Xoodoo cookbook

Joan Daemen[2,1], Seth Hoffert, 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. The final 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. 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, e.g., with a hash function or a dedicated lightweight MAC function and we will update it accordingly.

Keywords: permutation-based crypto · Farfalle · 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-$p[1600, n_r]$ 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-$p[400, n_r]$ is problematic as it is defined in terms of operates 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 keyed 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.

This document does not include extensive design rationale or analysis, nor does it provide performance benchmarks. We plan to publish those in one or more follow-up papers. 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. Note that the security claim in this stage serves as a challenge for cryptanalists rather than a promise of security strength for users.

Additionally, we will make a reference implementation in C++ available of Xoodoo and the members of the Xoodoo suite in [10] and optimized implementations in C and assembler in the *eXtended (or* Xoodoo *and)* Keccak *Code Package* (XKCP) [12].

Over time, we may add cryptographic functions to the Xoodoo suite, e.g., for hashing or dedicated lightweight MAC computation. When that is the case, we will update this document accordingly. In ISO/IEC terminology this would be called a *standing document*.

1.1 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. 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.

1.2 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 [8] or the Secure Shell (SSH) protocol [13]. 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].

1.3 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 Xooffff 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 Xooffff-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 Xooffff. Xooffff-SANE relies on user-provided nonces for confidentiality.
- The Xooffff-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 Xooffff. Xooffff-SANSE is more robust against nonce misuse and realizes this by using the SIV mechanism.
- The Xooffff-WBC wide block cipher, specified in Section 6. We obtain this by instantiating Farfalle-WBC with Xooffff and Xoofffie, a variant of Xooffff whose purpose is solely to provide differential uniformity. We give security claims for Xooffff-WBC and a dedicated claim for Xoofffie. Finally, we define the Xooffff-WBC-AE authenticated encryption scheme by applying Farfalle-WBC-AE on top of Xooffff-WBC.

We make no security claims for the Xooffff-SANE and Xooffff-SANSE SAE schemes as their claimed security follows immediately from the security claim of Xooffff. Similarly, the claimed security of Xooffff-WBC-AE follows directly from the security claim of Xooffff-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.

2 Xoodoo

Xoodoo is a family of permutations parameterized by its number of rounds n_r and denoted Xoodoo$[n_\mathrm{r}]$. The number of rounds is determined by the construction or mode the permutation is used in.

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

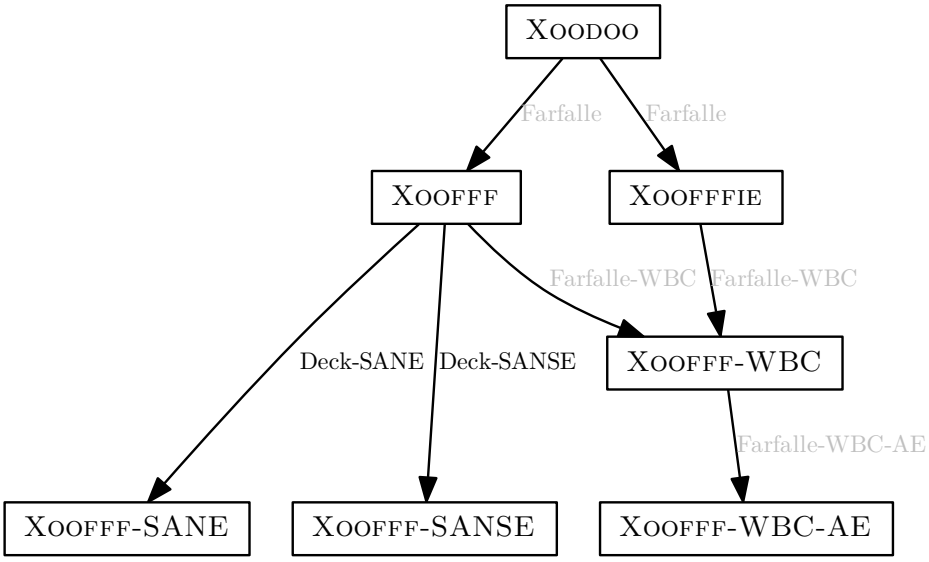

Figure 1: Overview of the Xoodoo suite, with the schemes in boxes and the modes indicated on the edges. All schemes and the modes in black print are specified in this document, the modes in grey are defined in [2].

