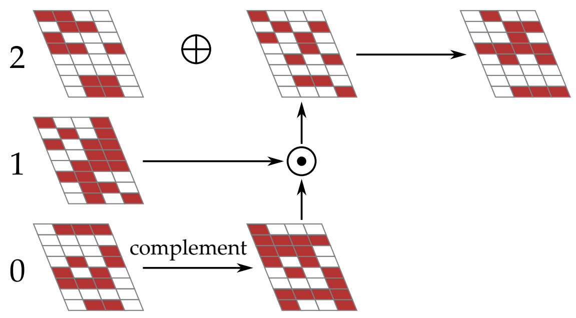

Figure 3: Effect of χ on one plane.

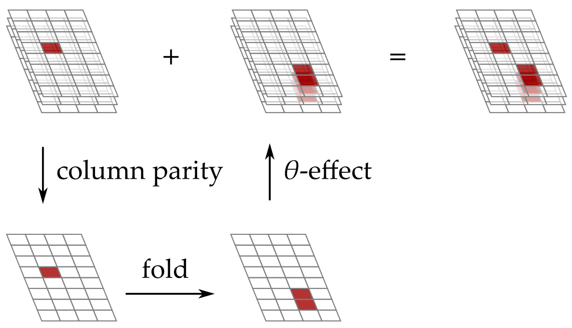

Figure 4: Effect of θ on a single-bit state.

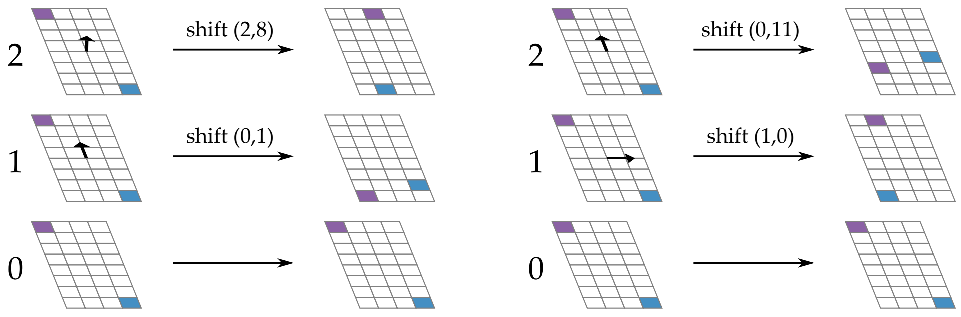

Figure 5: Illustration of ρ_{east} (left) and ρ_{west} (right).

Table 3: Notational conventions for specification of the rolling functions

$A_{y,x}$	Lane x of plane A_y
B	An auxiliary variable that has the shape of a plane
$A_{y,x} \lll v$	Cyclic shift of lane $A_{y,x}$ moving bit from x to $x+v$
$A_{y,x} \ll v$	Shift of lane $A_{y,x}$ moving bit from x to $x+v$, setting bits $x<v$ to 0
$A_{y,x} + A_{y',x'}$	Bitwise sum (XOR) of lanes $A_{y,x}$ and $A_{y',x'}$
$A_{y,x} \cdot A_{y',x'}$	Bitwise product (AND) of lanes $A_{y,x}$ and $A_{y',x'}$

3 Xoofff

Xoofff is a deck function obtained by applying the Farfalle construction on Xoodoo[6] and two rolling functions: roll_{Xc} for rolling the input masks and roll_{Xe} for rolling the state. We specify them with operations on the lanes of the state, following the conventions of Table 1 and Table 3.
The input mask rolling function roll_{Xc} updates a state A in the following way:

$$\begin{aligned}
A_{0,0} &\leftarrow A_{0,0} + (A_{0,0} \ll 13) + (A_{1,0} \lll 3)\\
B &\leftarrow A_0 \lll (3,0)\\
A_0 &\leftarrow A_1\\
A_1 &\leftarrow A_2\\
A_2 &\leftarrow B
\end{aligned}$$

The state rolling function roll_{Xe} updates a state A in the following way:

$$\begin{aligned}
A_{0,0} &\leftarrow A_{1,0} \cdot A_{2,0} + (A_{0,0} \lll 5) + (A_{1,0} \lll 13) + \texttt{0x00000007}\\
B &\leftarrow A_0 \lll (3,0)\\
A_0 &\leftarrow A_1\\
A_1 &\leftarrow A_2\\
A_2 &\leftarrow B
\end{aligned}$$

Definition 3 (Xoofff). Xoofff is Farfalle$[p_{\text{b}}, p_{\text{c}}, p_{\text{d}}, p_{\text{e}}, \text{roll}_{\text{c}}, \text{roll}_{\text{e}}]$ with the following parameters:

- $p_{\text{b}} = p_{\text{c}} = p_{\text{d}} = p_{\text{e}} =$ Xoodoo[6],
- $\text{roll}_{\text{c}} = \text{roll}_{\text{Xc}}$ and
- $\text{roll}_{\text{e}} = \text{roll}_{\text{Xe}}$.

