Shaking up authenticated encryption

Joan Daemen*, Seth Hoffert†, Silvia Mella*, Gilles Van Assche‡ and Ronny Van Keer‡
* *Radboud University, Nijmegen, The Netherlands*
joan@cs.ru.nl, silvia.mella@ru.nl
† *Nebraska, USA*
seth.hoffert@gmail.com
‡ *STMicroelectronics, Diegem, Belgium*
gilles.vanassche@st.com, ronny.vankeer@st.com

***Abstract*—Authenticated encryption (AE) is a cryptographic mechanism that allows communicating parties to protect the confidentiality and integrity of messages exchanged over a public channel, provided they share a secret key. In this work, we present new AE schemes leveraging the SHA-3 standard functions SHAKE128 and SHAKE256, offering 128 and 256 bits of security strength, respectively, and their "Turbo" counterparts. They support session-based communication, where a ciphertext authenticates the sequence of messages since the start of the session. The chaining in the session allows decryption in segments, avoiding the need to buffer the entire deciphered cryptogram between decryption and validation. And, thanks to the collision resistance of (Turbo)SHAKE, they provide so-called CMT-4 committing security, meaning that they provide strong guarantees that a ciphertext uniquely binds to the key, plaintext and associated data. The AE schemes we propose have a unique combination of advantages. The most important are that 1) their security is based on the security claim of SHAKE, that has received a large amount of public scrutiny, that 2) they make use of the standard KECCAK-p permutation that not only receives more and more dedicated hardware support, but also allows competitive software-only implementations thanks to the TurboSHAKE instances, and that 3) they do not suffer from a 64-bit birthday bound like most AES-based schemes. Of independent interest, we introduce the deck cipher as the stateful counterpart of the deck function and the duplex cipher generalizing keyed duplex and harmonize their security notions. Finally, we provide an elegant solution for multi-layer domain separation.**

1. Introduction

An authenticated encryption (AE) scheme is a cryptographic primitive that combines encryption with authentication. The operation of encryption is often called *wrapping* to avoid confusion, and we will follow this convention. Symmetrically, decryption is called *unwrapping*. Wrapping converts a plaintext P into a ciphertext C under a secret key K and (for almost all schemes also) so-called associated data AD. Due to the presence of the latter, such schemes are often called AEAD schemes, but we will stick to the term AE schemes for brevity. Given the secrecy of the key, the ciphertext authenticates both plaintext P and associated data AD: it contains the encrypted plaintext X and a tag T that is computed over P and AD. The unwrap function verifies this tag and will return the plaintext P only if valid, and an error otherwise. The tag length τ determines the resistance of the AE scheme against forgery. In some AE schemes, the ciphertext does not contain a separate tag but has a certain redundancy that is verified during unwrap. For this reason, the tag length is sometimes referred to as *ciphertext expansion*.

Also, AE schemes are usually built as a mode of operation on top of a simpler primitive (e.g., blockcipher), and we will follow this terminology for instance when distinguishing properties that apply generically to the mode from those that apply to the scheme more specifically.

The main advantages of AE schemes over separate encryption and MAC computation are efficiency and convenience. Combining the two functions in one allows achieving a given security strength in a more efficient way, as witnessed by duplex-based AE schemes [1]. And instead of having to figure out how to combine the two functions with all its pitfalls as illustrated by [2], protocol designers can just take a primitive that does both. However, looking at existing AE schemes, there is room for improvement in this last department, and that is what we address in this paper.

First of all, in most AE schemes the wrap and unwrap functions have an additional input field that is sometimes called initial value IV, but most often *nonce* N. In many cases its length is fixed to 12 bytes or so. For security reasons, N must be unique for each wrap operation with the same key, and violation of this nonce property may result in a breakdown of security. There are different reasons for this.

Let us first look at the encryption. Many modes and schemes, including NIST standards GCM [3], CCM [4] and industry standard ChaCha20-Poly1305 [5], perform stream encryption where the keystream only depends on the nonce and key. Clearly, nonce violation leads to keystream equality, and the ciphertext difference leaks the plaintext difference. In duplex-based AE modes and schemes such as SpongeWrap [1] or Ascon [6] the AD is absorbed before the plaintext, and therefore this leakage occurs only if the pair (N, AD) is repeated. In modes that do block encryption, such as OCB [7], in case of nonce violation ciphertext block equality leaks plaintext block equality.

The second reason is the way tag is computed. Let us take a look at the most popular AE schemes. GCM computes the tag by applying the Wegman-Carter con-

struction [8]: it performs stream encryption to the output of the polynomial hash function GHASH applied to C and AD. The former uses a secret subkey L derived from the AE key K, and nonce reuse allows an adversary to compute L and so produce a forgery for all nonce values. ChaCha20-Poly1305 applies the polynomial hash function Poly1305 to C and AD but derives its subkeys from the AE key K using the nonce. Also here nonce violation allows computing the tag subkey, but its knowledge limits forgeries to the value of the repeated nonce. Also in OCB, nonce violation allows forgeries for the values of the repeated nonce.

The third reason is that in some AE schemes the nonce is required for secure state initialization. Examples of this are Ascon but also NIST lightweight candidate Subterranean 2.0 [9]. Massive nonce violation would allow the recovery of Ascon's internal state violating the confidentiality of all plaintexts encrypted with the given nonce, and even key recovery in Subterranean 2.0 [10].

Next to all these reasons that make the nonce a critical input, there are also some other reasons why the presence of a nonce field is perceived as desirable.

Modern AE schemes are deterministic, so wrapping equal plaintext under the equal N and AD, yields equal ciphertext. By imposing uniqueness of the nonce N, this leakage is prevented. However, if we include the AD in the encryption context, as is the case in spongeWrap and Ascon, one may as well impose uniqueness of the AD to prevent this leakage.

In many use cases one wants protection against replay attacks: the receiver shall be able to detect whether it receives a cryptogram for the first time or whether it is a replay. This can be achieved by a message counter: the receiver keeps track of this counter and for each received cryptogram it ensures this counter is larger than for previous cryptograms. It seems natural to use the nonce field N for this counter. However, as the tag also authenticates the AD, one can include this counter in the AD field. In some other use cases, one wants freshness: the receiver shall be able to detect whether the cryptogram has been sent recently. This can be done by including a receiver-originating unpredictable challenge in the nonce N but of course this can be included in the AD as well.

The short length of the nonce field is often a problem. If nonces are generated randomly, the birthday bound forms a limitation. Alternatively, if uniqueness is obtained by concatenating multiple data elements (e.g., date and time, session identifier), it may be too short.

The problem of the short nonce field has been addressed in so-called Synthetic Initial Value (SIV) AE schemes where the tag over the plaintext and AD is used as a diversifier for encryption [11]. A prominent example of an SIV AE scheme is AES-GCM-SIV [12], [13]. SIV schemes require two passes over the plaintext during wrapping, which requires buffering the whole plaintext.

A common limitation often encountered when using AE schemes is the need to hold the full ciphertext in memory during unwrap. Releasing the decrypted ciphertext before the tag verification (a.k.a. release of unverified plaintext, or RUP) may have serious consequences. This, and the buffering of plaintext in SIV make the exchange of long messages difficult to handle, especially on systems that do not have much memory.

A solution to this last problem is to support *intermediate tags*. The plaintext is split into fragments, and each is wrapped separately, resulting in a ciphertext and a tag. In its simplest form, the tag authenticates the fragment together with a fragment counter and a flag indicating whether this is the last fragment. Fragments can thus be unwrapped out of order, but their position in the sequence is authenticated. Also, the AD has to be sent only once but is included in the encryption context of all fragments. In another form, each tag authenticates the sequence of all the fragments up to the current one, so that the last tag ensures the authenticity of the whole plaintext. In any case, this makes the wrapping and unwrapping *stateful*.

In modern applications, parties do not limit themselves to exchanging individual messages, but usually have to wrap sequences of messages, often in bi-directional communications. A simple example is secure messaging applications, such as Signal, where users exchange messages in a continuous dialogue. The concept of *sessions* generalizes the idea of intermediate tags by allowing fragments to be individual messages and by supporting a continuous stream of messages. With the term session, we refer to a sequence of messages, where a message is authenticated in the context of those previously sent within the sequence.

A *session-supporting* AE scheme deals with such sequences of messages by having intermediate tags ensuring that the encryption context and authenticity of the current ciphertext depends on all previous messages in the session. The intended behavior of this type of AE schemes can be formalized, and the *jammin cipher* provides a good solution [14]. Note that session-supporting AE covers the traditional notion of authenticated encryption of a single message (i.e., a pair (AD, P)) as well, where each session then contains a single message. Back to the nonce requirement, sessions help ensuring uniqueness: As soon as the first message in a session has unique AD, all the subsequent messages automatically have a unique context.

Yet another common limitation of many AE schemes is the so-called *birthday bound* that limits the amount of data that can be encrypted or authenticated. Many schemes are built using modes of operation on top of the AES [15], and for simple such schemes, the security breaks down when the amount of data encrypted with a given key approaches 2^{64}. There are AES-based AE modes that do not suffer from this limitation, but they tend to be more complicated and/or expensive, see for instance the AE mode specified in [16].

Finally, most common AE schemes do not offer *committing security*. When a sender and a receiver hold a secret key K that was not shared with anyone else, then the successful unwrap of a ciphertext C authenticates the origin of the decrypted plaintext, and the receiver knows that it comes from the legitimate sender. However, as soon as the key is leaked or under adversarial control, we fall outside of AE's usual security model and all bets are off. In general, AE does not ensure that the key used to successfully unwrap a ciphertext is the same as the one that was used when it was wrapped.

As a matter of fact, for AES-GCM and ChaCha20-Poly1305, one can find pairs (K, P) that lead to equal ciphertexts [17]. Schemes that are susceptible to this are said to *not commit to the key*. On the contrary, the property of key commitment implies that a ciphertext can only

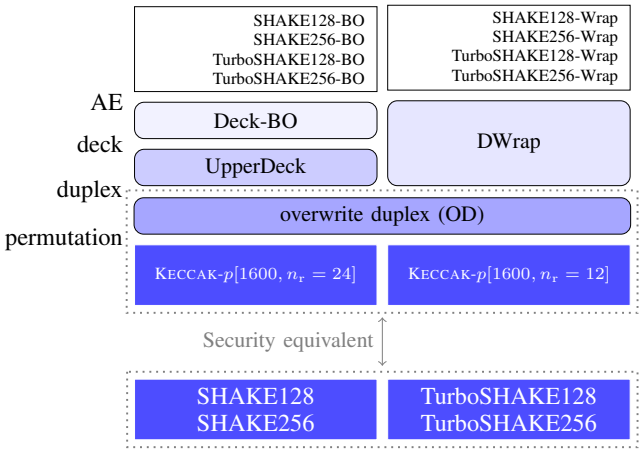

Figure 1. The hierarchy of modes and schemes that we propose.

be successfully unwrapped using the same key that was used for wrapping it. The strongest committing notion is called CMT-4, denoting the infeasibility to generate colliding ciphertexts for different $(K, [N,]AD, P)$ tuples [18]. Several generic solutions have been presented to turn existing AE schemes into committing AE schemes. Most of them rely on the application of a collision-resistant hash function or a so-called key-robust PRF on top of the AE scheme, relying on two or more primitives. A more extended overview on committing AE is given in Appendix C.

2. Our contribution

The main contributions of this paper are:

1) We introduce the deck cipher, an alternative definition of a deck function [19] modeling it as a (stateful) object, and we harmonize the security definition of the duplex cipher (a generalization of keyed duplex) [1] and the deck cipher with an ideal-world object that we call *idaho*.
2) We present two session-supporting AE modes, one that is based on a duplex cipher and the other that adapts Deck-BO [14] to deck ciphers.
3) We build concrete AE schemes whose security is equivalent to that of the SHAKE and TurboSHAKE extendable output functions (XOFs) [20], [21].

We now give more details on the concrete schemes and their advantages.

2.1. Overview of the AE schemes

We build our schemes in multiple layers, as depicted in Figure 1:

- At the bottom are the KECCAK-$p[1600, n_r = 24]$ and KECCAK-$p[1600, n_r = 12]$ permutations, on top of which we build everything else. These are the permutations used by the sponge functions SHAKE and TurboSHAKE, respectively, see Section 5.1.
- Then, we define a new construction, called *overwrite duplex* or OD, for building a duplex cipher, combining the advantages of duplex and overwrite sponge [1]. Each duplex cipher is parameterized by a permutation f and has a capacity c, and its security is equivalent to that of the sponge function with the same f and c, i.e., SHAKE128, SHAKE256, TurboSHAKE128 or TurboSHAKE256, see Section 8.
- Finally, on top of these, we build AE schemes with two different modes. The first set of schemes, defined in Section 9, use the authenticated encryption mode DWrap, that is similar to SpongeWrap [1] and that requires the unicity of the first AD field of the session. The second set of schemes, defined in Section 11, build upon the Deck-BO mode [14], a session-supporting version of the SIV AE mode. This mode in turn requires a deck cipher that we build with the UpperDeck mode on top of OD, and this is presented in Section 10.

2.2. Advantages

Layering is advantageous from the point of view of analysis, allowing easy security reduction proofs, and yet implementations remain simple. Thanks to the fact that TurboSHAKE, OD and UpperDeck accept payload data in byte string inputs and accumulate domain separation bits in a separate single-byte trailer, the multiple layers can be easily merged, as illustrated in Figures 3 and 5. The schemes can then be expressed in terms of some simple state operations and calls to the underlying permutation.

The modes we present have robustness and usability advantages, which may help security by simplifying the protocols that make use of them:

- They do not constrain the user in having to satisfy the uniqueness property of a distinct data element N, usually of short fixed size, but instead allow one to use the complete variable-length AD for that purpose.
- The Deck-BO mode has no nonce requirement: it only leaks equality of (AD, P) pairs through equal ciphertexts.
- Both modes support sessions. This, among others, relieves one from buffering long messages by instead using intermediate tags, reduces AD uniqueness constraints to the first AD of a session, and limits potential leakage in Deck-BO to sessions with equal leading (AD, P) pairs.

When instantiated with (Turbo)SHAKE, our schemes have the following security advantages:

- Their security is based on the security claim of a NIST standard that has received a large amount of public scrutiny [20]. As long as the underlying SHAKE and TurboSHAKE functions satisfy their accompanying security claim, the distinguishing advantage of resulting AE schemes from an ideal AE is negligible, even in a multi-user setup. Moreover, the intermediate schemes are hard to distinguish from (an object equivalent to) a random oracle.
- Unlike blockcipher-based modes, they do not suffer from the birthday bound, but instead fully achieve the security strength implied by that of the underlying XOF, the chosen key length and the tag length. E.g., schemes based on SHAKE128 offer a security strength of 128 bits.
- They offer CMT-4 committing security based on the collision resistance of (Turbo)SHAKE.

Finally, their implementations are competitive in terms of performance, both in software and in hardware. In software, this is mainly thanks to TurboSHAKE instances, while in pure hardware, KECCAK is famous for being extremely efficient. Furthermore, (Turbo)SHAKE AE can benefit from dedicated hardware support (e.g., in the recent Apple™ processors) and leverage synergies with implementations of KECCAK inside ML-KEM and ML-DSA [22], [23]. And, where it matters, its degree-2 round function lends itself well to protections like masking against side-channel attacks.

3. Comparison with prior art

In this section, we discuss some other AE schemes based on widespread standards, in particular based on long-time NIST standard AES [15], industry standard ChaCha20-Poly1305 [5] and upcoming lightweight cryptography NIST standard Ascon [6]. We summarize their properties in Table 1.

There are numerous AE schemes built using modes of operation on top of AES, and they are very efficient on platforms with dedicated AES instructions. However, simple AES-based schemes such as AES-GCM do not support sessions, have a dedicated short nonce (96 bits in the case of AES-GCM), can only achieve 64 bits of security strength due to the birthday bound and leak authentication keys in case of nonce repetition. Moreover, achieving committing security requires adding redundancy or cryptographic processing, see Appendix C.