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

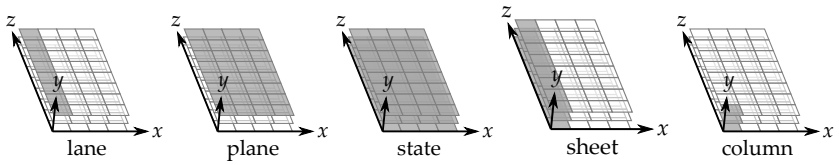

Figure 2: Toy version of the Xoodoo state, with lanes reduced to 8 bits, and different parts of the state highlighted.

Table 1: Notational conventions

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in position (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$[r]$ with r the number of rounds

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

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

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

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

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

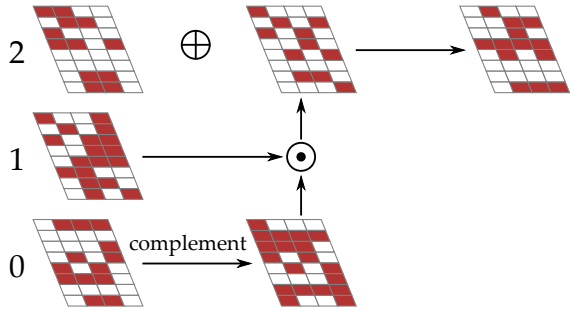

Figure 3: Effect of χ on one plane.

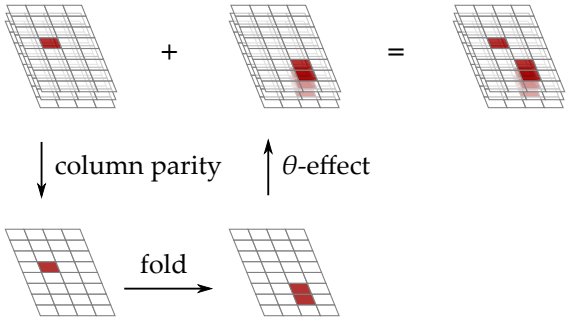

Figure 4: Effect of θ on a single-bit state.

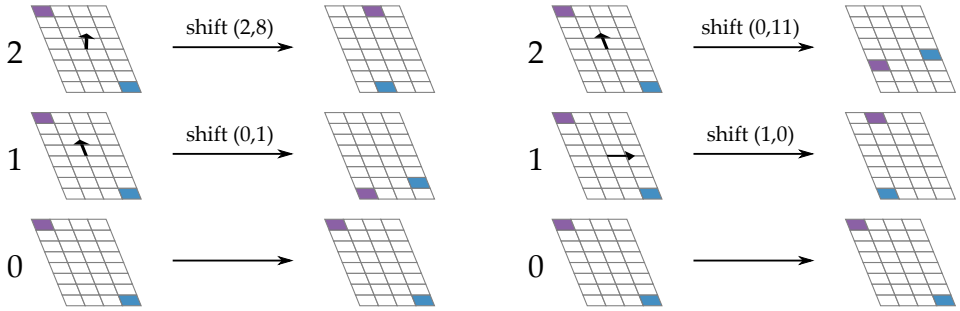

Figure 5: Illustration of ρ_{east} (left) and ρ_{west} (right).

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 $x+v$ and setting bits at $x < v$ 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:

$$\begin{aligned} A_{0,0} &\leftarrow A_{1,0} \cdot A_{2,0} + (A_{0,0} \lll 5) + (A_{1,0} \lll 13) + 111||0^{29} \\ B &\leftarrow A_0 \lll (3,0) \\ A_0 &\leftarrow A_1 \\ A_1 &\leftarrow A_2 \\ A_2 &\leftarrow B \end{aligned}$$

Definition 1 (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:

- $p_{\text{b}} =$ Xoodoo[6],
- $p_{\text{c}} =$ Xoodoo[6],
- $p_{\text{d}} =$ Xoodoo[6],
- $p_{\text{e}} =$ 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* $\textsc{Xoofff}_{K_i}(\cdot)$*.*

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 [9, 7]. The probability of success after N' iterations is $\sin^2\left((2N'+1)\,\theta\right)$ 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 such as KANGAROOTWELVE [6].