We make the following security claim on Xoofff.

Claim 1. *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 $\kappa < 384$. Then, the advantage of distinguishing the array of functions* $\text{Xoofff}_{K_i}(\cdot)$ *with $i \in \mathbb{Z}_u$ from an array of random oracles $\mathcal{RO}(i, \cdot)$, is at most*

$$\frac{uN + \binom{u}{2}}{2^{\kappa}} + \frac{N}{2^{192}} + \frac{M}{2^{128}} + \frac{\sqrt{u}N'}{2^{\kappa/2-1}} + \frac{N'}{2^{95}}\,. \tag{1}$$

Here,

- *N is the* computational complexity *expressed in the (computationally equivalent) number of executions of* Xoodoo[6],
- *N' is the* quantum computational complexity *expressed in the (equivalent) number of quantum oracle accesses to* Xoodoo[6], *and*

- *M is the* online *or* data complexity *expressed in the total number of input and output blocks processed by* Xoofff$_{K_i}(\cdot)$.

In (1), the first term accounts for the effort to find one of the u secret keys by exhaustive search, and for the probability that two keys are equal. The second term expresses that the complexity of recovering the accumulator or any rolling state inside Xoofff must be as hard as recovering 192 secret bits. The third term expresses the effort to find a collision in the accumulator.

The fourth and fifth terms only apply if the adversary has access to a quantum computer. The fourth term accounts for a quantum search (or quantum amplification algorithm) to find one of the u keys [11, 7]. The probability of success after N' iterations is $\sin^2((2N'+1)\theta)$ with $\theta = \arcsin\sqrt{u/2^\kappa}$. We upper bound this as $2N'\sqrt{u/2^\kappa}$. The fifth term similarly accounts for a quantum search of a 192-bit secret.

Note that we assume that Xoofff is implemented on a classical computer. In other words, we do not make claims w.r.t. adversaries who would make quantum superpositions of queries to the device implementing Xoofff and holding its secret key(s).

We restrict keys to the uniform distribution to keep our claim simple and to avoid pathological cases that would not offer good security. In the multi-user setting, we require the keys to be independently drawn. If an adversary can manipulate K_i, such as in so-called *unique keys* that consist of a long-term key with a counter appended, we recommend hashing the key and the counter with a proper hash function.

We do support the use of variable-length keys in the multi-user setting, where we assume that a key of given length is selected uniformly of the strings with that length. The claimed distinguishing bound then becomes slightly more complex and is given in Equation (2):

$$\sum_{\kappa\in\mathbf{L}} \frac{u_\kappa N + \binom{u_\kappa}{2}}{2^\kappa} + \frac{N}{2^{192}} + \frac{M}{2^{128}} + \sum_{\kappa\in\mathbf{L}} \frac{\sqrt{u_\kappa}N'}{2^{\kappa/2-1}} + \frac{N'}{2^{95}}\,, \tag{2}$$

with $\mathbf{L}$ the array of the distinct key lengths in use and u_l the number of keys of length l.

4 Xoofff-SANE

Xoofff-SANE is an SAE function built on top of Xoofff with a mode we introduce in this document called Deck-SANE: *deck function based Session Authentication and Nonce-based Encryption.* This mode keeps track of a nonce and the sequence of messages in a string sequence called the *history.* It encrypts the plaintext of a message by adding a keystream that is the result of applying the deck function to the history covering all previous messages in this session. The consequence is that for confidentiality, the history must be unique across all sessions for a given key. For that reason, Deck-SANE initializes the history at the beginning of a session with a user-provided nonce.

4.1 A flaw in Farfalle-SAE

In [2], we presented a mode very similar to Deck-SANE, Farfalle-SAE. Unfortunately, we found a flaw in Farfalle-SAE after closely inspecting it. This was triggered by an email we received from Ted Krovetz reporting a weakness in Farfalle-SIV (see Section 5.1). So, Deck-SANE can be seen as a fixed version of Farfalle-SAE.

The flaw in Farfalle-SAE is related to sequences of messages with empty plaintexts and/or metadata. Let ϵ be the empty string. Namely, the way it constructs the history as a sequence of strings does not allow distinguishing between the following two message sequences:

- a message (A, P) with $A \neq \epsilon$ and $P \neq \epsilon$,
- a message (A, ϵ) followed by a message (ϵ, P).

Both message sequences extend the history with $C||\texttt{1} \circ A||\texttt{0}$, so the tag returned by $\text{wrap}(A, P)$ in the first case and the tag returned by $\text{wrap}(\epsilon, P)$ in the second case are equal. A situation where an adversary could exploit this would be if the sender intends on sending two messages: one containing metadata only, followed by another containing plaintext only. We would have $T_1 = \text{wrap}(A, \epsilon)$ then $(C, T_2) = \text{wrap}(\epsilon, P)$. The adversary withholds T_1, and passes off (A, C) as a single message with resulting tag T_2. The receiver is unable to detect this and successfully authenticates the message, and returns garbled plaintext. We fix this in Deck-SANE by using an additional frame bit that toggles on every message.