ChaCha20-Poly1305 is an AE scheme based on a stream cipher (ChaCha20) and a one-time authenticator (Poly1305) that has received massive adoption mostly thanks to its excellent performance. However, it suffers from a short nonce (96 bits) and key leakage in case of nonce repetition, too. Moreover, it can only achieve 106 bits of security. Finally, ChaCha20-Poly1305 does not support sessions, nor it offers committing security [24].

Ascon is the family of algorithms selected by NIST for lightweight AE and hashing, and at the time of writing the standard is being drafted. Ascon borrows design ideas from SHA-3, such as the sponge construction and building blocks of the underlying permutation, to build a duplex-based AE scheme with a small memory footprint. Despite this, it does not support sessions, and its CMT-4 security is only 64 bits [25]. Ascon has a 128-bit nonce. If the nonce is repeated, it does not leak key material, but it may lead to partial or complete loss of confidentiality, depending on the attack [26] Unlike the other cryptosystems we discuss here, Ascon is not a general-purpose cipher: It is optimized to fit in constrained environments, rather than for performance on mainstream platforms.

TABLE 1. COMPARISON WITH STANDARD AE

Solution	Security strength	Nonce-misuse resistance	Session support	CMT-4 security
(Turbo)SHAKE128-Wrap	128 bits	no	yes	128 bits
(Turbo)SHAKE256-Wrap	256 bits	no	yes	256 bits
(Turbo)SHAKE128-BO	128 bits	yes	yes	128 bits
(Turbo)SHAKE256-BO	256 bits	yes	yes	256 bits
ChaCha20-Poly1305	106 bits	no	no	no
AES128-GCM	64 bits	no	no	no
AES128-GCM-SIV	64 bits	yes	no	no
Ascon-AEAD128	128 bits	no	no	64 bits

4. String processing

In this section, we introduce our conventions related to string processing, the representations of bit strings as pairs of byte strings and single-byte trailers and the parsing of byte strings into sequences of blocks.

4.1. Notation

We denote the empty string by ϵ. The concatenation of two strings X, Y is denoted as $X||Y$, and their bitwise addition as $X+Y$. For a byte string X, its length in bytes is denoted by $|X|$ and the bytes in a string are indexed from 0 to $|X|-1$. If $|X| \neq |Y|$, bytes with equal indices are added, and $X+Y$ has length $\min(|X|, |Y|)$.

Bit string values are noted with a typewriter font, such as `01101`. Byte values are noted with two hexadecimal digits in a typewriter font and preceded by `0x`, e.g., `0x1F`. The function $\mathrm{enc}_8(x)$ denotes the one-byte string with integer value x, e.g., $\mathrm{enc}_8(31) = \texttt{0x1F}$. The repetition of a bit is noted in exponent, e.g., $\texttt{0}^3 = \texttt{000}$. Similarly, for bytes, e.g., $\texttt{0x00}^3 = \texttt{0x00}||\texttt{0x00}||\texttt{0x00}$. We denote sequences of strings as $A; B; C \ldots$. In some cases we need to provide multiple strings $A, B, \ldots$ as input to a function that takes only a single string as input. Then we write $F(A; B)$ where $A; B$ denotes a string that is an injective encoding of strings A and B.

4.2. Byte strings and trailers

In real-world use cases, the inputs, i.e., keys, plaintexts, tags and associated data, are strings of bytes. Nevertheless, as we go down the stack of our modes and constructions as depicted in Figure 1, most layers append domain separation bits, and mandating byte strings at each level would imply that this extends the input string by one byte per layer. To avoid this blowup, we accumulate domain separation bits in a dedicated single-byte data element called *trailer*.

A trailer encodes a bit string of length between 0 and 7 that we call *trailerstring*. Clearly, a bit string of any length can be unambiguously converted to a pair of byte string and trailer, and vice-versa. As a convention throughout this paper, we often give as input such a pair, rather than a bit string, as it allows to clearly separate payload data (e.g., a block of plaintext) from domain separation bits. This split simplifies the implementation when different layers are merged, using constant values for trailers, as for instance in Figures 4 and 5.

In our algorithmic descriptions, we represent constant trailers as integer values, similarly to the approach taken in the definition of TurboSHAKE's domain separation byte D [21]. For a n-bit trailerstring $e = (e_0, e_1, \ldots, e_{n-1})$, we define the value of its trailer as $E = \mathrm{padint}(e)$, with

$$\mathrm{padint}(e) = 2^n + \sum_{i=0}^{n-1} 2^i e_i \ . \tag{1}$$

For instance, $\mathrm{padint}(\epsilon) = 1$ and $\mathrm{padint}(\texttt{011}) = 2^3 + 6 = 14$.

The inverse function, $\mathrm{unpad}(E)$, converts an integer $E \geq 1$ to a bit string e. The trailerstring e is obtained

by taking the representation of the integer E in base 2, least significant bit first, and removing its last bit (i.e., the most significant bit that is always 1 since $E \geq 1$). By limiting E to the range $[1, 255]$, we ensure that there is a one-to-one mapping between a pair of a byte string and a trailer (B, E) and a bit string X through the relation $X = B||\text{unpad}(E)$.

When merging layers, a typical case consists in appending a bit p to a trailerstring that comes from the upper layer with integer value E. The resulting trailer has integer value $E' = \text{padint}(\text{unpad}(E)||p)$. For readability, we abuse of notation and write the above expression as $E' = E||p$.

4.3. Parsing byte strings into blocks

In several places we need to split byte strings into a sequence of blocks, each short enough to serve as input to a duplexing call. We specify how we do this in Algorithm 1.

Algorithm 1 Definition of $\text{parse}(X, \ell_1, \ell_2)$

Input: Byte string X, length ℓ_1, and length ℓ_2
Output: sequence $\mathbf{x}$ of blocks $x_1, x_2, \ldots, x_{|\mathbf{x}|}$ of at least one block
Split X into a first block of ℓ_1 bytes and remaining blocks of ℓ_2 bytes. If $|X| \leq \ell_1$, the sequence $\mathbf{x}$ has a single block of length $|X|$. Otherwise, the last block of $\mathbf{x}$ may be shorter than ℓ_2.

5. SHAKE, TurboSHAKE and their security

In this section, we discuss the SHAKE and TurboSHAKE functions, their security claims and the security properties that follow from there, namely collision resistance and multi-user PRF security.

5.1. SHAKE

XOFs are similar to hash functions but support arbitrary-length outputs. SHAKE128 and SHAKE256 are two XOFs standardized by NIST in [20]. They are defined on top of the KECCAK$[c] \triangleq$ KECCAK$[1600 - c, c]$ sponge function. Internally, both use the 1600-bit 24-round permutation KECCAK-$p[1600, n_\text{r} = 24]$ and are parameterized by the capacity c, a quantity expressed in bits. The capacity determines the targeted security strength level, as $c/2$, as well as the efficiency since the number of bits a sponge function can absorb or squeeze per call to the underlying permutation is $r = 1600 - c$. Here, r the (bit) rate, and we denote with $r_B = r/8$ the rate in bytes. In particular, we have $c = 256$ for SHAKE128 and $c = 512$ for SHAKE256, giving byte rates of $r_B = 168$ and $r_B = 136$, respectively.

An instance of SHAKE takes as input a variable-length bit string M and an output length d and appends four bits to M before presenting it to KECCAK$[c]$. In particular,

$$\text{SHAKE128}(M, d) = \text{KECCAK}[256](M||\texttt{1111}, d) \text{ and}$$
$$\text{SHAKE256}(M, d) = \text{KECCAK}[512](M||\texttt{1111}, d)\,.$$

The SHAKE functions are equal to KECCAK$[c]$, where the input is padded with some domain separation bits, and the latter comes with a security claim attached, that we here repeat:

Claim 1 (Flat sponge claim [27])**.** *The expected success probability of any attack against* KECCAK$[c]$ *with a workload equivalent to t calls to* KECCAK-$p[1600, n_\text{r} = 24]$ *or its inverse shall be smaller than or equal to that for a random oracle plus*

$$1 - \exp\left(-t(t+1)2^{-(c+1)}\right)\,. \tag{2}$$

We exclude here weaknesses *due to the mere fact that* KECCAK-$p[1600, n_\text{r} = 24]$ *can be described compactly and can be efficiently executed, e.g., the so-called random oracle implementation impossibility [27, Section "The impossibility of implementing a random oracle"], as well as properties that cannot be modeled as a single-stage game [28].*

Informally, this claim says for capacity c, the success probability of any attack against KECCAK$[c]$ is at most $t^2/2^{c+1}$ higher than the same attack against a random oracle, with t the computational complexity expressed as the number of executions of the permutation or equivalent computations. In other words, KECCAK$[c]$, and therefore SHAKE shall offer the same security strength as a random oracle whenever that offers a strength below $c/2$ bits, and a strength of $c/2$ bits in all other cases.

5.2. TurboSHAKE

TurboSHAKE is a family of XOFs that was originally introduced for use in KANGAROOTWELVE [29] and later formally defined in [21]. As SHAKE, it is parameterized by the capacity c, but it is based on the 12-round permutation KECCAK-$p[1600, n_\text{r} = 12]$ instead. It was introduced to have a variant that, at the same time, is more efficient than *and* relies on all the cryptanalysis done for KECCAK. We consider two instances: TurboSHAKE128 with $c = 256$ and TurboSHAKE256 with $c = 512$.

An instance of TurboSHAKE takes as input a byte string of arbitrary length M and a trailer D in the range $[1, 127]$. TurboSHAKE forms the bit string X from M and the trailer D as $X \leftarrow M||\text{unpad}(D)$ and then applies the sponge function with permutation KECCAK-$p[1600, n_\text{r} = 12]$ and byte rate r_B to it. This is equivalent to the following processing: It appends the byte D to M and pads the resulting string with the minimum number (possibly zero) of bytes `0x00` until $M' = M||D||\texttt{0x00}^*$ has length a multiple of the rate r_B. Then, it bitwise adds the byte `0x80` to the last byte of M'. The resulting string M' represents the blocks that are absorbed by the sponge function.

TurboSHAKE comes with a claim like Claim 2, but with KECCAK-$p[1600, n_\text{r} = 24]$ replaced by KECCAK-$p[1600, n_\text{r} = 12]$ [21].

5.3. Collision resistance

The security claims for (Turbo)SHAKE imply collision resistance as a corollary: a (Turbo)SHAKE instance with capacity c and output length n is claimed to have

$\min(c/2, n/2)$ bits of collision resistance. This property has been challenged by cryptanalysists on reduced-round versions of (Turbo)SHAKE, as can be seen in the references of [21]. Quoting FIPS 202 [20]:

> The two SHA-3 XOFs are designed to resist collision, preimage, and second-preimage attacks, and other attacks that would be resisted by a random function of the requested output length, up to the security strength of 128 bits for SHAKE128, and 256 bits for SHAKE256.

6. General keyed security

It is customary to express security of cryptographic primitives in the *distinguishability framework*, where one bounds the advantage of an adversary $\mathcal{D}$ in distinguishing a real-world primitive from an ideal-world primitive (typically, a random oracle). The adversary is faced with one of them but does not know which, can query the primitive and make computations, and must at the end guess whether she was talking with the real-world or ideal world primitive.

Definition 1. *Let $\mathcal{O}, \mathcal{P}$ be two collections of oracles with the same interface, i.e., that support the same set of functions with the same input and output types. The advantage of an adversary $\mathcal{D}$ in distinguishing $\mathcal{O}$ from $\mathcal{P}$ is defined as*

$$\Delta_{\mathcal{D}}(\mathcal{O} \parallel \mathcal{P}) = \left|\Pr\left(\mathcal{D}^{\mathcal{O}} \to 1\right) - \Pr\left(\mathcal{D}^{\mathcal{P}} \to 1\right)\right| .$$

Here $\mathcal{D}$ is an algorithm that returns 0 or 1. Furthermore, if we replace the adversary with maximal resource specifications R, this means we are maximizing over all adversaries that use at most these resources,

$$\Delta_R(\mathcal{O} \parallel \mathcal{P}) = \max_{\mathcal{D} \,:\, \text{resources}(\mathcal{D}) \le R} \Delta_{\mathcal{D}}(\mathcal{O} \parallel \mathcal{P}) .$$

An important tool in security reductions is the triangle inequality [30]:

Lemma 1. *Let $\mathcal{O}, \mathcal{Q}$ and $\mathcal{P}$ be three collections of oracles with the same interface. The advantage of an adversary $\mathcal{D}$ in distinguishing $\mathcal{O}$ from $\mathcal{P}$ is upper bound by*

$$\Delta_{\mathcal{D}}(\mathcal{O} \parallel \mathcal{P}) \le \Delta_{\mathcal{D}}(\mathcal{O} \parallel \mathcal{Q}) + \Delta_{\mathcal{D}}(\mathcal{Q} \parallel \mathcal{P}) .$$

Furthermore, if we replace the adversary with maximal resource specifications, this means we are maximizing over all adversaries that use at most these resources,

$$\Delta_R(\mathcal{O} \parallel \mathcal{P}) \le \Delta_R(\mathcal{O} \parallel \mathcal{Q}) + \Delta_R(\mathcal{Q} \parallel \mathcal{P}) .$$

6.1. Security of a mode

The triangle equality is convenient in expressing the security of schemes that are obtained as modes from other primitives. Let $M[P]$ be a scheme obtained by instantiating the mode M with primitive P, $\mathcal{S}$ the ideal-world counterpart of the scheme and $\mathcal{I}$ that of the primitive. Then, applying the triangle inequality yields:

$$\Delta_{\mathcal{D}}(M[P] \parallel \mathcal{S}) \le \Delta_{\mathcal{D}}(M[\mathcal{I}] \parallel \mathcal{S}) + \Delta_{\mathcal{D}}(M[P] \parallel M[\mathcal{I}]) .$$

We have $\Delta_{\mathcal{D}}(M[P] \parallel M[\mathcal{I}]) \le \Delta_{\mathcal{D}'}(P \parallel \mathcal{I})$ as $\mathcal{D}$ queries to $M[P]$ can be emulated by an adversary $\mathcal{D}'$ with access to P, and similar for $M[\mathcal{I}]$ and $\mathcal{I}$. It follows that

$$\Delta_{\mathcal{D}}(M[P] \parallel \mathcal{S}) \le \Delta_{\mathcal{D}}(M[\mathcal{I}] \parallel \mathcal{S}) + \Delta_{\mathcal{D}'}(P \parallel \mathcal{I}) .$$

So, we can split up the distance in two terms: that between the primitive and its ideal counterpart and that between the mode calling an ideal primitive and its ideal counterpart.

Definition 2. *We denote* $\mathrm{Adv}^{\mathrm{idealS}}_{M[\cdot]}(R) \triangleq \Delta_{\mathcal{D}}(M[\mathcal{I}] \parallel \mathcal{S})$ *as the* advantage of the mode M.

With the advantage formalism, we thus have:

$$\mathrm{Adv}^{\mathrm{idealS}}_{M[P]}(R) \le \mathrm{Adv}^{\mathrm{idealS}}_{M[\cdot]}(R) + \mathrm{Adv}^{\mathrm{idealP}}_{P}(R) . \quad (3)$$

6.2. Multi-user security and key distributions

For all primitives and modes we discuss in this paper we consider multi-user security. We do this by defining the keying of primitives through a *key scheme*.

A key scheme is based on a key distribution $\mathcal{K}$ and works similarly to a random oracle. We assume the existence of a finite set ID of possible identifiers. Users can query the key scheme for a key by sending it an identifier $ID \in \mathsf{ID}$. If this is the first time the key scheme is queried with that ID, it generates the key $K[ID]$ according to the distribution $\mathcal{K}(ID)$, i.e., the distribution may depend on ID but different draws are independent. Otherwise, it returns the previously drawn key $K[ID]$. We will indicate a key scheme with its corresponding key distribution $\mathcal{K}$.