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 $C||\mathtt{1} \circ A||\mathtt{0}$, so the tag returned by $\mathrm{wrap}(A, P)$ in the first case and the tag returned by $\mathrm{wrap}(\epsilon, P)$ 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 $T_1 = \mathrm{wrap}(A, \epsilon)$ then $(C, T_2) = \mathrm{wrap}(\epsilon, P)$. 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.

Definition 2 (Xoofff-SANE)**.** Xoofff-SANE is Deck-SANE(F, t, ℓ) with

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

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 ℓ 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) $\ll$ 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) $\ll$ 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 [11]. 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.

5.1.1 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.

5.1.2 Xoofff-SANSE

Definition 3 (Xoofff-SANSE)**.** Xoofff-SANSE is Deck-SANSE(F, t) with

- $F =$ Xoofff and
- $t = 256$

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 $\leftarrow A||0||e \circ$ 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 $\leftarrow A||0||e \circ$ 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 Xooffffie with the mode Farfalle-WBC [2], that constructs the block cipher in a four-round Feistel network.

We first define Xooffffie, 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

Xooffffie 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 4. Xooffffie 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],

- $p_{\mathrm{c}} = \textsc{Xoodoo}[6]$,
- $p_{\mathrm{d}} = \mathrm{Id}$,
- $p_{\mathrm{e}} = \textsc{Xoodoo}[6]$,
- $\mathrm{roll}_{\mathrm{c}} = \mathrm{roll}_{\mathrm{Xc}}$ and
- $\mathrm{roll}_{\mathrm{e}} = \mathrm{roll}_{\mathrm{Xe}}$

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:*

- *$\mathcal{RO}(\Delta + \textsc{Xoofffie}_{K_i}(X))$: 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 $\mathcal{RO}(\Delta + \textsc{Xoofffie}_{K_i}(X))$ from $\mathcal{RO}(i, X, \Delta)$, the adversary is limited to observing equality in the expression $\Delta + \textsc{Xoofffie}_{K_i}(X)$ for chosen inputs (i, X, Δ). In other words, an adversary succeeds if she can find $\textsc{Xoofffie}_K$ outputs with a predictable difference Δ, i.e., $\textsc{Xoofffie}_{K_i}(X) + \textsc{Xoofffie}_{K_j}(Y) = \Delta$ 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 $\Delta_{\min}$-bit outputs. For an ideal function the second term would be birthday bound $\frac{M^2}{2^{\Delta_{\min}+1}}$. 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 5 (Xoofff-WBC)**.** Xoofff-WBC is Farfalle-WBC$[H, G, \ell]$ with

- $H = \textsc{Xoofffie}$,
- $G = \textsc{Xoofff}$ and
- $\ell = 8$.

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

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

Acknowledgement

We thank Bart Mennink, Guido Bertoni and Michaël Peeters 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 (Wieland Fischer and Naofumi 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] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer, and B. Viguier, *KangarooTwelve: Fast hashing based on Keccak-p*, Applied Cryptography and Network Security, ACNS 2018, Proceedings (B. Preneel and F. Vercauteren, eds.), Lecture Notes in Computer Science, vol. 10892, Springer, 2018, pp. 400–418.

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

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

[9] 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.

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

[11] 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.

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

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

Table 4: 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 $C_i(t)$ 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 4.

B Single-page definition sheet

Modes specified in this document:

- Deck-SANE(F, t, ℓ)
- Deck-SANSE(F, t)

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

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

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

- Xoofff$\triangleq$Farfalle[Xoodoo[6], Xoodoo[6], Xoodoo[6], Xoodoo[6], $\mathrm{roll}_{\mathrm{Xc}}, \mathrm{roll}_{\mathrm{Xe}}$]
- Xoofffie$\triangleq$Farfalle[Xoodoo[6], Xoodoo[6], Id, Xoodoo[6], $\mathrm{roll}_{\mathrm{Xc}}, \mathrm{roll}_{\mathrm{Xe}}$]
- Xoofff-SANE$\triangleq$Deck-SANE(Xoofff, 128, 8)
- Xoofff-SANSE$\triangleq$Deck-SANSE(Xoofff, 256)
- Xoofff-WBC$\triangleq$Farfalle-WBC[Xoofffie, Xoofff, 8]
- Xoofff-WBC-AE$\triangleq$Farfalle-WBC-AE[Xoofffie, Xoofff, 8, 128]