4.2 Deck-SANE

We define the SAE mode for deck functions Deck-SANE in Algorithm 2. The session presents the *history* to a deck function for generating tags and keystream. Starting a session initializes the history to a nonce N and returns a tag.

From then on, it supports messages consisting of metadata A and/or plaintext P. Deck-SANE wraps a message in four phases:

1. Encryption: If the plaintext is non-empty, it generates the ciphertext by adding to the plaintext the output of the deck function applied to the history.
2. If the metadata is non-empty or if the ciphertext is empty, it appends the metadata to the history.
3. If the plaintext is non-empty, it appends the ciphertext to the history.
4. Tag generation: It generates the tag by applying the deck function to the history.

Note that a tag authenticates the full history of the session up to that point. Unwrapping is similar.

Deck-SANE has two length parameters: the tag length t and an alignment unit length ℓ. It reserves the first t bits of the output of the deck function for tags and takes keystream from the output of the deck function from an offset that is the smallest multiple of ℓ not shorter than t. It applies domain separation between metadata and ciphertext strings in the history to skip the second phase for plaintext-only messages or the first and third phase for metadata-only or even empty messages. Moreover, Deck-SANE has an attribute e that takes the 1-bit string value `0` or `1` and toggles at each call to (un)wrap. Hence, the individual calls to (un)wrap can be identified in the history without ambiguity.

4.3 Xoofff-SANE

Definition 4 (Xoofff-SANE)**.** Xoofff-SANE is Deck-SANE(F, t, ℓ) with

- $F =$ Xoofff,
- $t = 128$ bits and
- $\ell = 8$ bits.

Algorithm 2 Definition of Deck-SANE(F, t, ℓ)

```
**Parameters:** deck function $F$, tag length $t \in \mathbb{N}$ and alignment unit length $\ell \in \mathbb{N}$

**Initialization** taking key $K \in \mathbb{Z}_2^*$ and nonce $N \in \mathbb{Z}_2^*$, and returning tag $T \in \mathbb{Z}_2^t$
offset $= \ell \left\lceil \frac{t}{\ell} \right\rceil$: the smallest multiple of $\ell$ not smaller than $t$
$e \leftarrow \mathtt{0}^1$
history $\leftarrow N$
$T \leftarrow \mathtt{0}^t + F_K$ (history)
**return** $T$

**Wrap** taking metadata $A \in \mathbb{Z}_2^*$ and plaintext $P \in \mathbb{Z}_2^*$, and returning ciphertext $C \in \mathbb{Z}_2^{|P|}$ and tag $T \in \mathbb{Z}_2^t$
$C \leftarrow P + F_K$ (history) $\lll$ offset
**if** $|A| > 0$ OR $|P| = 0$ **then**
  history $\leftarrow A||\mathtt{0}||e \circ$ history
**if** $|P| > 0$ **then**
  history $\leftarrow C||\mathtt{1}||e \circ$ history
$T \leftarrow \mathtt{0}^t + F_K$ (history)
$e \leftarrow e + \mathtt{1}^1$
**return** $C, T$

**Unwrap** taking metadata $A \in \mathbb{Z}_2^*$, ciphertext $C \in \mathbb{Z}_2^*$ and tag $T \in \mathbb{Z}_2^t$, and returning plaintext $P \in \mathbb{Z}_2^{|C|}$ or an error
$P \leftarrow C + F_K$ (history) $\lll$ offset
**if** $|A| > 0$ OR $|C| = 0$ **then**
  history $\leftarrow A||\mathtt{0}||e \circ$ history
**if** $|C| > 0$ **then**
  history $\leftarrow C||\mathtt{1}||e \circ$ history
$T' \leftarrow \mathtt{0}^t + F_K$ (history)
$e \leftarrow e + \mathtt{1}^1$
**if** $T' = T$ **then**
  **return** $P$
**else**
  **return** error!
```

5 Xoofff-SANSE

Xoofff-SANSE is an SAE function built on top of Xoofff with a mode we introduce in this document called Deck-SANSE: *deck function based Session Authentication and Nonce-Synthetic-based Encryption.* Where Deck-SANE requires to user to ensure that each session is started with a unique combination of key and nonce for confidentiality, in Deck-SANSE this requirement is relaxed. It does this by constructing a nonce of the metadata and plaintext with a generalization of Synthetic IV method of [19]. Similar to Deck-SANE, encryption of plaintext is done by adding a keystream that is the output of the deck function to a history. The difference is in what is covered in that history: In Deck-SANSE, it covers all previous messages *and* the current message. In order to allow decryption, this is realized through the tag: the history for the keystream generation contains previous messages, the metadata of the current message and the tag. The tag is computed before the keystream generation and covers the history of all messages, including the current one. The consequence is that even if two sessions have equal history up to some point and then have different plaintexts, they will likely lead to different tags and the keystreams will be unrelated. Confidentiality still breaks down when these tags collide and the user can eliminate the risk of (history,tag) collisions altogether by including a nonce in the metadata of the first message. In any case, as long as two sessions have the same sequence of messages, they will produce the same sequence of cryptograms. This is unavoidable in a deterministic SAE scheme.