Keying a primitive P with a key scheme $\mathcal{K}$ results in a cryptographic scheme that we denote as $P[\mathcal{K}]$. In such a scheme, a user (or adversary) can *create* new primitive instances with a $\mathsf{create}()$ operation that takes as argument an identifier ID: $P[K[ID]] \leftarrow \mathsf{create}(P, ID)$. We call primitive instances with the same ID *coupled* as they share the same key: Alice and Bob can each have the member of a coupled pair of primitive to secure their communications. When a scheme $P[\mathcal{K}]$ is being used, we denote the number of keys in use by μ and call it the *key multiplicity*. It equals the number of different ID values occurring in queries to the key scheme.

We adopt metrics similar to those defined in [31, Section 2.1] to characterize the key distribution $\mathcal{K}$ and its impact on our security bounds. First, we define the *multi-target min-entropy* of order μ as:

$$H_{\mathrm{mtmin}}[\mu](\mathcal{K}) = \\ -\log_2 \max_{S \subseteq \mathsf{ID} \,:\, |S| = \mu} \max_{K^*} \Pr\left(K^* \in \{K[ID] : ID \in S\}\right) .$$

This equals the success probability of guessing one of μ keys selected according to $\mathcal{K}$ by an adversary that can choose the ID values. Second, we define its *collision entropy* as

$$H_{\mathrm{coll}}(\mathcal{K}) = -\log_2 \max_{ID \neq ID'} \Pr(K[ID] = K[ID']) .$$

We now express the multi-user pseudorandom function (mPRF) security of a function F as an upper bound of the advantage of distinguishing $F[\mathcal{K}]$ from an ideal-world counterpart $\mathcal{RO}[\mathcal{ID}]$. The latter is a scheme of random oracles $\mathcal{RO}(ID; \cdot)$, domain separated with ID between instances and hence returning independent responses. This

corresponds with a simple zero-entropy key scheme where $K[ID] = ID$ that we denote by $\mathcal{ID}$. We say the random oracle is *indexed* with $\mathcal{ID}$ and denote this as $\mathcal{RO}[\mathcal{ID}]$.

Definition 3 (multi-user PRF advantage)**.** *The* multi-user PRF advantage *of a function F keyed with $\mathcal{K}$ is defined as:*

$$\mathrm{Adv}^{\mathrm{mPRF}}_{F[\mathcal{K}]}(R) = \Delta_R(F[\mathcal{K}] \parallel \mathcal{RO}[\mathcal{ID}])\,,$$

with R the adversarial resources.

A key distribution $\mathcal{K}$ introduces two terms in the multi-user PRF advantage of any deterministic primitive that generates output depending on the used key. To isolate them, we compare a real-world system with an ideal-world system that can only be distinguished due to the application of a key scheme $\mathcal{K}$. The real-world system has a random oracle $\mathcal{RO}$ keyed with $\mathcal{K}$ and that same random oracle $\mathcal{RO}$ unkeyed. The ideal-world system has a random oracle $\mathcal{RO}'$ (different from $\mathcal{RO}$) indexed with $\mathcal{ID}$ and $\mathcal{RO}$ unkeyed.

The adversarial resources depend on the primitive at hand but at this abstraction level we can provide following definition.

Definition 4. *The generic adversarial resources are the following:*

- *μ, the key multiplicity.*
- *q, the computational complexity, counts the number of queries to $\mathcal{RO}$, the underlying primitive that is considered a public function.*
- *σ, the data complexity, counts the number of construction queries to either $\mathcal{RO}[\mathcal{K}]$ or $\mathcal{RO}'[\mathcal{ID}]$.*

When dealing with more concrete primitives, we may need to translate q into an equivalent computational complexity t. We now define the security of a key distribution expressed in these resources.

Definition 5 (multi-key advantage)**.** *The* multi-key advantage *of a key distribution $\mathcal{K}$ is the advantage of distinguishing real-world scheme $(\mathcal{RO}[\mathcal{K}], \mathcal{RO})$ from ideal-world scheme $(\mathcal{RO}'[\mathcal{ID}], \mathcal{RO})$. We denote this by $\mathrm{Adv}^{\mathrm{mKey}}_{\mathcal{K}}(\mu, q)$, with the adversarial resources defined in Definition 4.*

Clearly, in the ideal-world scheme, all random oracle instances return independent responses. In the real-world scheme, this is not the case, and we can bound the multi-key advantage.

Theorem 1. *The multi-key advantage of a key distribution $\mathcal{K}$ is upper bound by:*

$$\mathrm{Adv}^{\mathrm{mKey}}_{\mathcal{K}}(\mu, q) \le \frac{q}{2^{H_{\mathrm{mtmin}}[\mu](\mathcal{K})}} + \frac{\binom{\mu}{2}}{2^{H_{\mathrm{coll}}(\mathcal{K})}}\,, \quad (4)$$

with the adversarial resources defined in Definition 4.

Proof. The adversary $\mathcal{D}$ has to tell $\mathcal{RO}(K[ID]; x)$ and $\mathcal{RO}'(ID; x)$ apart, with (ID, x) pairs of its choice. We define two generic bad events and argue that, if they do not occur, $\mathcal{RO}(K[ID]; x)$ and $\mathcal{RO}'(ID; x)$ cannot be distinguished. The two bad events are as follows.

- The first is *key guessing*, that is, the event that the adversary queries $\mathcal{RO}(K^*; x)$ through the public primitive $\mathcal{RO}$, with K^* one of the μ keys $K[ID]$ and some string x. For a given key candidate K^*, the probability that there exists one key among the μ keys with this value is upper bounded by $2^{-H_{\mathrm{mtmin}}[\mu](\mathcal{K})}$. This probability of guessing one of the μ keys correctly after q attempts is at most $q2^{-H_{\mathrm{mtmin}}[\mu](\mathcal{K})}$.
- The second is *key collision*, that is, the event that two keys among μ are equal, i.e., $K[ID] = K[ID']$ with $ID \neq ID'$. The probability of such a collision is at most $\binom{\mu}{2}2^{-H_{\mathrm{coll}}(\mathcal{K})}$.

On the condition that these bad events do not occur, there are no colliding inputs, the outputs are all independent, and hence the adversary cannot distinguish between the two worlds. Therefore,

$$\Delta_{\mathcal{D}}((\mathcal{RO}[\mathcal{K}], \mathcal{RO}) \parallel (\mathcal{RO}'[\mathcal{ID}], \mathcal{RO})) \le \frac{q}{2^{H_{\mathrm{mtmin}}[\mu](\mathcal{K})}} + \frac{\binom{\mu}{2}}{2^{H_{\mathrm{coll}}(\mathcal{K})}}\,.$$

□

6.3. Two special key distributions

We will consider two particular distributions, each parametrized by a key length k:

- $\mathcal{U}[k]$: keys $K[ID]$ are drawn uniformly from $\mathbb{Z}_2^k$.
- $\mathcal{I}[k]$: keys $K[ID]$ are the concatenation of a key drawn uniformly from $\mathbb{Z}_2^k$ and ID as a string.

For $\mathcal{U}[k]$, we have that for any arbitrary ID and any S with $|S| = \mu$:

$$\Pr(K^* \in \{K[ID] : ID \in S\}) \le \mu \Pr(K^* = K[ID])\,,$$

and hence $H_{\mathrm{mtmin}}[\mu](\mathcal{U}[k]) \ge k - \log_2 \mu$. This shows the degradation of the security by $\log_2 \mu$ bits, and the probability of guessing one of the keys is $\mu t/2^k$ in this case. Also, the probability of collision between two given keys is 2^{-k}, hence $H_{\mathrm{coll}}(\mathcal{U}[k]) = k$, and among μ keys this becomes $\binom{\mu}{2}2^{-k}$.

Theorem 2. *The multi-key advantage of key distribution $\mathcal{U}[k]$ is upper bound by:*

$$\mathrm{Adv}^{\mathrm{mKey}}_{\mathcal{U}[k]}(\mu, q) \le \frac{\mu q}{2^k} + \frac{\binom{\mu}{2}}{2^k}\,.$$

The distribution $\mathcal{I}[k]$ avoids this degradation. Here, for a key candidate K^* to match a key $K[ID]$, the last bits must match the ID, and $\Pr(K^* \in \{K[ID] : ID \in S\}) = \Pr(K[ID^*] = K^*) = 2^{-k}$, with ID^* the ID that is encoded in K^*. Hence, $H_{\mathrm{mtmin}}[\mu](\mathcal{I}[k]) = k$, and the probability of guessing one of the keys is $q/2^k$ in this case, regardless of the number of users. Moreover, the presence of the unique key ID prevents collisions, so the collision term vanishes.

Theorem 3. *The multi-key advantage of key distribution $\mathcal{I}[k]$ is upper bound by:*

$$\mathrm{Adv}^{\mathrm{mKey}}_{\mathcal{I}[k]}(\mu, q) \le \frac{q}{2^k}\,.$$

6.4. Security of keyed (Turbo)SHAKE

We now discuss the keyed security of the SHAKE and TurboSHAKE functions. For this, we refine the adversarial resources to the case of sponge functions or stateful counterparts thereof; this is what the next definition gives.

Definition 6 (adversarial resources for sponge functions and stateful equivalents)**.** *For a function calling a permutation we define the adversarial resources as*

- μ*: as in Definition 4*
- t*: computational complexity expressed in the number of evaluations of the permutations or equivalent computations,*
- σ*: data complexity expressed in total number of input and output blocks in queries to the function or object, taking into account that common full-block prefixes with a given key are counted only once.*

Note that the q in the expressions for the multi-key advantage maps to t by $t \geq q$ in the case of sponge functions. If $K[ID]$ can be absorbed in a single block this simplifies to $t = q$. We will make that assumption in the remainder of the paper.

Multi-user PRF security claims for specific functions such as (Turbo)SHAKE get credibility through public scrutiny by cryptanalists. As a matter of fact, since its publication, there has been plenty of cryptanalysis of reduced-round KECCAK in the keyed setting that provides evidence for the multi-user PRF security of (Turbo)SHAKE, see [21] for references. Still, we can prove multi-user PRF security bounds for (Turbo)SHAKE if we assume it stands by it security claim.

Let F stand for any (Turbo)SHAKE function with capacity c. We study the security of $F[\mathcal{K}]$ and denote by $F_{K[ID]}$ the function defined as $F_{K[ID]}(M) = F(K[ID]; M)$.

Theorem 4. *On the condition that (Turbo)SHAKE stands by its claimed security [21], [32], the multi-user PRF advantage of keyed (Turbo)SHAKE with key distribution $\mathcal{K}$ is upper bounded as*

$$\mathrm{Adv}^{\mathrm{mPRF}}_{F[\mathcal{K}]}(\mu, t, \sigma) \leq \mathrm{Adv}^{\mathrm{mKey}}_{\mathcal{K}}(\mu, t) + \frac{(t+\sigma)^2}{2^{c+1}}, \quad (5)$$

where c is the capacity and μ, t, σ are the adversarial resources as defined in Definition 6.

Proof. We apply the security claim to the attack of distinguishing $F[\mathcal{K}]$ from $\mathcal{RO}'[\mathcal{ID}]$. The claim then states that this distinguishing problem shall not have an advantage greater than the term in (2) plus the distinguishing problem with F replaced with a random oracle. The latter is the advantage of distinguishing $(\mathcal{RO}[\mathcal{K}], \mathcal{RO})$ from $(\mathcal{RO}'[\mathcal{ID}], \mathcal{RO})$ that we know as $\mathrm{Adv}^{\mathrm{mKey}}_{\mathcal{K}}$. The term (2) expresses the complexity of queries to F. First, we upper bound it with $t(t+1)/2^{c+1}$ and approximate it as $t^2/2^{c+1}$. The complexity t in (2) here translates to queries to $F[\mathcal{K}]$ or to computations expressed in permutation calls or equivalent. Therefore it maps to the sum of the computational and data complexity $t + \sigma$. □

7. Stateful primitives and their security

In this section, we provide definitions of stateful objects and schemes together with their security notions. To this end, we recall the jammin cipher and use it to define the security of (session-supporting) AE schemes. Similarly, we define idaho as the stateful counterpart of the random oracle and use it to define the security of some stateful objects, in particular, of the deck and duplex ciphers that share the idaho interface.

7.1. Stateful objects and schemes

Many cryptographic primitives like hash functions, block ciphers and stream ciphers, are *functions* in the sense that they produce an output upon receiving an input, and their job stops there. There are, however, cases where one needs to keep state and to have a primitive whose processing depends on the state and makes it evolve. In those cases, we speak about *objects* that have *attributes* and support *methods*: The attributes represent the state, and the methods are functions that perform processing on the attributes and may have input and return output. We can apply the keying specified in Section 6.2 to objects instead of functions: A *scheme* is obtained by keying an object with a key distribution $\mathcal{K}$. In such a scheme, the object can be instantiated multiple times, and we speak of (object) *instances*.

Two objects that are of a different type may have the same *interface*, where the interface is the set of methods, and possibly a parameter, they support. In distinguishing setups, it is required that the real-world object and the ideal-world object support the same interface as otherwise, the objects can be distinguished trivially.

An example is the Duplex construction [1] that defines an object with which one can hash strings of limited lengths and get a digest of the sequence of strings received so far. Here, the state is the object's attribute and its methods consist of initialization and duplexing.

Another example is that of session-supporting authentication encryption primitives. The wrap and unwrap operations are performed by objects that keep track of past operations in attributes.

Finally we have the *doubly-extendable cryptographic keyed*, or *deck*, function, a keyed primitive that supports as input a sequence of variable-length strings and can output a string of arbitrary length [19]. Examples of deck functions are KRAVATTE and XOOFFF, two instances of the Farfalle construction based on the KECCAK-p and XOODOO permutations, respectively [19], [33]. One of the main properties of deck functions is extendability: the cost of computing $D_K(X, Y)$ depends only on the processing of Y if $D_K(X)$ was previously computed. This can be modeled in a natural way by seeing the deck function as an object that we call *deck cipher*. A *deck cipher* is an object that holds the state necessary to incrementally evaluate the deck function for each input string. A deck cipher is functionally equivalent to a deck function but the object form better reflects its implementation and typical usage. In short, a deck cipher is to its deck function what a duplex object is to its sponge function.

For instance, the description of Deck-BO in [14] makes use of a variable, called *history*, that starts with the empty sequence and then accumulates strings. A typical pattern consists in first updating the history and then evaluating the deck function with the history as input:

$$\begin{aligned} \text{history} &\leftarrow \text{history}, X \\ Y &\leftarrow 0^\ell + D_K(\text{history}) \end{aligned} \quad (6)$$

In a concrete implementation, however, the history is not materialized as an actual sequence of strings. Instead, a deck cipher keeps state and offers a duplexing method that replaces (6). In addition, we replace bit strings with pairs

of byte strings and trailers. So, most generally, the deck function evaluation

$$Y \leftarrow 0^{\ell} + D_{K[ID]}(X_1||\mathrm{unpad}(E_1), \ldots, X_n||\mathrm{unpad}(E_n))$$

becomes the following sequence with a deck cipher:

$$\begin{aligned} &\mathrm{D} \leftarrow \mathsf{create}(\ldots, ID)\\ &\mathrm{D}.\mathsf{duplexing}(X_1, E_1)\\ &\ldots\\ &Y \leftarrow 0^{\ell} + \mathrm{D}.\mathsf{duplexing}(X_n, E_n)\,. \end{aligned}$$

7.2. AE and the jammin cipher

For the AE modes and schemes of this paper, we express the security as the difficulty of distinguishing from the *jammin cipher* [14]. The jammin cipher is a scheme obtained by indexing a jammin object by $\mathcal{ID}$ resulting in instances that output random responses to wrap() calls, correct plaintexts as a response to unwrap() calls for valid ciphertexts, and errors for invalid ciphertexts. The jammin cipher is parameterized by a *ciphertext expansion* function $\mathrm{WrapExpand}()$ that expresses the length of the ciphertext given the length of the plaintext. For the modes in this paper, we have $|C| = \mathrm{WrapExpand}(|P|) = |P| + \tau/8$. The jammin cipher has no nonce field.

We chose the jammin cipher for the following reasons. First, most ideal-world AE schemes return an error upon any unwrap call, and in the distinguishing experiment the adversary is not allowed to do unwrap queries with ciphertext that is the result of wrap queries. This makes them *non-operational*. In the world of session-supporting AE this has led to the simultaneous definition of operational ideal-world schemes that are hard to work with and non-operational ideal-world schemes that are used in distinguishing experiments [34]. Unfortunately, these two types of ideal-world schemes are not equivalent and leave a security gap. This issue was resolved by the jammin cipher [14] as it is *operational*: it supports all functions that a real-world scheme does, but with ideal behaviour. Namely, it achieves the highest possible security: The probability of forgery is 0 and the ciphertexts it produces are as random as injectivity allows, while behaving deterministically, meaning equal inputs give equal outputs.

Second, the jammin cipher is inherently multi-user in that it supports multiple instances that can exchange encrypted messages on the condition that they are coupled. Any of such instances is able to process wrap and unwrap calls in any order.

Third, the jammin cipher supports sessions, and of course it can also serve as an ideal world scheme for AE schemes that do not support sessions by limiting each session to a single message (AD, P).

In the jammin cipher, the *encryption context* of a wrap() query to an instance is the sequence composed of the (AD, P) inputs received during the previous wrap() and unwrap() queries in a session and of the AD value of the current wrap() query. Also, we say that the *encryption context is a nonce* iff all wrap() queries with non-empty plaintext have a different encryption context.

We define the *pseudo-jammin cipher* (PJC) advantage of an AE scheme as the advantage of distinguishing it from the jammin cipher. If the AE scheme, for its security, requires the context to be a nonce, we speak of a *nonce-based* PJC (nPJC). The jammin cipher and PJC-secure AE schemes leak information due to the fact that equal ciphertexts with equal encryption contexts indicate equal plaintexts. If that is a concern, the user shall guarantee encryption context uniqueness. One set of the schemes that we define have nonce-based PJC security, while the others have plain PJC security.

A full specification of the jammin cipher and more explanations can be found in Appendix B.

Definition 7 (PJC advantage)**.** *Let* $\mathrm{AE}[\mathcal{K}]$ *be an AE scheme keyed with key distribution* $\mathcal{K}$ *and* $\mathcal{J}^+$ *the jammin cipher with the same ciphertext expansion as* $\mathrm{AE}[\mathcal{K}]$*. We denote the PJC advantage of* $\mathrm{AE}[\mathcal{K}]$ *by:*

$$\mathrm{Adv}^{\mathrm{PJC}}_{\mathrm{AE}[\mathcal{K}]}(R) = \Delta_R(\mathrm{AE}[\mathcal{K}] \parallel \mathcal{J}^+)\,.$$

with R *the adversarial resources.*

Definition 8 (nPJC advantage)**.** *Let* $\mathrm{AE}[\mathcal{K}]$ *be an AE scheme keyed with key distribution* $\mathcal{K}$ *and* $\mathcal{J}^+$ *the jammin cipher with the same ciphertext expansion as* $\mathrm{AE}[\mathcal{K}]$*. We denote the nPJC advantage of* $\mathrm{AE}[\mathcal{K}]$ *by:*

$$\mathrm{Adv}^{\mathrm{nPJC}}_{\mathrm{AE}[\mathcal{K}]}(R) = \Delta_R(\mathrm{AE}[\mathcal{K}] \parallel \mathcal{J}^+)\,.$$

where all wrap queries with non-empty plaintext have a different encryption context and with R *the adversarial resources.*

For AE schemes with a fixed-length nonce field, one may need a more restrictive nonce requirement on the encryption context These can be accommodated by re-casting N to the first $|N|$ bytes of AD and stipulating that the first $|N|$ bytes of AD shall be a nonce. Variants of nPJC security may be defined accordingly, with the exact requirement specified explicitly.

7.3. Idaho, duplex ciphers and deck ciphers

We introduce the object counterpart of the random oracle in Algorithm 2 and call it an *ideal extendable hashing object* (idaho). It will serve both as our ideal model for duplex and deck ciphers and as a blueprint of the interface those objects shall support.

An idaho object simply remembers the sequence of input strings received so far, called the *path*, and produces outputs by calling a random oracle $\mathcal{RO}$ with that path as input. Additionally, it provides output progressively with strings of user-chosen length. The attributes of the idaho simply consist of the path and of the offset in the random oracle's output.

After creation, an idaho instance does not support squeezing() call before any duplexing() call. Moreover, one can use clone() to create a clone of an idaho instance with a copy of its attributes or a variant called cloneCompact() that does not allow squeezing() before any duplexing() call.

The two other methods an idaho object supports are duplexing() and squeezing(). The former takes as input a byte string B (typically the payload) and a trailer E and returns a string of ℓ bytes. These two inputs are appended to the current path, then the output is formed by calling $\mathcal{RO}$ with the path as input. The latter extends the output of the previous duplexing() call, i.e., calling

Algorithm 2 Definition of IDAHO$[\rho]$

Parameter:
block length (in bytes) ρ, a positive integer, or ∞ to denote a practically unrestricted block length

Object attributes:
path, a sequence of bit strings (each encoded as a byte string and a trailer)
output offset o, an integer

Constructor: I $\leftarrow$ create(IDAHO$[\rho]$, ID)
return I: IDAHO object with $(\text{path}, o) = (ID, \rho)$

Method I $\leftarrow$ IDAHO.clone()
return I: a clone of IDAHO

Method I $\leftarrow$ IDAHO.cloneCompact()
return I: a clone of IDAHO but with $o = \rho$

Method $Z \leftarrow$ IDAHO.duplexing$(B, E)[\ell]$ with $\ell \leq \rho$ with B a byte string and E a trailer
path $\leftarrow$ path$; B; E$
$o \leftarrow \ell$
return the first ℓ bytes of $\mathcal{RO}(\text{path})$

Method $Z \leftarrow$ IDAHO.squeezing()$[\ell]$ with $\ell \leq \rho - o$
return ℓ bytes of $\mathcal{RO}(\text{path})$ starting from offset o and update $o \leftarrow o + \ell$

duplexing$(B, E)[\ell_1]$ followed by squeezing()$[\ell_2]$ yields the same result as calling duplexing$(B, E)[\ell_1 + \ell_2]$.

To keep the notation light and avoid using the square brackets, we assume for both methods that the output length ℓ is determined from the context. Specifically, when writing $X +$ duplexing(B, E), it is understood that $\ell = |X|$, and similarly $0^{8\ell} +$ squeezing() returns ℓ bytes.

A real-world scheme may impose restrictions on length of input and output strings to its ciphers. To serve as an ideal-world scheme for such schemes, idaho has the ρ parameter. When $\rho \neq \infty$, the idaho object enforces that $|B| \leq \rho$ in any duplexing(B, E) call, and the joint outputs of a duplexing() call and subsequent squeezing() calls is limited to to a total of ρ bytes. The method cloneCompact() also originates from a concern in real-world ciphers: if there is no need for squeezing before the next duplexing call by the clone, the cloning process can be made more efficient.

Definition 9 (Idaho interface)**.** *The* idaho interface *refers to the set of methods and a parameter supported by an object. It comprises the set of methods* clone(), cloneCompact(), duplexing() *and* squeezing() *and a parameter called* ρ*.*

We distinguish between two kinds of ciphers that support the idaho interface:

duplex cipher has $\rho \neq \infty$ relatively small. Notably, the name duplex (object) is used for the (keyed) Duplex construction defined in [35]. However, in this paper, we overload the duplex term and consider as such also objects that would be constructed differently.

deck cipher has $\rho = \infty$ meaning a value so huge that in practice it is never reached. Where a deck function accepts as input a sequence of strings $M_1; M_2; M_3; \ldots$, a deck cipher absorbs them one by one in separate duplexing() calls.

Both types of ciphers can be turned into schemes by keying them with a key distribution.

7.4. Security of keyed objects

When defining the security of an object by the advantage of distinguishing its corresponding scheme from an idaho scheme, we call this the *idaho advantage*. In Definition 5, we compare a keyed function to an indexed random oracle. Here, we compare a stateful object to the idaho object instead. Arguably, this is only a syntactic change since the output of the duplexing() and squeezing() methods is, in the end, a function of the key and of the sequence of inputs (in the real world) or a call to the random oracle (in the ideal world).

In the case of a function F with multi-user PRF security, the adversary can query instances $F_{K[ID]}$ with chosen input and chosen ID values. To give a level of freedom to the adversary with an object-based scheme that covers that, we allow the adversary to create instances with chosen ID, clone them with clone() or cloneCompact() and query the resulting instances with chosen input.

Definition 10 (idaho advantage)**.** *The idaho advantage of an object O keyed with key distribution $\mathcal{K}$ and block length ρ is defined as the advantage of distinguishing $O[\mathcal{K}]$ from the idaho object indexed by $\mathcal{ID}$ with the same block length ρ, that is,*

$$\text{Adv}^{\text{idaho}}_{O[\mathcal{K}]}(R) = \Delta_R(O[\mathcal{K}] \parallel \text{IDAHO}[\rho][\mathcal{ID}])\,,$$

with R the adversarial resources.

8. The overwrite duplex construction

In this section, we define the *overwrite duplex* (OD) construction, that can be seen as a (restricted) interface to a sponge function. In a nutshell, the OD construction builds a duplex cipher with an idaho interface.

OD combines the ideas of the duplex and overwrite constructions, both introduced in [1]. We define the OD construction in terms of the permutation underlying the corresponding sponge function and prove that the security strength of OD is at least that of the corresponding sponge function, both as a keyed function and for its collision resistance.

8.1. Specification of OD

An OD cipher is parameterized with a permutation f, a payload block length ρ and a trailer encoding function trailenc. An OD cipher acts as a keyed function of all the inputs received so far, where each duplexing call takes as input a string B with $|B| \leq \rho$ and a trailer $E \in [1, 63]$, and returns up to ρ bytes of output. The cipher keeps track of how many bytes it returned, and the squeezing method allows returning more output in between duplexing calls. cloneCompact() needs to copy only the last $b - 8\rho$ bits of the state, with b the width of the permutation. See Figure 2 for an illustration.

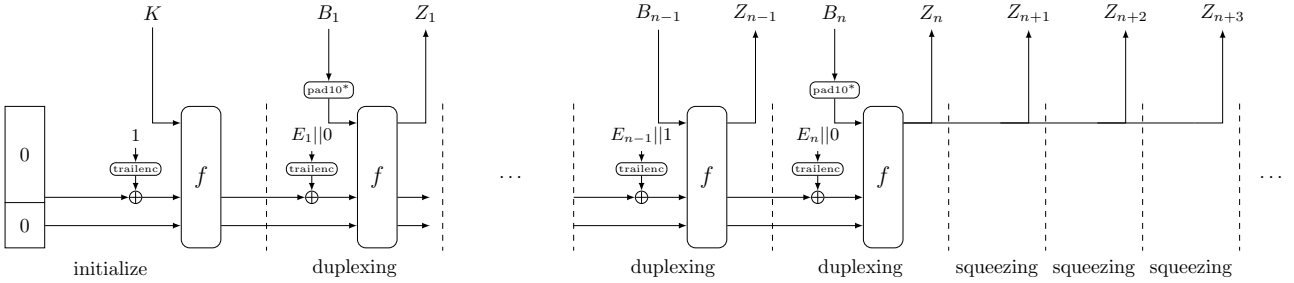

Figure 2. Illustration of the OD construction.

Algorithm 3 Definition of $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$

Parameters:
permutation f operating on b-bit strings
block length ρ, an integer
trailer encoding function $\mathrm{trailenc}()$ taking an integer in range $[1, 127]$ and returning a byte string

Object attributes:
state s, a b-bit string
output index o, an integer

Constructor: $\mathrm{O} \leftarrow \mathsf{create}(\mathrm{OD}[f, \rho, \mathrm{trailenc}], ID)$
O with $(s, o) \leftarrow (0^b, \rho)$
$\mathrm{O}.\mathsf{duplexing}(K[ID], 1)$
return O

Method $\mathrm{O} \leftarrow \mathrm{OD}.\mathsf{clone}()$
return O: a clone of OD

Method $\mathrm{O} \leftarrow \mathrm{OD}.\mathsf{cloneCompact}()$
return O: a clone of OD with $o = \rho$

Method $Z \leftarrow \mathrm{OD}.\mathsf{duplexing}(B, E)[\ell]$ with $|B| \leq \rho$ and $E \leq 63$
if $|B| = \rho$ **then**
  Replace the first ρ bytes of s with B
  XOR the next bytes of s with $\mathrm{trailenc}(E||1)$
else
  Replace the first ρ bytes of s with $\mathrm{pad10}^*(B)$
  XOR the next bytes of s with $\mathrm{trailenc}(E||0)$
$s \leftarrow f(s)$
$o \leftarrow \ell$
return the first ℓ bytes of s

Method $Z \leftarrow \mathrm{OD}.\mathsf{squeezing}()[\ell]$ with $\ell \leq \rho - o$
return ℓ bytes of s starting from offset o and update $o \leftarrow o + \ell$

Algorithm 3 defines the OD construction and uses the following conventions. While a duplexing call overwrites part of the state with the (padded) payload input string, it (bitwise) adds the input trailer after applying to it an encoding function trailenc. For an input string B shorter than ρ bytes, it applies padding such that it results in a ρ-byte block. We denote the padding rule as $\mathrm{pad10}^*(B)$ and we have $\mathrm{pad10}^*(B) = B||\texttt{0x01}||\texttt{0x00}^*$. OD does not apply padding to input strings of exactly ρ bytes, but distinguishes between padded and not-padded blocks using domain separation by adapting the trailer E before adding it to the state. Namely, it absorbs $\mathrm{trailenc}(D)$ with $D = E||0$ or $D = E||1$ depending if padding occurs or not, respectively.

The output of an OD cipher to the n-th duplexing call is fully determined by the sequence of inputs $(B_1; E_1; \ldots; B_n; E_n)$ that it received in the n duplexing calls $\mathrm{OD}.\mathsf{duplexing}(B_i, E_i)$ since its creation. The output of any intermediate $\mathrm{OD}.\mathsf{squeezing}()$ call can be seen as the delayed output of the most recent duplexing call. Moreover, the output lengths $\ell_i \leq \rho$ for $i < n$ do not influence the output of the n-th duplexing call.

8.2. OD applied to (Turbo)SHAKE

For SHAKE and TurboSHAKE, f is KECCAK-p[1600] with 24 or 12 rounds, respectively. The capacity is $c = 256$ for 128-bit security strength and $c = 512$ for 256-bit security strength. Finally, the block length is $\rho = (1600 - c - 64)/8$. The term 64 in the formula of ρ is given by the trailer encoding function, that outputs a string of 8 bytes specific for the underlying sponge function. For TurboSHAKE, we have

$$\mathrm{trailenc}(D) = \mathrm{enc}_8(D)||\texttt{0x00}^6||\texttt{0x80}\,,$$

while for SHAKE, this is slightly different to account for the suffix `1111` that FIPS 202 appends to the input string:

$$\mathrm{trailenc}(D) = \mathrm{enc}_8(D)||\texttt{0x00}^6||\texttt{0x9F}\,.$$

The value `0x9F` comes from the combination of the suffix and the 10^*1 padding.

The format of $\mathrm{trailenc}(D)$ is chosen so as to match that of TurboSHAKE's domain separation byte and padding as shown in Section 5.1. In this way, the output of the first duplexing call $\mathrm{OD}.\mathsf{duplexing}(B, E)[\ell]$ equals the output of $\mathrm{TurboSHAKE}[c](B, E||1)$ or $\mathrm{TurboSHAKE}[c](\mathrm{pad10}^*(B), E||0)$ truncated to ℓ bytes. See Lemma 2 for more details.

We now show that an $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$ cipher can be seen as a restricted interface to a (Turbo)SHAKE instance F with permutation f and capacity $c = 1600 - 64 - 8\rho$. Consequently, it inherits the collision resistance of F, and its idaho advantage is the same as the multi-user PRF advantage of F. This is formalized in Lemma 2. Without loss of generality, we treat the case of *full-block outputs*, that is, output blocks of ρ bytes.