5.1 A flaw in Farfalle-SIV

In [2], we presented a similar mode, albeit with no support for sessions, called Farfalle-SIV. In an email Ted Krovetz drew our attention on a flaw in Farfalle-SIV. So Deck-SANSE is a fixed version of Farfalle-SIV and we took advantage of the occasion to extend to an SAE mode.

The flaw in Farfalle-SIV is the following. Let a legitimate user do a first call to wrap with $(C_1, T_1) = \text{wrap}(A, P_1)$. We have $T_1 = 0^t + F_K(P_1 \circ A)$ and $C_1 = P_1 + F_K(T_1 \circ A)$. Then an adversary makes a second call to wrap with $P_2 = T_1$. She gets $(C_2, T_2) = \text{wrap}(A, P_2)$ with $T_2 = 0^t + F_K(P_2 \circ A) = 0^t + F_K(T_1 \circ A)$. So the tag T_2 reveals the first t bits of the keystream used to encrypt P_1. The root of the problem is the lack of separation between the tag and the keystream generation. We fix this in Deck-SANSE by enforcing domain separation between calls to $F_K(\cdot)$ for tag or keystream.

5.2 Deck-SANSE

Deck-SANSE combines the SIV approach with the session support of Deck-SANE. We define it in Algorithm 3. Deck-SANSE wraps a message in four phases:

1. If the metadata is non-empty or if the ciphertext is empty, it appends the metadata to the history.
2. Tag generation: It generates the tag by applying the deck function to the history, extended with the plaintext of the current messsage, if non-empty.
3. Encryption: If the plaintext is non-empty, it generates the ciphertext by adding to the plaintext the output of the deck function applied to the history extended with the tag.
4. If the plaintext is non-empty, it appends it to the history.

As in Deck-SANE, a tag authenticates the complete history of the session. Unwrapping is similar.

Deck-SANSE has a single length parameter: the tag length t. It applies domain separation between metadata and plaintext strings in the history, as well as between the generation of keystream and of tag. Moreover, as in Deck-SANE, Deck-SANSE has an attribute e that toggles at each call to (un)wrap.

5.3 Xoofff-SANSE

Definition 5 (Xoofff-SANSE)**.** Xoofff-SANSE is Deck-SANSE(F, t) with

- $F = \text{Xoofff}$ and
- $t = 256$ bits.

We take a 256-bit tag because collisions in the tag are likely to appear after generating $2^{t/2}$ tags and we target 128-bit security.

Algorithm 3 Definition of Deck-SANSE(F, t)

```
**Parameters:** deck function $F$ and tag length $t \in \mathbb{N}$

**Initialization**
$e \leftarrow \mathtt{0}^1$
history is initialized to the empty string sequence

**Wrap** taking metadata $A \in \mathbb{Z}_2^*$ and plaintext $P \in \mathbb{Z}_2^*$, and returning ciphertext $C \in \mathbb{Z}_2^{|P|}$ and tag $T \in \mathbb{Z}_2^t$
if $|A| > 0$ OR $|P| = 0$ then
    history $\leftarrow A||\mathtt{0}||e \circ$ history
if $|P| > 0$ then
    $T \leftarrow \mathtt{0}^t + F_K\,(P||\mathtt{01}||e \circ \text{history})$
    $C \leftarrow P + F_K\,(T||\mathtt{11}||e \circ \text{history})$
    history $\leftarrow P||\mathtt{01}||e \circ$ history
else
    $T \leftarrow \mathtt{0}^t + F_K\,(\text{history})$
$e \leftarrow e + \mathtt{1}^1$
return $C, T$

**Unwrap** taking metadata $A \in \mathbb{Z}_2^*$, ciphertext $C \in \mathbb{Z}_2^*$ and tag $T \in \mathbb{Z}_2^t$, and returning plaintext $P \in \mathbb{Z}_2^{|C|}$ or an error
if $|A| > 0$ OR $|C| = 0$ then
    history $\leftarrow A||\mathtt{0}||e \circ$ history
if $|C| > 0$ then
    $P \leftarrow C + F_K\,(T||\mathtt{11}||e \circ \text{history})$
    history $\leftarrow P||\mathtt{01}||e \circ$ history
$T' \leftarrow \mathtt{0}^t + F_K\,(\text{history})$
$e \leftarrow e + \mathtt{1}^1$
if $T' = T$ then
    return $P$
else
    return error!
```

6 Xoofff-WBC and Xoofff-WBC-AE

Xoofff-WBC is a tweakable block cipher built on top of Xoofff and a variant Xoofffie with the mode Farfalle-WBC [2], that constructs the block cipher in a four-round Feistel network.

We first define Xoofffie, a variant of Xoofff aimed at providing differential uniformity, and used in the first and last rounds of Xoofff-WBC. Then, we define and make a security claim on Xoofff-WBC. Finally, we instantiate the Xoofff-WBC-AE authenticated encryption scheme.