Lemma 2. *Let F be a (Turbo)SHAKE function and its underlying permutation f. The full-block output of the n-th duplexing call to* $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$ *is the ρ-byte output of F applied to an input that is an injective mapping of $(B_1, E_1, \ldots, B_n, E_n)$. In addition, the input to F has B_1 as a prefix.*

Proof. For simplicity, we focus on the case that F is TurboSHAKE128, but the proofs for TurboSHAKE256, SHAKE128, SHAKE256 or any sponge function are essentially the same.

We first preprocess the sequence $(B_1, E_1, \ldots, B_n, E_n)$ by applying the padding to blocks B_i shorter than ρ bytes and transforming E_i accordingly, as the OD object does during duplexing calls. We call the resulting sequence $(\beta_1, D_1, \ldots, \beta_n, D_n)$. More precisely, if $|B_i| < \rho$, $\beta_i \leftarrow \mathrm{pad10^*}(B_i)$ and $D_i = E_i||0$. Otherwise $\beta_i \leftarrow B_i$ and $D_i = E_i||1$. As the last bit of $\mathrm{unpad}(D_i)$ indicates whether padding was applied and the padding itself is injective, this mapping is injective.

We denote by $\mathrm{TS}(M, D)$ the output of TurboSHAKE128 with byte string M and trailer D as inputs, truncated to its first ρ bytes, and by $\overline{\mathrm{OD}}(\beta_1, D_1, \ldots, \beta_n, D_n)$ the full-block output of OD to the preprocessed input sequence $(\beta_1, D_1, \ldots, \beta_n, D_n)$.

We first prove the theorem for $n = 1$ by expressing $\overline{\mathrm{OD}}(\beta_1, D_1)$ as TurboSHAKE128 applied to an input that is an injective mapping of (β_1, D_1), and then proceed recursively.

Before the first duplexing call the state of the OD object is all-zero and overwriting equals XORing. We XOR $\beta_1||D_1$, in total $\rho + 1$ bytes, that fits in a single $(b - c)$-bit block. From the TurboSHAKE128 specifications, we see that for a single-block $\overline{\mathrm{OD}}(\beta_1, D_1) = \mathrm{TS}(\beta_1, D_1)$. Clearly, the mapping from (β_1, D_1) to the TurboSHAKE128 input is injective, and this shows that B_1 is a prefix of the input.

For the second duplexing call, we need to take into account a major difference between the OD object and the plain sponge construction underlying TurboSHAKE: The former overwrites the input block in the state, while the latter XORs it. Referring to [1], overwriting the (outer part of) the state is actually equivalent to first XORing the block with the previous output and then XORing the result into the state. This can be expressed as follows:

$$\overline{\mathrm{OD}}(\beta_1, D_1, \beta_2, D_2) = \mathrm{TS}(\beta_1||\mathrm{trailenc}(D_1)||(\beta_2 \oplus \overline{\mathrm{OD}}(\beta_1, D_1)), D_2)\,.$$

We can continue recursively. Let $O(\beta_1) = \beta_1$ and

$$O(\beta_1, D_1, \ldots, \beta_n) = O(\beta_1, D_1, \ldots, \beta_{n-1}) ||\mathrm{trailenc}(D_{n-1})||(\beta_n \oplus \overline{\mathrm{OD}}(\beta_1, \ldots, \beta_{n-1}, D_{n-1}))\,.$$

Then,

$$\overline{\mathrm{OD}}(\beta_1, D_1, \ldots, \beta_n, D_n) = \mathrm{TS}(O(\beta_1, D_1, \ldots, \beta_n), D_n)\,.$$

We can now finish the proof with the recursion on the injectivity of the input mapping to the TurboSHAKE128 input and so by proving that if $(\beta_1, D_1, \ldots, \beta_{n-1}, D_{n-1}) \to (O(\beta_1, D_1, \ldots, \beta_{n-1}), D_{n-1})$ is injective, then $(\beta_1, D_1, \ldots, \beta_n, D_n) \to (O(\beta_1, D_1, \ldots, \beta_n), D_n)$ is injective too. By assumption, any difference in the first $n - 1$ components of the mapping's input necessarily leads to a difference in the mapping's output, so let us consider the case of two inputs that have the same first $n - 1$ components. In this case, the value $\overline{\mathrm{OD}}(\beta_1, D_1, \ldots, \beta_{n-1}, D_{n-1})$ is fixed, and XORing β_n with it preserves the injectivity. □

8.3. Security of keyed OD

We define the security of OD keyed with $\mathcal{K}$ by the advantage of distinguishing it from an idaho scheme with the same ρ. Furthermore, we prove that this advantage is upper bound by the multi-user PRF security of the corresponding sponge function with the same equivalent capacity c.

Theorem 5. *Let F be a (Turbo)SHAKE function and f its underlying permutation. The idaho advantage of* $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$ *keyed with key distribution $\mathcal{K}$ is upper bounded as*

$$\mathrm{Adv}^{\mathrm{idaho}}_{\mathrm{OD}[f,\rho,\mathrm{trailenc}][\mathcal{K}]}(\mu, t, \sigma) \leq \mathrm{Adv}^{\mathrm{mPRF}}_{F[\mathcal{K}]}(\mu, t, \sigma)\,.$$

with resources t, σ and μ defined as in Definition 6.

Proof. Lemma 2 tells us that all the outputs of $\mathrm{OD}[f, \rho, c]$ can be simulated by calls to F with an injective coding. The OD object is keyed with $\mathsf{duplexing}(K[ID], 1)$ just after initialization, and hence Lemma 2 also tells us that $K[\mathrm{ID}]$ is a prefix of F's input. Hence, the subsequent outputs of $\mathrm{OD}[f, \rho, c]$ can be simulated by calls to $F_{K[\mathrm{ID}]}$ instead. We can therefore view $\mathrm{OD}[f, \rho, \mathrm{trailenc}][\mathcal{K}]$ hybridly as an idaho object where the random oracle $\mathcal{RO}$ has been replaced with $F[\mathcal{K}]$, which we will denote as $\textsc{Idaho}[F[\mathcal{K}]]$. The adversary then has to distinguish $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$ from $\textsc{Idaho}[\rho]$, which is not easier than distinguishing $F[\mathcal{K}]$ from $\mathcal{RO}[\mathcal{ID}]$, and this the multi-user PRF security of F. □

8.4. Collision resistance of OD

The following theorem expresses the collision resistance of the OD cipher in terms of the collision resistance of the corresponding sponge function. A collision for an OD cipher means there exists two different sequences of inputs $S = (K, 1, B_1, E_1, \ldots, B_n, E_n)$ and $S' = (K', 1, B'_1, E'_1, \ldots, B'_m, E'_m)$ such that the output of the n-th duplexing call to $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$ with S is the same as the output of the m-th duplexing call to $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$ with S'. Here, the keys K and K' can be chosen by the adversary, and $(K, 1), (K', 1)$ are the input of the duplexing call during the creation of the OD cipher.

Theorem 6. *Let F be a (Turbo)SHAKE function with permutation f and capacity c. If an adversary $\mathcal{D}$ outputs a collision for* $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$*, then one can efficiently transform it into an adversary $\mathcal{D}'$ that outputs a collision for F.*

Proof. Let assume that an adversary $\mathcal{A}$ gives two colliding sequences $S \neq S'$. As shown in Lemma 2, there is an injective mapping from a sequence S to a string X such that the output of $\mathrm{OD}[f, \rho, \mathrm{trailenc}]$ and $F(X)$ are the

same. It follows that an adversary $\mathcal{A}'$ can build two strings X and X' using such injective mapping from S and S', and such strings give a collision in F. □

9. The DWrap mode

In this section, we specify the DWrap mode that builds nonce-based authenticated encryption object from a duplex cipher. We name the concrete schemes by adding "-Wrap" to the underlying (Turbo)SHAKE instance name. We first specify the mode and then discuss the nPJC distinguishing advantage and the CMT-4 committing security of the schemes.

9.1. Specification of DWrap

In Algorithm 4, we specify DWrap. This mode is a refinement of spongeWrap defined in [1] and is illustrated in Figure 3.

DWrap objects make use of an underlying duplex cipher. Upon creation, the underlying duplex cipher is loaded with a secret key $K[ID]$. A wrap call takes as input associated data AD and plaintext P and returns a ciphertext C of $|P| + \tau/8$ bytes. An unwrap call takes as input associated data AD and ciphertext C with $|C| \geq \tau/8$ and returns a plaintext P of $|C| - \tau/8$ bytes or an error $\perp$ in case the ciphertext is invalid. Before unwrapping, the DWrap cipher makes a clone of its duplex cipher, allowing a roll-back in case of an invalid ciphertext. Such clone, denoted by D_{unwrap}, is local to the unwrapping method, i.e., it is deleted when unwrap() returns.

Each ciphertext authenticates all previous messages in the session since initialization. Both AD and P can be empty, leading to four cases. If P is empty, the *ciphertext* is basically a tag of τ bits.

A wrap call first splits the AD and P in sequences of blocks of ρ or less bytes and absorbs them in a number of serial duplexing calls of the underlying duplex cipher, where the trailer is used to indicate the type of block and the purpose of the corresponding duplex output. As a matter of fact, instead of absorbing the blocks of P, it first encrypts the block by adding to it the output of the previous duplexing call and absorbs the resulting ciphertext blocks instead. After absorbing the complete ciphertext, the first $\tau/8$ bytes of duplex output serve as the tag.

If there is no AD in a message, it encrypts the first block of the plaintext by the output of the last duplexing call of the previous wrap call (or the init call). As the first $\tau/8$ bytes of that output were already used for tag generation, this block will be at most $\rho - \tau/8$ bytes long.

9.2. Security of DWrap

Theorem 7. *The nPJC advantage of the* $\textsc{DWrap}[\cdot, \tau]$ *mode (as in Definition 2) is given by*

$$\mathrm{Adv}^{\mathrm{nPJC}}_{\textsc{DWrap}[\cdot,\tau]}(q_{\mathrm{forge}}) \leq \frac{q_{\mathrm{forge}}}{2^{\tau}}$$

with q_{forge} *the number of forgery attempts.*

Proof.

$$\mathrm{Adv}^{\mathrm{nPJC}}_{\textsc{DWrap}[\cdot,\tau]}(q_{\mathrm{forge}}) = \Delta_{q_{\mathrm{forge}}}(\textsc{DWrap}[\textsc{Idaho}[\rho], \tau] \parallel \mathcal{J}^{+}) .$$

Algorithm 4 Def. of $\textsc{DWrap}[\textsc{duplex}, \tau]$.

Parameters:
duplex cipher DUPLEX with ρ
tag length τ, an integer that is a multiple of 8

Object attributes:
D instance of DUPLEX

Constructor: W $\leftarrow$ create($\textsc{DWrap}[\textsc{duplex}, \tau], ID$)
return W with D $\leftarrow$ create(DUPLEX, ID)

Method W $\leftarrow$ DWRAP.clone()
return W: a clone of DWRAP

Method $C \leftarrow$ DWRAP.wrap(AD, P)
$\mathbf{a} \leftarrow \mathrm{parse}(AD, \rho, \rho)$
for $i = 1$ to $|\mathbf{a}| - 1$ **do** D.duplexing($a_i, 5$)
if $|P| > 0$ **then**
  if $|AD| > 0$ **then**
    D.duplexing($a_{|\mathbf{a}|}, 5$)
    $\mathbf{p} \leftarrow \mathrm{parse}(P, \rho, \rho)$
  else
    $\mathbf{p} \leftarrow \mathrm{parse}(P, \rho - t/8, \rho)$
  $x_1 \leftarrow p_1 +$ D.squeezing()
  for $i = 2$ to $|\mathbf{p}|$ **do**
    $x_i \leftarrow p_i +$ D.duplexing($x_{i-1}, 4$)
  $T \leftarrow 0^{\tau} +$ D.duplexing($x_{|\mathbf{p}|}, 6$)
  $X \leftarrow x_1 || \ldots || x_{|\mathbf{p}|}$
  return $C \leftarrow X || T$
else
  $T \leftarrow 0^{\tau} +$ D.duplexing($a_{|\mathbf{a}|}, 7$)
  return $C \leftarrow T$

Method $\{P$ or $\perp\} \leftarrow$ DWRAP.unwrap(AD, C)
if ($|C| < \tau/8$) **then return** $\perp$
$D_{\mathsf{unwrap}} \leftarrow$ D.clone()
$(X || T) \leftarrow C$ with $|T| = \tau/8$
$\mathbf{a} \leftarrow \mathrm{parse}(AD, \rho, \rho)$
for $i = 1$ to $|\mathbf{a}| - 1$ **do** D_{unwrap}.duplexing($a_i, 5$)
if $|X| > 0$ **then**
  if ($|AD| > 0$ **then**
    D_{unwrap}.duplexing($a_{|\mathbf{a}|}, 5$)
    $\mathbf{x} \leftarrow \mathrm{parse}(X, \rho, \rho)$
  else
    $\mathbf{x} \leftarrow \mathrm{parse}(X, \rho - \tau/8, \rho)$
  $p_1 \leftarrow x_1 + D_{\mathsf{unwrap}}$.squeezing()
  for $i = 2$ to $|\mathbf{x}|$ **do**
    $p_i \leftarrow x_i + D_{\mathsf{unwrap}}$.duplexing($x_{i-1}, 4$)
  $T' \leftarrow 0^{\tau} + D_{\mathsf{unwrap}}$.duplexing($x_{|\mathbf{x}|}, 6$)
else
  $T' \leftarrow 0^{\tau} + D_{\mathsf{unwrap}}$.duplexing($a_{|\mathbf{a}|}, 7$)
if $T = T'$ **then**
  D $\leftarrow D_{\mathsf{unwrap}}$
  if $|X| > 0$ **then** $P \leftarrow p_1 || \ldots || p_{|\mathbf{p}|}$ **else** $P \leftarrow \epsilon$
  return P
return $\perp$

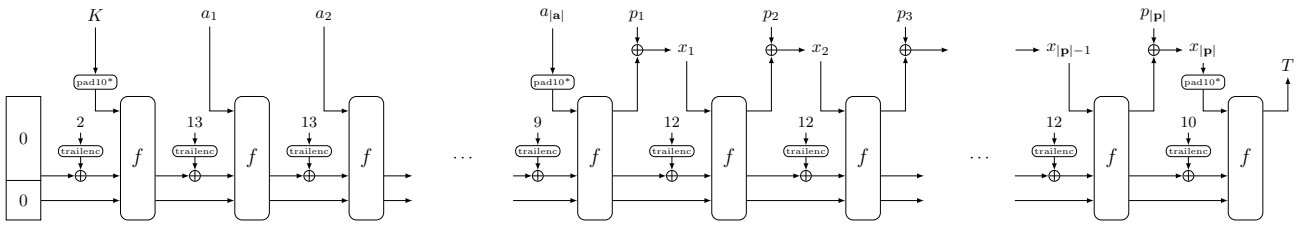

Figure 3. Illustration of the DWrap mode merged with the underlying OD construction. This figure shows a first call create(ID) and then DWRAP.wrap(AD, P). For the sake of representation, we assume that the last block of each input string has length smaller than ρ and therefore that pad10* is applied. Note that trailer values here are the accumulation of domain separation bits from DWrap and OD.

Each call that the mode makes to the underlying IDAHO objects to produce keystream has a different path as the encryption context is a nonce (e.g., the AD of the first wrap call of a session is a nonce). Therefore all tags and keystreams and hence ciphertexts X are uniformly random. The only way to distinguish DWRAP$[\cdot, \tau]$ from the jammin cipher is by a successful forgery: attempting to unwrap a ciphertext that was not generated in a call to wrap. As the tag has τ bits and all tags are uniformly random, the success probability for each attempt is $2^{-\tau}$. After q_{forge} attempts, this is upper bounded by $q_{\text{forge}}/2^{\tau}$. □

9.3. Security of (Turbo)SHAKE-Wrap

The nPJC security of (Turbo)SHAKE-Wrap follows from the nPJC security of the DWrap mode.

Corollary 1. *Let F be a (Turbo)SHAKE instance, with permutation f and capacity c, that per assumption stands by its claimed security and let F-Wrap$[\mathcal{K}]$ be defined as F-Wrap$[\mathcal{K}]$ = DWRAP$[$OD$[f, \rho, \text{trailenc}], \tau][\mathcal{K}]$. Assuming the encryption context is a nonce, the nPJC advantage of F-Wrap is upper bounded as*

$$\mathrm{Adv}^{\mathrm{nPJC}}_{F\text{-Wrap}[\mathcal{K}]}(\mu, t, \sigma, q_{\text{forge}}) \leq \mathrm{Adv}^{\mathrm{mKey}}_{\mathcal{K}}(\mu, t) + \frac{(t+\sigma)^2}{2^{c+1}} + \frac{q_{\text{forge}}}{2^{\tau}},$$

with q_{forge} the number of forgery attempts and μ, t and σ as in Definition 6.

Proof. Applying (3) yields:

$$\mathrm{Adv}^{\mathrm{nPJC}}_{\text{DWRAP}[\text{OD}[f,\rho,\text{trailenc}],\tau][\mathcal{K}]}(\mu, t, \sigma, q_{\text{forge}}) \leq \mathrm{Adv}^{\mathrm{nPJC}}_{\text{DWRAP}[\cdot,\tau]}(q_{\text{forge}}) + \mathrm{Adv}^{\mathrm{idaho}}_{\text{OD}[f,\rho,\text{trailenc}][\mathcal{K}]}(\mu, t, \sigma).$$

Applying Theorem 7 to the first term and Theorem 5 to the second term finishes the proof. □

The committing resistance of (Turbo)SHAKE-Wrap follows from the collision resistance of (Turbo)SHAKE. In (Turbo)SHAKE-Wrap the tag is the result of hashing an injective encoding of the key and all input data received up to that moment. As long as there are no collisions in the tag, the output commits to all inputs. The committing resistance of (Turbo)SHAKE-Wrap is therefore given by the security strength against collisions, namely $\tau/2$ bits, as long as $\tau \leq c$ with c the capacity of the underlying sponge function. Therefore, for tag length $\tau = 256$, the schemes guarantee a committing security strength of 128 bits. For (Turbo)SHAKE256-Wrap, a tag length of $\tau = 512$ bits guarantees committing security strength of 256 bits. If for a given application less committing security strength is considered sufficient, a shorter tag length can be chosen, say $\tau = 160$ for 80 bits of security. This is expressed more formally in the following theorem.

Theorem 8. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. If an adversary $\mathcal{D}$ outputs a tag collision for* DWRAP$[$OD$[f, \rho, \text{trailenc}], \tau]$, *then one can efficiently transform it into an adversary $\mathcal{D}'$ that outputs a collision for F.*

Proof. The tag is the output of the last duplexing call to the underlying OD object after processing the key and the messages. It is therefore sufficient to show that the mapping from a sequence of key and messages $(K, AD_1, P_1, \ldots, AD_n, P_n)$ to a sequence of inputs $(B_1, E_1, \ldots, B_m, E_m)$ to the underlying OD object is injective. The conclusion then follows from Theorem 6.

We start with $n = 1$. We can injectively map the tuple (K, AD, P) to a sequence of the general form $S = (K, 2), (a_1, 13), \ldots, (a_{|\mathbf{a}|}, 9), (c_1, 12), \ldots, (c_{|\mathbf{p}|}, 10)$ such that the tag output by DWRAP$[$OD$[f, \rho, \text{trailenc}], t]$ is equal to the output of OD$[f, \rho, \text{trailenc}]$ after the input sequence S. Here, $\mathbf{a} = \text{parse}(AD, \rho, \rho)$ and $\mathbf{p} = \text{parse}(P, \rho, \rho)$ or $\text{parse}(P, \rho - \tau, \rho)$, while $\mathbf{c}$ is obtained by adding $\mathbf{p}$ bitwise to the keystream.

Let $(K, AD, P) \neq (K', AD', P')$ be mapped to OD sequences S and S', respectively. If $(K, AD) \neq (K', AD')$, then clearly $S \neq S'$ because of the injectivity of AD to $\mathbf{a}$. So, let us assume now that $K = K'$ and $AD = AD'$, but $P \neq P'$. If P and P' have a different number of blocks, then $S \neq S'$. Otherwise, let i be such that $p_j = p'_j$ for all $j < i$ and $p_i \neq p'_i$. Then, the keystream used to encrypt p_i and p'_i is obtained from intermediate duplexing outputs, and it will be identical for p_i and p'_i so that $c_i \neq c'_i$ and therefore $S \neq S'$. This shows that the mapping is injective when $n = 1$.

The reasoning can be easily generalized to $n > 1$, and a simple inspection of Algorithm 4 shows that the value of trailers allows one to unambiguously separate (AD, P) messages in the OD sequence. □

10. The UpperDeck mode

In this section, we define a mode to build a deck cipher on top of a duplex object, called UpperDeck, and discuss its security.

10.1. Specification of UpperDeck

We specify UpperDeck in Algorithm 5 and illustrate it in Figure 4. It has two duplex instances as attributes, that we indicate by D an $\mathrm{D}_{\mathsf{squeeze}}$. Upon creation, it initializes D with the key $K[ID]$. Then, the user can absorb an arbitrarily long string X and trailer E and squeeze as many bits as needed. For this, the UpperDeck cipher splits the string X into blocks $x_1, x_2, \ldots, x_{|\mathbf{x}|}$. The length of each block is ρ except for the last block $x_{|\mathbf{x}|}$ that can be shorter. Then the UpperDeck cipher makes duplexing calls with $E' = \mathrm{padint}(0) = 2$ for $i < |\mathbf{x}|$ and $E' = E||1$ for the last block. When the last block is absorbed, the UpperDeck cipher clones D into $\mathrm{D}_{\mathsf{squeeze}}$ and gets output via $\mathrm{D}_{\mathsf{squeeze}}$. If more than ρ bytes are needed, these are obtained via duplexing calls to $\mathrm{D}_{\mathsf{squeeze}}$ with empty input blocks and trailer $E' = \mathrm{padint}(0) = 2$. Using a cloned duplex cipher for squeezing makes the output of the UpperDeck cipher independent of the output length ℓ of past duplexing or squeezing calls.

Algorithm 6 gives a functionally equivalent description of UpperDeck's duplexing() and squeezing() methods, yet bringing two optimizations. First, it uses D instead of $\mathrm{D}_{\mathsf{squeeze}}$ for the first ρ output bytes, hence avoiding an unnecessary cloning when at most ρ bytes are needed. Second, it defers the calls to $\mathrm{D}_{\mathsf{squeeze}}$.duplexing$(\epsilon, 2)$ until really necessary, hence avoiding extra such calls when the output length is a multiple of ρ bytes.

UpperDeck supports clone() and cloneCompact() where in the latter case it applies compact cloning to its duplex cipher instances.

10.2. Security of UpperDeck

In the following theorem, we express the security of UpperDeck.

Theorem 9. *The idaho advantage of the UpperDeck mode (as in Definition 2) is 0:* $\mathrm{Adv}^{\mathrm{idaho}}_{\textsc{UpperDeck}[\cdot]} = 0$.

Proof.

$$\mathrm{Adv}^{\mathrm{idaho}}_{\textsc{UpperDeck}[\cdot]} = \Delta(\textsc{UpperDeck}[\textsc{Idaho}[\rho]] \parallel \textsc{Idaho}[\infty]) .$$

The UpperDeck cipher converts the input string to each duplex call injectively to a sequence of input blocks and trailers it presents to the underlying idaho object. So the deck mode maps different paths to different paths to its underlying idaho object and equal paths to equal paths. It follows that the two worlds are indistinguishable. □

11. The Deck-BO mode

In this section, we specify the Deck-BO mode, reformulated in terms of a deck cipher that supports the idaho interface and remind its PJC advantage. We specify Deck-BO schemes defined on top of duplex ciphers with security equivalent to that of (Turbo)SHAKE functions that we name by adding "-BO" to the underlying (Turbo)SHAKE instance name. We prove an upper bound on their (plain) PJC advantage of Deck-BO in terms of the multi-user PRF advantage of the equivalent (Turbo)SHAKE function. Finally we prove their committing security.

Algorithm 5 Definition of UpperDeck[duplex]

Parameters:
duplex cipher DUPLEX with DUPLEX.ρ, that we denote as ρ here

Object attributes:
D instance of DUPLEX
$\mathrm{D}_{\mathsf{squeeze}}$ instance of DUPLEX, or $\perp$ if empty
output offset o, an integer

Constructor: U $\leftarrow$ create(UpperDeck[duplex], ID)

return U with D $\leftarrow$ create(DUPLEX, ID), $\mathrm{D}_{\mathsf{squeeze}} \leftarrow \perp$ and $o \leftarrow \infty$

Method U $\leftarrow$ UpperDeck.clone()
return U: a clone of UpperDeck with U.D $\leftarrow$ D.clone() and $\mathrm{U.D}_{\mathsf{squeeze}} \leftarrow \mathrm{D}_{\mathsf{squeeze}}$.clone() (or $\perp$ if $\mathrm{D}_{\mathsf{squeeze}} = \perp$)

Method U $\leftarrow$ UpperDeck.cloneCompact()
return U: a clone of UpperDeck with $\mathrm{U}.o = \infty$, U.D $\leftarrow$ D.cloneCompact() and $\mathrm{U.D}_{\mathsf{squeeze}} \leftarrow \perp$

Method $Z \leftarrow$ UpperDeck.duplexing$(X, E)[\ell]$
$\mathbf{x} \leftarrow \mathrm{parse}(X, \rho, \rho)$
for $i = 1$ to $|\mathbf{x}| - 1$ **do** D.duplexing$(x_i, 2)$
D.duplexing$(x_{|\mathbf{x}|}, E||1)$
$o \leftarrow 0$
$\mathrm{D}_{\mathsf{squeeze}} \leftarrow$ D.clone()
return $Z \leftarrow 0^{8\ell} +$ squeezing()

Method $Z \leftarrow$ UpperDeck.squeezing()$[\ell]$
$Z \leftarrow \epsilon$
while $|Z| < \ell$ **do**
  $x \leftarrow \min(\rho - (o \bmod \rho), \ell - |Z|)$
  $Z \leftarrow Z||(0^{8x} + \mathrm{D}_{\mathsf{squeeze}}.\mathsf{squeezing}())$
  $o \leftarrow o + x$
  if $o \bmod \rho = 0$ **then** $\mathrm{D}_{\mathsf{squeeze}}$.duplexing$(\epsilon, 2)$
return Z

Algorithm 6 Optimization of UpperDeck[duplex]

Method $Z \leftarrow$ UpperDeck.duplexing$(X, E)[\ell]$
[... beginning as in Algorithm 5 ...]
$o \leftarrow 0$
$\mathrm{D}_{\mathsf{squeeze}} \leftarrow \perp$
return $Z \leftarrow 0^{8\ell} +$ squeezing()

Method $Z \leftarrow$ UpperDeck.squeezing()$[\ell]$
$Z \leftarrow \epsilon$
if $o \leq \rho$ **then**
  $x \leftarrow \min(\rho - o, \ell)$
  $Z \leftarrow Z||(0^{8x} +$ D.squeezing()$)$
  if $o + \ell > \rho$ **then**
    $\mathrm{D}_{\mathsf{squeeze}} \leftarrow$ D.cloneCompact()
  $o \leftarrow o + x$
while $|Z| < \ell$ **do**
  if $o \bmod \rho = 0$ **then** $\mathrm{D}_{\mathsf{squeeze}}$.duplexing$(\epsilon, 2)$
  $x \leftarrow \min(\rho - (o \bmod \rho), \ell - |Z|)$
  $Z \leftarrow Z||(0^{8x} + \mathrm{D}_{\mathsf{squeeze}}.\mathsf{squeezing}())$
  $o \leftarrow o + x$
return Z

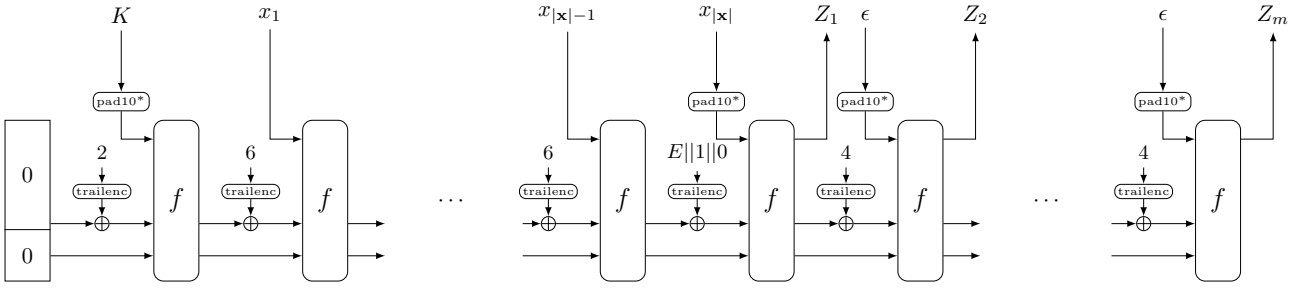

Figure 4. Illustration of the UpperDeck mode merged with the underlying OD cipher. This figure shows the call to create(ID), followed by UPPERDECK.duplexing(X, E) that returns Y. Again, we assume $K[ID]$ and the last block of X have size smaller than ρ.

11.1. Specification of Deck-BO

Deck-BO is the simplest of the four robust modes presented in [14]. It is inspired by the SIV approach in [11] and supports sessions. In Algorithm 7, we rewrite the Deck-BO mode in terms of a deck cipher with the idaho interface (see Definition 9). This is illustrated in Figure 5.

The Deck-BO mode builds on a deck cipher DECK and is parameterized by a tag length τ. It has as attribute a deck cipher instance D. Upon creation, it creates D with the key $K[ID]$. A wrap call takes as input associated data AD (possibly empty) and plaintext P. As output, it gives a ciphertext X, that encrypts P, and an authentication tag T of τ bits. It generates the tag by absorbing $AD; P$ into D and by squeezing the first τ bits from D. The tag T is thus a pseudorandom function of AD and P and is also used as a synthetic diversifier to produce the keystream used to encrypt P. Domain separation bits are used to distinguish between associated data and plaintext, as well as between the generation of tag and keystream. To compute the keystream, it clones its deck cipher instance D to D_{key} with compact cloning after absorbing AD but before P, then absorbs T in it and then generates the keystream Z to encipher the plaintext. This D_{key} cipher is local to the wrap() method, i.e., it is deleted when wrap() returns.

Upon unwrap, it first clones its deck cipher instance D to D_{unwrap} to be able to roll back to the original state in case of failure. Then, to compute the keystream, it in turn clones D_{unwrap} to D_{key} with compact cloning. The ciphers D_{unwrap} and D_{key} are local to the unwrap() method. If unwrap() is successful it updates the state of D with that of D_{unwrap}.