6.1 Definition and security claim of Xoofffie

Xoofffie is a function that has the same parameters as Xoofff, with the sole exception of p_d that is the identity function instead of Xoodoo[6].

Definition 6. Xoofffie is Farfalle$[p_\mathrm{b}, p_\mathrm{c}, p_\mathrm{d}, p_\mathrm{e}, \mathrm{roll}_\mathrm{c}, \mathrm{roll}_\mathrm{e}]$ with the following parameters:

- $p_\mathrm{b} = $ Xoodoo[6],

- $p_c = \text{XOODOO}[6]$,
- $p_d = \text{Id}$,
- $p_e = \text{XOODOO}[6]$,
- $\text{roll}_c = \text{roll}_{Xc}$ and
- $\text{roll}_e = \text{roll}_{Xe}$

with Id the identity permutation.

We make the following security claim on XOOFFFIE:

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 $\kappa < 384$. Consider an adversary that can query a function with chosen inputs (X, Δ, i), with $M \in (\mathbb{Z}_2^*)^+$, $\Delta \in \mathbb{Z}_2^*$ and $i \in \mathbb{Z}_u$ that is one of the two following, without knowing which one:*

- *$\mathcal{RO}(\Delta + \text{XOOFFFIE}_{K_i}(X))$: the sum of the output of XOOFFFIE and an offset Δ, and truncated to the length of that offset, and this filtered by random oracle $\mathcal{RO}$.*
- *$\mathcal{RO}_2(i, X, \Delta)$: random oracle $\mathcal{RO}$ applied to the combination of the three inputs.*

Then, the distinguishing advantage is at most:

$$\frac{M}{2^{128}} + \frac{M^2}{2^{\Delta_{min}-4}}\,, \tag{3}$$

with Δ_{min} the minimum length of Δ over the adversary's queries. Note that the adversary can not make direct queries to $\mathcal{RO}$.

This claim expresses a differential uniformity property. When trying to distinguish $\mathcal{RO}(\Delta + \text{XOOFFFIE}_{K_i}(X))$ from $\mathcal{RO}(i, X, \Delta)$, the adversary is limited to observing equality in the expression $\Delta + \text{XOOFFFIE}_{K_i}(X)$ for chosen inputs (i, X, Δ). In other words, an adversary succeeds if she can find XOOFFFIE_K outputs with a predictable difference Δ, i.e., $\text{XOOFFFIE}_{K_i}(X) + \text{XOOFFFIE}_{K_j}(Y) = \Delta$ for $(i, X) \neq (j, Y)$. The output blurring by a random oracle prevents state or key retrieval in the absence of collisions and hence the bound only contains terms related to generating collisions. The first term in (3) covers collisions in the accumulator and the second term in Δ_{min}-bit outputs. For an ideal function the second term would be birthday bound $\frac{M^2}{2^{\Delta_{\text{min}}+1}}$. We tolerate some non-ideal behaviour by multiplying the birthday expression by a factor 2^5.

6.2 Definition of Xoofff-WBC

The wide block cipher XOOFFF-WBC instantiates Farfalle-WBC [2] with two XOODOO-based deck functions that are in turn Farfalle instances.

Definition 7 (XOOFFF-WBC)**.** XOOFFF-WBC is Farfalle-WBC$[H, G, \ell]$ with

- $H = \text{XOOFFFIE}$,
- $G = \text{XOOFFF}$ and
- $\ell = 8$ bits.

Making joint use of XOOFFF and XOOFFFIE instances that share a key is not something we support in general. However, in XOOFFF-WBC we believe this is no problem and we make the following dedicated security claim on XOOFFF-WBC.

Claim 3. *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* $\kappa < 384$ *and let* $P_{K_i}(\cdot)$ *with* $i \in \mathbb{Z}_u$ *be instances of* Xoofff-WBC*. Each of these instances support two interfaces:*

Encipherment *denoted as* $C = P_{K_i}(W, P)$ *taking as input a tweak* W *and a plaintext* P *and returning a cryptogram* C*;*

Decipherment *denoted as* $P = P_{K_i}^{-1}(W, C)$ *taking as input a tweak* W *and a cryptogram* C *and returning a plaintext* P*.*

We express as $\mathrm{Adv}^{\mathrm{sprp}}$ *the probability of distinguishing* $P_{K_i}(W, \cdot)$ *from an array of uniformly and independently drawn random permutations* $\pi_{i,W,n}$ *indexed by the key index* i*, the value of* W *and the length* $n = |P| = |C|$*, where the adversary can query the inverse permutations.*

Let n_{min} *be the minimum length* n *among all the queries. The* $\mathrm{Adv}^{\mathrm{sprp}}$ *is claimed to be upper bounded by*

$$(1) + \frac{M^2}{2^{n_{min}/2-8}} . \tag{4}$$

Here, N*,* N' *and* M *are as in Claim 1, except that* M *also counts the number of input and output blocks processed by* Xoofffie*.*

The terms in (4) are those of Claim 1 and an additional term. The additional term covers the case of an adversary obtaining a collision in one of the branches of the Feistel network, see [2] for details. We relate this to the ability of doing this in the first and last rounds, that make use of Xoofffie. In this use case, it has $\Delta_{\min} \geq n_{\min}/2 - 4$ and hence the term $\frac{M^2}{2^{\Delta_{\min}-4}}$ becomes $\frac{M^2}{2^{n_{\min}/2-8}}$.