11.2. Security of Deck-BO

Theorem 10. *[14, Theorem 3] The PJC advantage of the* DECKBO$[\cdot,\tau]$ *mode (as in Definition 2) is given by*

$$\mathrm{Adv}^{\mathrm{PJC}}_{\mathrm{DECKBO}[\cdot,\tau]}(q_{\mathrm{forge}}, q_{\mathrm{w}}) \leq \frac{q_{\mathrm{forge}}}{2^{\tau}} + \sum_{\mathrm{context}} \frac{\binom{q_{\mathrm{w}}(\mathrm{context})}{2}}{2^{\tau}},$$

with q_{forge} *the number of forgery attempts,* $q_{\mathrm{w}}(\mathrm{context})$ *the number of wrap queries with* $P \neq \epsilon$ *for a given context and* q_{w} *the list of the* $q_{\mathrm{w}}(\mathrm{context})$ *values indexed by context.*

Algorithm 7 Definition of DECKBO$[$DECK$, \tau]$

Parameters:
deck cipher DECK
tag length τ, an integer divisible by 8

Object attributes:
D instance of DECK

Constructor: create(DECKBO$[$DECK$, \tau]$, ID)
return DB with D $\leftarrow$ create(DECK, ID)

Method DB $\leftarrow$ DECKBO.clone()
return DB: a clone of DECKBO

Method $C \leftarrow$ DECKBO.wrap(AD, P)
if $|P| = 0$ **then**
  return $C \leftarrow 0^{\tau} +$ D.duplexing($AD, 4$)
if $|AD| \neq 0$ **then**
  D.duplexing($AD, 5$)
$D_{\mathsf{key}} \leftarrow$ D.cloneCompact()
$T \leftarrow 0^{\tau} +$ D.duplexing($P, 14$)
$X \leftarrow P + D_{\mathsf{key}}$.duplexing($T, 13$)
return $C \leftarrow X||T$

Method $\{P$ or $\perp\} \leftarrow$ DECKBO.unwrap(AD, C)
if ($|C| < \tau/8$) **then return** $\perp$
$D_{\mathsf{unwrap}} \leftarrow$ D.cloneCompact()
parse C as $X||T$ with $|T| = \tau/8$
if $|X| = 0$ **then**
  $T' \leftarrow 0^{\tau} + D_{\mathsf{unwrap}}$.duplexing($AD, 4$)
  if $T' \neq T$ **then return** $\perp$
  $P \leftarrow \epsilon$
else
  if $|AD| \neq 0$ **then**
    D_{unwrap}.duplexing($AD, 5$)
  $D_{\mathsf{key}} \leftarrow D_{\mathsf{unwrap}}$.cloneCompact()
  $P \leftarrow X + D_{\mathsf{key}}$.duplexing($T, 13$)
  $T' \leftarrow 0^{\tau} + D_{\mathsf{unwrap}}$.duplexing($P, 14$)
  if $T' \neq T$ **then return** $\perp$
D $\leftarrow D_{\mathsf{unwrap}}$
return P

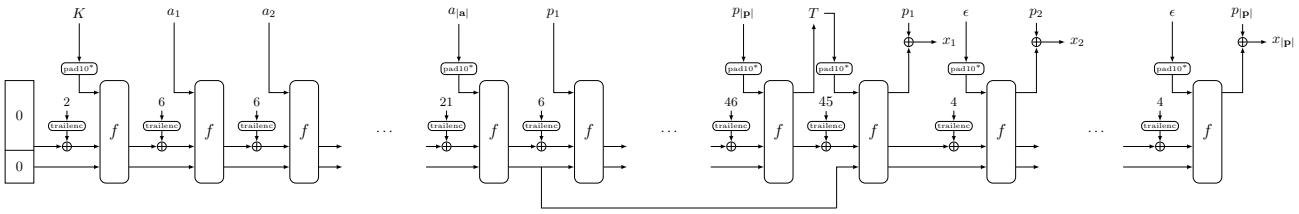

Figure 5. Illustration of the Deck-BO mode merged with the underlying UpperDeck mode and the OD construction. This figure shows the call to create(ID), followed by DECKBO.wrap(AD, P) that returns C. For the sake of visualization, we assume that K, T, and the last blocks of AD and P have length smaller than ρ. Note that trailer values here are the accumulation of domain separation bits from Deck-BO, UpperDeck, and OD.

11.3. PJC security of (Turbo)SHAKE-BO

(Turbo)SHAKE-BO is Deck-BO on top of UpperDeck on top of OD on top of the corresponding KECCAK-$p[]$ permutation with corresponding capacity value. The following corollary gives an upper bound on its PJC advantage (see Definition 7).

Corollary 2. *Let F be a (Turbo)SHAKE instance with capacity c, that per assumption stands by its claimed security and let*

$$F\text{-BO} = \text{DECKBO}[\text{UPPERDECK}[\text{OD}[f, \rho, \text{trailenc}]], \tau]\,.$$

Then the PJC advantage of F-BO$[\mathcal{K}]$ is upper bounded as

$$\text{Adv}^{\text{PJC}}_{F\text{-BO}[\mathcal{K}]}(\mu, t, \sigma, q_{\text{forge}}, q_{\text{w}}) \leq$$
$$\text{Adv}^{\text{mKey}}_{\mathcal{K}}(\mu, t) + \frac{(t+\sigma)^2}{2^{c+1}} + \frac{q_{\text{forge}}}{2^{\tau}} + \sum_{\text{context}} \frac{\binom{q_{\text{w}}(\text{context})}{2}}{2^{\tau}}\,,$$

with q_{forge} and q_{w} as defined in Theorem 10 and μ, t and σ as in Definition 6.

Proof. Applying (3) two times yields:

$$\text{Adv}^{\text{PJC}}_{\text{DECKBO}[\text{UPPERDECK}[\text{OD}[f,\rho,\text{trailenc}]],\tau][\mathcal{K}]}(\mu, t, \sigma, q_{\text{forge}}, q_{\text{w}})$$
$$\leq \text{Adv}^{\text{PJC}}_{\text{DECKBO}[\cdot,\tau]}(q_{\text{forge}}, q_{\text{w}}) + \text{Adv}^{\text{idaho}}_{\text{UPPERDECK}[\cdot]}$$
$$+ \text{Adv}^{\text{idaho}}_{\text{OD}[f,\rho,\text{trailenc}][\mathcal{K}]}(\mu, t, \sigma)$$

Applying Theorem 10 to the first term, Theorem 9 to the second and Theorem 5 to the third term finishes the proof. □

11.4. Committing security of (Turbo)SHAKE-BO

The committing strength of (Turbo)SHAKE-BO is given by the infeasibility of generating tag collisions. By construction, the tag is computed as the hash of all input data via (Turbo)SHAKE. Therefore, the committing security strength is given by the minimum of $c/2$ and $\tau/2$, half the tag length in bits τ. In (Turbo)SHAKE, $c = 256$ or 512, and if we choose $\tau \geq c$ this guarantees a committing security strength of 128 and 256 bits, respectively.

This is expressed more formally in the following theorem, that can be proved following the same approach used to prove Theorem 8.

Theorem 11. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. If an adversary $\mathcal{D}$ outputs a tag collision for (Turbo)SHAKE-BO, then one can efficiently transform it into an adversary $\mathcal{D}'$ that outputs a collision for F.*

12. Performance

In this section, we discuss the performance of the different schemes, $\{\text{TurboSHAKE}, \text{SHAKE}\} \times \{128, 256\} \times \{\text{-Wrap}, \text{-BO}\}$, and compare with ChaCha20-Poly1305, AES128-GCM, AES256-GCM and Ascon-128a.

First off, KECCAK is famous for being very efficient in hardware [36], and this naturally extends to the (Turbo)SHAKE-based schemes. In addition, when protections against side-channel attacks are a concern, its degree-2 round function makes KECCAK particularly well-suited to the masking countermeasures. We therefore focus on comparing our schemes to others in pure software.

With this in mind, we first benchmarked these schemes on a regular PC equipped with an Intel® Core™ i7-7800X CPU running at 3.50 GHz. With our current code, TurboSHAKE128-Wrap runs at 4.23 cycles/bytes and SHAKE256-BO at 18.48 c/b, to take two extreme cases. This has to be compared with ChaCha20-Poly1305 that runs at 1.33 c/b and AES-128-GCM at 0.66 c/b on the same platform. These results may not be completely representative of our schemes and can be explained by two factors. First, our current code is not yet optimized for this platform. An optimized implementation of TurboSHAKE128 XOF runs at 2.20 c/b on this platform and, accounting for the difference in block sizes (see also Table 3), we expect TurboSHAKE128-Wrap to reach about 2.31 c/b after optimization. Second, and more importantly, the AES-GCM implementation makes use of a dedicated cryptographic acceleration, and this falls short of a comparison in pure software.

We next show the performance on an Raspberry Pi 4 equipped with an ARM™ Cortex-A72 processor running at 1.5 GHz. This is a popular platform, yet one that does not have any dedicated cryptographic acceleration, to be able to compare relevant algorithms on a more equal footing. We implemented the algorithms on top of the code provided in XKCP [37]. Table 2 gives the cost, in nanosecond per byte, of wrapping, unwrapping and processing the associated data for the different schemes. We focus on the cost for long messages, i.e., the slope for increasing sizes of associated data, plaintext or ciphertext. Table 2 also gives the cost of ChaCha20-Poly1305, AES128-GCM and AES256-GCM on the same platform using the implementation in OpenSSL 3.0.2 [38].

On this platform, we can see that all the schemes defined in this paper outperform AES-based ones. ChaCha20-Poly1305 is a particularly efficient alternative that does not rely on hardware acceleration. Yet, TurboSHAKE128-Wrap outperforms it. We were not able

TABLE 2. PERFORMANCE ON RASPBERRY PI 4 (NS/BYTE).

	...-Wrap	...-BO	
		AD	P or C
TurboSHAKE128	3.33	3.04	6.23
TurboSHAKE256	4.06	3.84	7.82
SHAKE128	6.41	6.27	12.58
SHAKE256	8.07	7.80	15.72
ChaCha20-Poly1305	3.72		
AES128-GCM	32.32		
AES256-GCM	41.69		

TABLE 3. PERFORMANCE RELATIVE TO SHAKE128.

	...-Wrap	...-BO	
		AD	P or C
TurboSHAKE128	0.525	0.525	1.050
TurboSHAKE256	0.656	0.656	1.313
SHAKE128	1.050	1.050	2.100
SHAKE256	1.313	1.313	2.625

to benchmark Ascon ourselves due to the fact that the standard is not finalized yet. Still, the Ascon website[1] gives a good idea of its performance: On Cortex-A72, the fastest variant of Ascon, Ascon-128a, encrypts at 7.0 cycles/byte or 4.6 ns/byte at 1.5 GHz. This is faster than both SHAKE variants but slower than the TurboSHAKE ones.

Lastly, we discuss the cost relative to that of hashing with the standard function SHAKE128. Table 3 evaluates the cost of the different schemes under the assumption that the evaluation of the KECCAK-p permutation dominates.

Let us first discuss the relative cost of the OD layer. SHAKE128 processes input and output blocks of 168 bytes per call to the permutation, whereas $\rho = 160$ bytes and $\rho = 128$ bytes for OD on top of (Turbo)SHAKE128 and (Turbo)SHAKE256, respectively. Due to OD's smaller payload block length, this induces a relative cost of $168/160 = 1.05$ for OD on top of SHAKE128 and of $168/128 = 1.3125$ with SHAKE256. Due to their lower number of rounds, the "Turbo" variants benefit from a factor-2 speed-up.

Next, we discuss the relative cost of (Turbo)SHAKE-Wrap. This mode requires only one pass of the associated data, plaintext or ciphertext. Thanks to the duplexing, producing keystream blocks does not induce any extra costs. Associated data, plaintext or ciphertext blocks translate directly to OD's payload blocks, so the long-message performance of (Turbo)SHAKE-Wrap is the same as that of the OD layer.

Finally, we discuss the relative cost of Deck-BO. This mode needs one pass of the deck function to process the associated data. Here again, associated data blocks from Deck-BO translate directly to OD's payload blocks. However, it needs two passes to process the plaintext or ciphertext, so the cost per plaintext or ciphertext byte is twice that of the underlying OD.

Table 2 is consistent with Table 3 as the cost of evaluating SHAKE128 on a Raspberry Pi 4 is about 6.11 ns/byte. There is a small discrepancy between the processing of plaintext and ciphertext in Wrap and that of the associated data in BO, e.g., 3.33 vs 3.04 for TurboSHAKE128-Wrap. Processing associated data is faster because there is no keystream to add with the plaintext or with the ciphertext.

1. https://ascon.iaik.tugraz.at/implementations.html

13. Conclusions

In this work we introduce session-supporting authenticated encryption schemes with inherent committing properties. The committing security of our schemes is based the fact that the tags are a hash of all inputs. Specifically, they are based on SHAKE and TurboSHAKE, whose collision resistance properties guarantee committing security in a natural way. Besides committing security, our proposed schemes are user-friendly in the sense that they do not restrict the size of the input that needs to be a nonce, they support sessions, which relaxes the need for nonce management in some cases, and generally they have strong indistinguishably properties based on the security claim in the SHA-3 standard.

Our schemes have also some implementation advantages. They require a single primitive in contrast to other committing solutions which usually require two. The underlying permutation is standard and there is an increasing hardware support for it. Yet, even without hardware acceleration, our schemes have competitive performance. Also, the definition of the overwrite duplex cipher allows smaller state footprint during clone functions, i.e., 40 bytes instead of 200 for (Turbo)SHAKE128 and 72 instead of 200 for (Turbo)SHAKE256.

Acknowledgments

Silvia Mella and Joan Daemen were funded by the Dutch Research Council (NWO) through the PROACT project (NWA.1215.18.014) and the CiCS project of the research programme Gravitation under the grant 024.006.037.

References

[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Duplexing the sponge: Single-pass authenticated encryption and other applications," in *Selected Areas in Cryptography - SAC 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Miri and S. Vaudenay, Eds., vol. 7118. Springer, 2011, pp. 320–337. [Online]. Available: https://doi.org/10.1007/978-3-642-28496-0_19

[2] H. Krawczyk, "The order of encryption and authentication for protecting communications (or: How secure is ssl?)," in *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, ser. Lecture Notes in Computer Science, J. Kilian, Ed., vol. 2139. Springer, 2001, pp. 310–331. [Online]. Available: https://doi.org/10.1007/3-540-44647-8_19

[3] NIST, "NIST special publication 800-38D, recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC," November 2007.

[4] ——, "NIST special publication 800-38C, recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality," July 2007.

[5] Y. Nir and A. Langley, "Chacha20 and poly1305 for IETF protocols," *RFC*, vol. 8439, pp. 1–46, 2018. [Online]. Available: https://doi.org/10.17487/RFC8439

[6] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," *J. Cryptol.*, vol. 34, no. 3, p. 33, 2021. [Online]. Available: https://doi.org/10.1007/s00145-021-09398-9

[7] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, "OCB: a block-cipher mode of operation for efficient authenticated encryption," in *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001*, M. K. Reiter and P. Samarati, Eds. ACM, 2001, pp. 196–205.

[8] M. N. Wegman and L. Carter, "New hash functions and their use in authentication and set equality," *J. Comput. Syst. Sci.*, vol. 22, no. 3, pp. 265–279, 1981. [Online]. Available: https://doi.org/10.1016/0022-0000(81)90033-7

[9] J. Daemen, P. M. C. Massolino, A. Mehrdad, and Y. Rotella, "The subterranean 2.0 cipher suite," *IACR Trans. Symmetric Cryptol.*, vol. 2020, no. S1, pp. 262–294, 2020. [Online]. Available: https://doi.org/10.13154/tosc.v2020.iS1.262-294

[10] F. Liu, T. Isobe, and W. Meier, "Cube-based cryptanalysis of subterranean-sae," *IACR Trans. Symmetric Cryptol.*, vol. 2019, no. 4, pp. 192–222, 2019. [Online]. Available: https://doi.org/10.13154/tosc.v2019.i4.192-222

[11] P. Rogaway and T. Shrimpton, "A provable-security treatment of the key-wrap problem," in *Advances in Cryptology - EUROCRYPT 2006, Proceedings*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 4004. Springer, 2006, pp. 373–390. [Online]. Available: https://doi.org/10.1007/11761679_23

[12] S. Gueron and Y. Lindell, "GCM-SIV: full nonce misuse-resistant authenticated encryption at under one cycle per byte," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 109–119. [Online]. Available: https://doi.org/10.1145/2810103.2813613

[13] S. Gueron, A. Langley, and Y. Lindell, "AES-GCM-SIV: nonce misuse-resistant authenticated encryption," *RFC*, vol. 8452, pp. 1–42, 2019. [Online]. Available: https://doi.org/10.17487/RFC8452

[14] N. Băcuieți, J. Daemen, S. Hoffert, G. V. Assche, and R. V. Keer, "Jammin' on the deck," in *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Agrawal and D. Lin, Eds., vol. 13792. Springer, 2022, pp. 555–584. [Online]. Available: https://doi.org/10.1007/978-3-031-22966-4_19

[15] NIST, "Federal information processing standard 197, advanced encryption standard (AES)," November 2001.