6.3 Definition of Xoofff-WBC-AE

On top of Xoofff-WBC, we define the Xoofff-WBC-AE authenticated encryption scheme as an instance of Farfalle-WBC-AE [2] with the same parameters as Xoofff-WBC and with $t = 128$. In a nutshell, when wrapping, Xoofff-WBC-AE adds t bits of redundancy at the end of the plaintext P before encipherment with Xoofff-WBC: it enciphers $P||0^t$ with the metadata A as tweak. When unwrapping, Xoofff-WBC-AE calls Xoofff-WBC decryption and checks that the last t bits of the result are indeed 0^t.

Definition 8 (Xoofff-WBC-AE)**.** Xoofff-WBC-AE is Farfalle-WBC-AE$[H, G, \ell, t]$ with

- $H =$ Xoofffie,
- $G =$ Xoofff,
- $\ell = 8$ bits,
- $t = 128$ bits.

7 Xoodyak

In this section, we specify Xoodyak, 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, 9]. It is stateful and shares features with Markku Saarinen's Blinker [20], Mike Hamburg's Strobe protocol framework [12] 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. The mode of operation employed in XOODYAK is called *Cyclist*, as a lightweight counterpart to KEYAK's Motorist mode [5]. 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.

7.1 Notation

XOODYAK works with bytes, i.e., a sequence of 8 bits, 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 $|X|_8$, which is equal to its bit length divided by 8. We denote with $\mathrm{enc}_8(x)$ a byte whose value is the integer $x \in \mathbb{Z}_{256}$. Byte values are noted in hexadecimal in a typewriter font enclosed in single quotes, e.g., '1A' is a byte whose value is the integer 26.

7.2 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 $\max(R_{\mathrm{hash}}, R_{\mathrm{kin}}, R_{\mathrm{kout}}) + 2 \leq b'$, where b' 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 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 so far, and that any two subsequent output strings are in 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

Table 4: 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:	
$\textsc{Cyclist}(K, \text{id}, \text{counter})$	$\text{counter} \circ \text{id} \circ \textsc{AbsorbKey}$
$\textsc{Absorb}(X)$	$X \circ \textsc{Absorb}$
$C \leftarrow \textsc{Encrypt}(P)$	$P \circ \textsc{Crypt}$
$P \leftarrow \textsc{Decrypt}(C)$	$P \circ \textsc{Crypt}$
$\textsc{Squeeze}(\ell)$	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ \textsc{Squeeze}$
$\textsc{SqueezeKey}(\ell)$	$\textsc{Block}^{n_{\text{kout}}(\ell)} \circ \textsc{SqueezeKey}$
$\textsc{Ratchet}()$	$\textsc{Ratchet}$

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 9 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 5.

Definition 9. 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 4, 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 4 and 5.

Algorithm 4 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: $(\textsc{phase}, s) \leftarrow (\text{up}, \texttt{`00`}^{f.b'})$
  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:** $\textsc{Absorb}(X)$
  $\textsc{AbsorbAny}(X, R_{\text{absorb}}, \texttt{`03`}\ \text{(absorb)})$

**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** $\textsc{SqueezeAny}(\ell, \texttt{`40`}\ \text{(squeeze)})$

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

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

Algorithm 5 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 $|K \mid\mid \text{id}|_8 \leq R_{\text{kin}} - 1$
  $(\textsc{mode}, R_{\text{absorb}}, R_{\text{squeeze}}) \leftarrow (\text{keyed}, R_{\text{kin}}, R_{\text{kout}})$
  $\textsc{AbsorbAny}(K \mid\mid \text{id} \mid\mid \text{enc}_8(|\text{id}|_8), R_{\text{absorb}}, \text{‘02‘ (key)})$
  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
    $O_i \leftarrow I_i \oplus \textsc{Up}(|I_i|_8, \text{‘80‘ (crypt)}$ 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 $|Y|_8 < \ell$ do
    $\textsc{Down}(\epsilon, \text{‘00‘})$
    $Y \leftarrow Y \mid\mid \textsc{Up}(\min(\ell - |Y|_8, R_{\text{squeeze}}), \text{‘00‘})$
  return $Y$

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

Internal interface: $Y_i \leftarrow \textsc{Up}(|Y_i|_8, c_U)$
  $(\textsc{phase}, s) \leftarrow (\text{up}, f(s$ if $\textsc{mode} = \text{hash}$ else $s \oplus (\text{‘00‘}^* \mid\mid c_U)))$
  return $s[0] \mid\mid s[1] \mid\mid \ldots \mid\mid s[|Y_i|_8 - 1]$

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 $|X_i|_8 = n$ except possibly the last block.
```

7.3 Xoodyak and its claimed security

We instantiate Xoodyak in Definition 10 and attach to it security Claims 4 and 5.

Definition 10. 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 4. *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 5. *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*

$$\frac{q_{\text{iv}}N + \binom{u}{2}}{2^\kappa} + \frac{N}{2^{184}} + \frac{(L+\Omega)N + \binom{L+\Omega+1}{2}}{2^{192}} + \frac{M^2}{2^{382}} + \frac{Mq}{2^{\min(192+\kappa,384)}} \,. \tag{5}$$

Here we follow the notation of the generic security bound of the FSKD [9], 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 plaintext 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.,*

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

Claims 4 and 5 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 7.4. Given that they are bounded by M, Xoodyak resists to a data complexity of up to 2^{64} blocks, as the probability in Eq. (5) 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.

7.4 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.

7.5 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:

```
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 4. 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:

```
Cyclist($\epsilon, \epsilon, \epsilon$)
Absorb($x_1$)
Absorb($x_2$)
Absorb($x_3$)
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 7.6.4, 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.

7.6 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 $\text{Cyclist}(K, \text{id}, \text{counter})$ 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.

7.6.1 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 [6]. Claim 5 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.

7.6.2 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 [21]. 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.

7.6.3 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:

```
$\textsc{Cyclist}(K, \text{id}, \epsilon)$
$\textsc{Absorb}(\text{nonce})$
$\textsc{Absorb}(A)$
$C \leftarrow \textsc{Encrypt}(P)$
$T \leftarrow \textsc{Squeeze}(t)$
**return** $(C, T)$
```

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

```
$\textsc{Cyclist}(K, \text{id}, \epsilon)$
$\textsc{Absorb}(\text{nonce})$
$\textsc{Absorb}(A)$
$P \leftarrow \textsc{Decrypt}(C)$
$T' \leftarrow \textsc{Squeeze}(t)$
**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 5. The resulting simplified security claim is given in the following corollary.

Corollary 2. *Assume that (1)* Xoodyak *indeed satisfies Claim 5; (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	$\min(184, \kappa, 8t)$	$96 + \kappa/2$