[16] C. Dobraunig, K. Matusiewicz, B. Mennink, and A. Tereschenko, "Efficient instances of docked double decker with AES," *IACR Cryptol. ePrint Arch.*, p. 84, 2024. [Online]. Available: https://eprint.iacr.org/2024/084

[17] P. Grubbs, J. Lu, and T. Ristenpart, "Message franking via committing authenticated encryption," in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, ser. Lecture Notes in Computer Science, J. Katz and H. Shacham, Eds., vol. 10403. Springer, 2017, pp. 66–97. [Online]. Available: https://doi.org/10.1007/978-3-319-63697-9_3

[18] M. Bellare and V. T. Hoang, "Efficient schemes for committing authenticated encryption," in *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, O. Dunkelman and S. Dziembowski, Eds., vol. 13276. Springer, 2022, pp. 845–875. [Online]. Available: https://doi.org/10.1007/978-3-031-07085-3_29

[19] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, "The design of Xoodoo and Xoofff," *IACR Trans. Symmetric Cryptol.*, vol. 2018, no. 4, pp. 1–38, 2018. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/7359

[20] 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.

[21] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, R. V. Keer, and B. Viguier, "TurboSHAKE," *IACR Cryptol. ePrint Arch.*, p. 342, 2023. [Online]. Available: https://eprint.iacr.org/2023/342

[22] NIST, "Federal information processing standard 203, module-lattice-based key-encapsulation mechanism standard," August 2024, https://doi.org/10.6028/NIST.FIPS.203.

[23] ——, "Federal information processing standard 204, module-lattice-based digital signature standard," August 2024, https://doi.org/10.6028/NIST.FIPS.204.

[24] A. Albertini, T. Duong, S. Gueron, S. Kölbl, A. Luykx, and S. Schmieg, "How to abuse and fix authenticated encryption without key commitment," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 3291–3308. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/albertini

[25] Y. Naito, Y. Sasaki, and T. Sugawara, "Committing security of ascon: Cryptanalysis on primitive and proof on mode," *IACR Trans. Symmetric Cryptol.*, vol. 2023, no. 4, pp. 420–451, 2023. [Online]. Available: https://doi.org/10.46586/tosc.v2023.i4.420-451

[26] J. Baudrin, A. Canteaut, and L. Perrin, "Practical cube attack against nonce-misused ascon," *IACR Trans. Symmetric Cryptol.*, vol. 2022, no. 4, pp. 120–144, 2022. [Online]. Available: https://doi.org/10.46586/tosc.v2022.i4.120-144

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

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

[29] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer, and B. Viguier, "KangarooTwelve: Fast hashing based on Keccak-p," in *Applied Cryptography and Network Security, ACNS 2018, Proceedings*, ser. Lecture Notes in Computer Science, B. Preneel and F. Vercauteren, Eds., vol. 10892. Springer, 2018, pp. 400–418. [Online]. Available: https://doi.org/10.1007/978-3-319-93387-0_21

[30] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 4004. Springer, 2006, pp. 409–426. [Online]. Available: https://doi.org/10.1007/11761679_25

[31] J. Daemen, B. Mennink, and G. Van Assche, "Full-state keyed duplex with built-in multi-user support," in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Takagi and T. Peyrin, Eds., vol. 10625. Springer, 2017, pp. 606–637. [Online]. Available: https://doi.org/10.1007/978-3-319-70697-9_21

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

[33] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, "Farfalle: parallel permutation-based cryptography," *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 4, pp. 1–38, 2017. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/855

[34] V. T. Hoang, R. Reyhanitabar, P. Rogaway, and D. Vizár, "Online authenticated-encryption and its nonce-reuse misuse-resistance," in *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Gennaro and M. Robshaw, Eds., vol. 9215. Springer, 2015, pp. 493–517. [Online]. Available: https://doi.org/10.1007/978-3-662-47989-6_24

[35] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Duplexing the sponge: single-pass authenticated encryption and other applications," in *Selected Areas in Cryptography (SAC)*, 2011.

[36] P. Yalla, E. Homsirikamol, and J. Kaps, "Comparison of multi-purpose cores of keccak and AES," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, W. Nebel and D. Atienza, Eds. ACM, 2015, pp. 585–588. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755885

[37] G. Van Assche, R. Van Keer, and Contributors, "Extended KECCAK code package," January 2024, https://github.com/XKCP/XKCP.

[38] OpenSSL community, "OpenSSL – cryptography and SSL/TLS toolkit," https://github.com/openssl/openssl.

[39] Y. Dodis, P. Grubbs, T. Ristenpart, and J. Woodage, "Fast message franking: From invisible salamanders to encryptment," in *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, 2018, pp. 155–186. [Online]. Available: https://doi.org/10.1007/978-3-319-96884-1_6

[40] J. Albrecht, "Introducing subscribe with Google," https://blog.google/outreach-initiatives/google-news-initiative/introducing-subscribe-google/.

[41] J. Salowey, A. Choudhury, and D. A. McGrew, "AES galois counter mode (GCM) cipher suites for TLS," *RFC*, vol. 5288, pp. 1–8, 2008. [Online]. Available: https://doi.org/10.17487/RFC5288

[42] T. Krovetz and P. Rogaway, "The software performance of authenticated-encryption modes," in *Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Joux, Ed., vol. 6733. Springer, 2011, pp. 306–327. [Online]. Available: https://doi.org/10.1007/978-3-642-21702-9_18

[43] J. Chan and P. Rogaway, "On committing authenticated-encryption," in *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, V. Atluri, R. D. Pietro, C. D. Jensen, and W. Meng, Eds., vol. 13555. Springer, 2022, pp. 275–294. [Online]. Available: https://doi.org/10.1007/978-3-031-17146-8_14

[44] J. Len, P. Grubbs, and T. Ristenpart, "Partitioning oracle attacks," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 195–212. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/len

[45] S. Jarecki, H. Krawczyk, and J. Xu, "OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks," in *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, ser. Lecture Notes in Computer Science, J. B. Nielsen and V. Rijmen, Eds., vol. 10822. Springer, 2018, pp. 456–486. [Online]. Available: https://doi.org/10.1007/978-3-319-78372-7_15

[46] P. Farshim, C. Orlandi, and R. Rosie, "Security of symmetric primitives under incorrect usage of keys," *IACR Trans. Symmetric Cryptol.*, vol. 2017, no. 1, pp. 449–473, 2017. [Online]. Available: https://doi.org/10.13154/tosc.v2017.i1.449-473

[47] H. Krawczyk, "The opaque asymmetric pake protocol," Internet-Draft draft-krawczyk-cfrgopaque-03, Internet Engineering Task Force, 2019.

Appendix A. Data availability

The source code of our schemes will be available at the XKCP Github repository soon. This will include both the reference implementation and the optimized implementation benchmarked in Section 12.

Appendix B. The jammin cipher, an ideal-world AE scheme

Algorithm 8 The jammin cipher $\mathcal{J}^{\mathrm{WrapExpand}(p)}$

Parameters
WrapExpand, a τ-expanding function

Global variables
codebook initially set to $\perp$ for all
taboo initially set to *empty*

Object attributes:
history: a sequence of strings

Constructor: inst $\leftarrow$ create($\mathcal{J}^{\mathrm{WrapExpand}(p)}, ID$)
return inst of jammin with attribute inst.history $= ID$

Method inst $\leftarrow$ jammin.clone()
return inst: a clone of jammin

Method $C \leftarrow$ jammin.wrap(AD, P)
context $\leftarrow$ history; AD
if codebook(context; P) $= \perp$ **then**
  $\mathcal{C} = \mathbb{Z}_2^{\mathrm{WrapExpand}(|P|)} \setminus$ (codebook(context; $*$) $\cup$ taboo(context))
  if $\mathcal{C} = \emptyset$ **then return** $\perp$
  codebook(context; P) $\xleftarrow{\$} \mathcal{C}$
history $\leftarrow$ history; AD; P
return codebook(context; P)

Method P or$\perp \leftarrow$ jammin.unwrap(AD, C)
context $\leftarrow$ history; AD
if $\exists! P$: codebook(context; P) $= C$ **then**
  history $\leftarrow$ history; AD; P
  return P
else
  taboo(context) $\leftarrow C$
  return $\perp$

In Algorithm 8, we recall the definition of the *jammin cipher* [14]. We describe it in an object-oriented way, with *object instances* (or *instances* for short) held by the communicating parties. An instance belongs to a given party who initializes it with an object identifier ID. Such an identifier is the counterpart of a secret key in the real world: Encryption and decryption will work consistently only between instances initialized with the same identifier. This setup models independent pairs (or groups) that make use of the AE scheme simultaneously. For example, Alice and Bob may secure their communication each using coupled instances that share the same identifier $ID_{\text{Alice and Bob}}$, while Edward and Emma use coupled instances initialized with $ID_{\text{Edward and Emma}}$.

The jammin object supports both wrap() and unwrap(). With the wrap() operation the object computes a ciphertext C from a message that has a plaintext P and associated data AD, both arbitrary bit strings. With the unwrap() operation the object computes the plaintext P

from the ciphertext C and AD again. The ciphertext C is the encryption of P for a given AD.

The jammin cipher is parameterized with a function $\mathrm{WrapExpand}(p)$ that specifies the length of the ciphertext given the length p of the plaintext. Typical examples observed in AE schemes in the literature are $\mathrm{WrapExpand}(p) = p + \tau$ with τ some fixed length, e.g., 128 for stream encryption followed by a 128-bit tag. For use with the jammin cipher, we require $\mathrm{WrapExpand}$ to satisfy this property, defined below.

Definition 11. *A function* $f\colon \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ *is* τ-expanding *iff (i)* $\forall \ell > 0\colon f(\ell) > f(0)$ *and (ii)* $\forall \ell\colon f(\ell) \geq \ell + \tau$.

When two parties communicate, they usually have more than one message to send to each other. And a message is often a response to a previous request, or in general its meaning is to be understood in the context of the previous messages. The jammin cipher is *stateful*, where the sequence of messages exchanged so far is tracked in the attribute history. Initialization sets this attribute to the object identifier and each wrap() and (successful) unwrap() appends a message (AD, P). So history is a sequence with ID followed by zero, one or more messages (AD, P).

A *session* is the process in which the history grows with the messages exchanged so far. The wrap() and unwrap() operations make the history act as associated data, so that a ciphertext authenticates not only the message (AD, P) but also the sequence of messages exchanged so far. An important application of this are intermediate tags, which authenticate a long message in an incremental way.

Finally, a jammin cipher object can be cloned. This is the ideal world's equivalent of making a copy of the state of the cipher. This means the user can save the history and restart from it ad libitum.

B.1. Properties

The jammin cipher enjoys the following properties:

Deterministic wrapping: In a given context, an object wraps equal messages (AD, P) to equal ciphertexts C. It achieves this by tracking the ciphertexts in the codebook archive.

Injective wrapping: An object wraps messages with equal context and AD and different P to different ciphertexts. It achieves this by excluding ciphertext values that it returned in earlier wrap calls for the same context and AD.

Random ciphertexts: Except for determinism and injectivity, all ciphertexts C are fully random.

Deterministic unwrapping: In a given context, an object unwraps equal ciphertexts to equal responses. It achieves this by tracking in taboo ciphertext values that it returns an error to.

Correctness: Thanks to deterministic (un)wrapping and injective wrapping, one jammin cipher object correctly unwraps what another wrapped, whenever their contexts are equal.

Forgery-freeness: In a given context, an object will only unwrap successfully ciphertexts C resulting from prior wrap calls in the same context.

The jammin cipher does not enforce the encryption context to be a nonce, this is left up to the higher level protocol or use case.

The jammin cipher takes as encryption context the sequence of messages exchanged so far, including the associated data in the message containing the plaintext to be encrypted (in a message without plaintext, there is no encryption and hence no encryption context). The advantage of doing authenticated encryption in sessions is immediate as this reduces the requirement for global diversifiers of one per session rather than one per message. Session-level diversifiers may even be omitted unless communicating parties wish to start parallel threads or start afresh from the same shared key.

Definition 12. *We say that the* encryption context is a nonce *iff all wrap queries with non-empty plaintext have a different context* context.

In case of re-use of encryption context, the jammin cipher will leak equality of plaintexts given equal ciphertexts obtained with equal encryption contexts, but nothing more. In some use cases this may be acceptable. For such use cases, the jammin cipher can serve as a security reference for modes or schemes. A proven upper bound on the distinguishing advantage between such a mode and the jammin cipher, proves that leakage is limited to equal plaintexts and encryption contexts, plus the proven advantage that is typically negligible.

In particular, stream encryption with a keystream that is generated from the encryption context is perfectly secure if each wrap query has a different encryption context, but its security completely breaks down when re-using encryption contexts. Therefore, if we wish security in case of repeating encryption contexts, we must use a more elaborate encryption mechanism than stream encryption.

Appendix C. Committing AE

Certain settings or applications require AE with committing property, as shown in the following examples. Dodis et al. [39] and Grubbs et al. [17] showed how to exploit non-committing AE schemes in old versions of Facebook's end-to-end encrypted message service. In [24], Albertini et al. study weaknesses of key rotation in key management services, envelope encryption, and "Subscribe with Google" [40], due to the lack of key commitment. They first introduce new theoretical attacks against commonly used AE schemes, such as AES-GCM [41], AES-GCM-SIV [12], [13], ChaCha20-Poly1305 [5], and AES-OCB3 [42], which they turn into practical ones by creating *binary polyglots* (i.e., files which are valid in two different file formats). In [43], Chan and Rogaway show how in GCM and OCB modes, for any ciphertext C generated under a "honest" key, the adversary can compute an AD that together with C results in a successful unwrap under another known key. In [44], Len et al. show how Shadowsocks proxy servers and the OPAQUE [45] protocol can be vulnerable to partitioning oracle attacks due to using non-committing AE.

Farshim et al. ported the notion of key-commitment to the AE setting in 2017, with the name *key-robustness* [46].

Later, different definitions have been introduced. Bellare and Hoang [18] and Chan and Rogaway [43] independently and contemporarily gave a number of committing AE definitions, the strongest requiring that the ciphertext commits to key, nonce, associated data, and plaintext.

Generic solutions have been presented to turn existing AE schemes into committing AE schemes. Farshim et al. [46] propose to apply a collision-resistant pseudorandom function (PRF) (not to be confused with the PRF security notion) to the entire message or ciphertext, to achieve key commitment. Grubbs et al. [17] presented *compactly committing AE*, requiring a collision-resistant hash function in HMAC mode and a stream cipher such as AES-CTR or ChaCha20. In [24], Albertini et al. achieve key commitment by deriving a new encryption key and a commitment string from the scheme's key, by using a collision resistant hash function like SHA256. Chan and Rogaway [43] propose a generic construction that makes a nonce-based AE scheme committing in the strongest sense, at the cost of a hash call over the tag. Bellare and Hoang [18] introduce two generic constructions. The former makes use of a committing PRF, which is a generalization of a key-robust PRF based on a block cipher. This construction however does not guarantee resistance against nonce-misuse. The latter construction preserves misuse-resistance and makes use of the same key-robust PRF and a collision resistant PRF. Dodis et al. [39] design *encryptment schemes* as a building block to achieve compact committing AE. They give a concrete encryptment scheme that uses a compression function and a padding scheme. In the appendix of their work, the authors also discuss a SpongeWrap-like encryptment scheme, but without discussing the details. None of these generic solutions achieves the efficiency of AES-GCM, and the majority of them requires two passes and the use of more than one primitive.

Alternative solutions exist that aim to achieve commitment for specific schemes. One of such solutions consists in adding a padding block to the plaintext and verify the correctness of the key by checking the presence of such padding block upon decryption [24], [47]. However the commitment security of such padding solution is not guaranteed for every AE scheme, but must be verified on a case-by-case basis, which was done for AES-GCM, ChaCha20-Poly1305 [24] and Ascon [25]. In [18], Bellare and Hoang also propose modifications to the GCM and GCM-SIV modes to make them key-committing. With the addition of the generic transformation cited above, they become committing in the strongest sense. However, these solutions are intrusive, as they require modifications to GCM and GCM-SIV.