7.6.4 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 7.6.3, we add the support for messages (A_i, P_i), where A_i, P_i or both can be empty.

```
$\textsc{Cyclist}(K, \text{id}, \epsilon)$
$\textsc{Absorb}(\text{nonce})$
$\textsc{Absorb}(A_1)$
$C_1 \leftarrow \textsc{Encrypt}(P_1)$
$T_1 \leftarrow \textsc{Squeeze}(t)$
  $\Rightarrow$ 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 5, 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 $\Omega = 0$.

7.6.5 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.

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

7.6.6 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 $\textsc{Cyclist}(K_i, \epsilon, \epsilon)$
  $K_{i+1} \leftarrow \textsc{SqueezeKey}(\ell_{\text{sub}})$ {and store $K_{i+1}$ by overwriting $K_i$}
  $\textsc{Ratchet}()$ {optional}
  $\textsc{Absorb}(A_i)$
  $C_i \leftarrow \textsc{Encrypt}(P_i)$
  $T_i \leftarrow \textsc{Squeeze}(t)$
    $\Rightarrow$ 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 $\textsc{Ratchet}()$.

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.

7.6.7 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 5; (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

7.7 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 of the state, it needs only to set the rest of the state to the appropriate value before calling f.

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

Acknowledgement

We thank Bart Mennink and Guido Bertoni for useful discussion, Ted Krovetz for finding and reporting to us the flaw in Farfalle-SIV, as noted in Section 5.1 and Johan De Meulder for his contributions on the early stages of the definition of Xoodoo.

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, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, *Farfalle: parallel permutation-based cryptography*, IACR Trans. Symmetric Cryptol. **2017** (2017), no. 4, 1–38.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *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] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, *CAESAR submission:* KETJE *v2*, September 2016, https://keccak.team/ketje.html.

[5] ______, *CAESAR submission:* KEYAK *v2, document version 2.2*, September 2016, https://keccak.team/keyak.html.

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

[7] G. Brassard, P. Hoyer, M. Mosca, and A. Tapp, *Quantum amplitude amplification and estimation*, Contemporary Mathematics **305** (2002), 53–74.

[8] 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.

[9] 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.

[10] T. Dierks and E. Rescorla, *The transport layer security (TLS) protocol version 1.2*, Network Working Group of the IETF, RFC 5246, August 2008.

[11] L. K. Grover, *A fast quantum mechanical algorithm for database search*, Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, May 1996 (Gary L. Miller, ed.), ACM, 1996, pp. 212–219.

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

[13] S. Hoffert, *Xoodoo reference code in C++*, August 2018, https://github.com/XoodooTeam/.

[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] P. Rogaway and T. Shrimpton, *A provable-security treatment of the key-wrap problem*, Advances in Cryptology - EUROCRYPT 2006, Proceedings (S. Vaudenay, ed.), Lecture Notes in Computer Science, vol. 4004, Springer, 2006, pp. 373–390.

[20] 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.

[21] 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.

[22] G. Van Assche, *On dec(k) functions*, INDOCRYPT 2018, December 2018, `https://keccak.team/files/OnDecAndDeckFunctions-Indocrypt2018.pdf`.

[23] G. Van Assche, R. Van Keer, and Contributors, *Extended* Keccak *code package*, August 2018, `https://github.com/XKCP/XKCP`.

[24] T. Ylonen and C. Lonvick, *The secure shell (SSH) protocol architecture*, Network Working Group of the IETF, RFC 4251, January 2006.

Table 5: The round constants with indices -11 to 0

i	q_i	s_i	c_i	in hex
-11	$1+t$	3	$t^3+t^4+t^6$	`0x00000058`
-10	$t+t^2$	2	$t^3+t^4+t^5$	`0x00000038`
-9	$1+t+t^2$	6	$t^6+t^7+t^8+t^9$	`0x000003C0`
-8	$1+t^2$	4	$t^4+t^6+t^7$	`0x000000D0`
-7	1	5	t^5+t^8	`0x00000120`
-6	t	1	t^2+t^4	`0x00000014`
-5	t^2	3	t^5+t^6	`0x00000060`
-4	$1+t$	2	$t^2+t^3+t^5$	`0x0000002C`
-3	$t+t^2$	6	$t^7+t^8+t^9$	`0x00000380`
-2	$1+t+t^2$	4	$t^4+t^5+t^6+t^7$	`0x000000F0`
-1	$1+t^2$	5	$t^5+t^7+t^8$	`0x000001A0`
0	1	1	t^1+t^4	`0x00000012`

A Constants for any number of rounds

We here detail how the round constants are constructed and, following the formula, how to compute them for any number of rounds.

The round constants C_i are planes with a single non-zero lane at $x = 0$. We specify the value of the lanes at $x = 0$ in the round constants as binary polynomials $p_i(t)$ where the coefficient of t^i denotes the bit of the lane with coordinate $z = i$. We define $p_i(t)$ in terms of a polynomial $q_i(t)$ and an integer s_i in the following way:

$$p_i(t) = t^{s_i}\left(q_i(t) + t^3\right) \text{ with } q_i(t) = t^i \bmod 1 + t + t^3 \text{ and } s_i = 3^i \bmod 7\,.$$

The sequence of polynomials $q_i(t)$ has period 7 and the sequence of offsets s_i has period 6. It follows that the sequence of round constants $\mathrm{C}_i(t)$ have period 42. An instance of Xoodoo with r rounds uses the round constants with indices $1-r$ to 0. We list the round constants with indices -11 to 0 in Table 5.

B Single-page definition sheet

Modes specified in this document:

- Deck-SANE(F, t, ℓ)
- Deck-SANSE(F, t)
- $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$

Modes instantiated in this document, specified in [2]:

- Farfalle$[p_{\text{b}}, p_{\text{c}}, p_{\text{d}}, p_{\text{e}}, \text{roll}_{\text{c}}, \text{roll}_{\text{e}}]$
- Farfalle-WBC$[H, G, \ell]$
- Farfalle-WBC-AE$[H, G, \ell, t]$

Members of the $\textsc{Xoodoo}$ suite, built on top of $\textsc{Xoodoo}[n_{\text{r}}]$: $\textsc{Xoodoo}$ with n_{r} rounds:

- $\textsc{Xoofff} \triangleq \text{Farfalle}[\textsc{Xoodoo}[6], \textsc{Xoodoo}[6], \textsc{Xoodoo}[6], \textsc{Xoodoo}[6], \text{roll}_{\text{Xc}}, \text{roll}_{\text{Xe}}]$
- $\textsc{Xoofffie} \triangleq \text{Farfalle}[\textsc{Xoodoo}[6], \textsc{Xoodoo}[6], \text{Id}, \textsc{Xoodoo}[6], \text{roll}_{\text{Xc}}, \text{roll}_{\text{Xe}}]$
- $\textsc{Xoofff}\text{-SANE} \triangleq \text{Deck-SANE}(\textsc{Xoofff}, 128, 8)$
- $\textsc{Xoofff}\text{-SANSE} \triangleq \text{Deck-SANSE}(\textsc{Xoofff}, 256)$
- $\textsc{Xoofff}\text{-WBC} \triangleq \text{Farfalle-WBC}[\textsc{Xoofffie}, \textsc{Xoofff}, 8]$
- $\textsc{Xoofff}\text{-WBC-AE} \triangleq \text{Farfalle-WBC-AE}[\textsc{Xoofffie}, \textsc{Xoofff}, 8, 128]$
- $\textsc{Xoodyak} \triangleq \textsc{Cyclist}[\textsc{Xoodoo}[12], 16, 44, 24, 16]$

C Change log

C.1 14 March 2019

This version adds Xoodyak to the Xoodoo suite, see Section 7.

C.2 25 August 2018

Initial release.