Committing authenticated encryption based on SHAKE

Joan Daemen[1], Silvia Mella[1] and Gilles Van Assche[2]

[1] Radboud University, Nijmegen, The Netherlands, first.last@ru.nl
[2] STMicroelectronics, Diegem, Belgium, first.lastname@st.com

Abstract. Authenticated encryption is a cryptographic mechanism that allows communicating parties to protect the confidentiality and integrity of message exchanged over a public channel, provided they share a secret key. Some applications require committing authenticated encryption schemes, a security notion that is not covered by the classical requirements of confidentiality and integrity given a secret key. An authenticated encryption (AE) scheme is *committing* in the strongest sense when it is impossible to generate the same ciphertext for different $(K, [N,]A, P)$ tuples, with K the key, N the nonce, A the associated data and P the plaintext.
In this work, we present authenticated encryption schemes for which we provably reduce their confidentiality, integrity and commitment security to the security of an underlying sponge function. In particular, we instantiate them with SHAKE128 and SHAKE256, offering 128 and 256 bits of security strength and based on the security claim in the SHA-3 standard FIPS 202. Cryptanalysis of reduced-round versions of SHA-3 and SHAKE functions suggests that the number of rounds can be divided by two without noticeable security degeneration, and this had lead to the definition of TurboSHAKE128 and TurboSHAKE256; hence we also instantiate our scheme with these functions, offering the same security strength at twice the speed. The AE schemes we propose therefore have the unique advantages that 1) their security is based on a security claim that has received a large amount of public scrutiny and that 2) it makes use of the standard Keccak-p permutation that has dedicated hardware support on more and more CPUs.
In more details, we build two online AE modes on top of a sponge function, in multiple layers. At the bottom layer, we use a variant of the duplex construction, referred to as *overwrite duplex* or *OD* for short, that uses an overwrite operation leading to a smaller state footprint. Our first AE mode is nonce-based and built using a variant of the SpongeWrap mode on top of OD, and security-equivalent to it. Our second AE mode makes use of the Deck-BO mode published at Asiacrypt 2022, an online version of a Synthetic Initial Value (SIV) authenticated encryption scheme. It requires a deck function that we build on top of the OD, again security-equivalent to it.

Keywords: Committing authenticated encryption · sponge/duplex · construction · SHA-3 · TurboSHAKE

1 Introduction

Authenticated encryption (AE) is a cryptographic mechanism that provides both confidentiality and integrity under a secret key. When a sender and a receiver hold such a secret key K that was not shared with anyone else, then the successful decryption 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 definition and all bets are off. In general, AE does

not ensure the integrity of the key, and one ciphertext C could succesfully decrypt under two (or more) different keys.

It has been proved that there are widely used AE methods, like AES-GCM and ChaCha20Poly1305, where one can successfully decrypt a ciphertext to different plaintexts using different keys [GLR17]. Schemes that are susceptible to this are said to *not commit to the key*. On the contrary, the property of key-commitment guarantees that a ciphertext can only be decrypted using the same key that was used to create it. This can be extended with the general name of *committing AE* to also include the inability to generate colliding ciphertexts for different $(K, [N,]A, P)$ tuples, with K the key, N the nonce, A the associated data and P the plaintext.

Certain settings or applications require the committing property, as shown in the following examples. Dodis et al. [DGRW18] and Grubbs et al. [GLR17] showed how to exploit AE schemes which do not commit to the key in the context of abuse reporting in Facebook's end-to-end encrypted message system. When encrypting attachments in Facebook's message franking protocol, it is in fact possible to send abusive images that cannot be reported. In [ADG+22], Albertini et al. study weaknesses of key rotation in key management services, envelope encryption, and "Subscribe with Google" [Alb], due to the lack of key commitment. In these contexts, they introduce new attacks against standardized AE schemes, such as AES-GCM-SIV and OCB, which they turn into practical ones by creating *binary polyglots* (i.e., files which are valid in two different file formats). In [CR22], Chan and Rogaway present a new attack on GCM and OCB where, for any ciphertext C generated under a "honest" key, the adversary can provide a legitimate decryption under another known key. In [LGR21], Len et al. built a practical partitioning oracle attack that recovers passwords from Shadowsocks proxy servers by exploiting the lack of key commitment. They also discuss how some early implementations of the OPAQUE protocol, for password-based key exchange, could be vulnerable to partitioning oracle attacks due to using non-committing AEAD.

The notion of committing encryption was introduced in 2003 by Gertner and Herzberg [GH03], who studied the problem in both the symmetric and asymmetric settings, but did not consider deterministic or authenticated encryption. In 2010, Abdalla et al. [ABN10] introduce the term *robustness* to denote the difficulty of producing a ciphertext valid under two different encryption keys. Their work covers public-key and identity-based encryption settings with honestly generated keys. Their robustness notion was then strengthened by Farshim et al. [FLPQ13] to include robustness against adversarially-chosen keys. Farshim et al. ported the notion of robustness to the AE setting in 2017, with the name *key-robustness* [FOR17]. Later, Grubbs et al. [GLR17] and Dodis et al. in [DGRW18] defined variants of committing AE schemes to support *message franking*, i.e., verifiable abuse reporting in end-to-end encrypted message systems like Facebook's Messenger. Grubbs et al. [GLR17] consider committing to the header and message. Bellare and Hoang [BH22] and Chan and Rogaway [CR22] independently and contemporarly gave a number of committing AE definitions, the strongest requiring that the ciphertext commits to key, nonce, AD, and plaintext. Bellare and Hoang also consider *multi-input committing* security, where more than two input tuples that give the same ciphertext are requested to the attacker. More recently, Menda et al. [MLGR23] introduced a new framework that allows to define commitment security with a better granularity in terms of what the adversary can control. In fact, in real settings, the attacker will not have full control over the input and will have to respect application-specific constraints. For instance, in the attack against Facebook's message franking protocol of [DGRW18], the attacker has to build a ciphertext that decrypts under two input tuples with equivalent nonces. In the key rotation attack of Albertini et al. [ADG+22], the keys must be previously imported in the key management service and therefore cannot be freely chosen by the attacker.

Generic solutions have been presented to turn existing AE schemes into committing

AE schemes. Some examples are the following. Farshim et al. [FOR17] propose a generic composition which applies a collision-resistant pseudorandom function (PRF) to the entire message or ciphertext, to achieve key-committing. As a practical instantiation, they propose using a hash function with a key, for example HMAC or KMAC. Grubbs et al. [GLR17] presented *compactly committing AE*, where a small portion of the ciphertext commits to the message. This requires computing HMAC and AES-CTR mode over the message. In [ADG+22], Albertini et al. propose a construction to transform any AE scheme into a key-committing one. The solution consists in deriving a new encryption key and a commitment string from the scheme's key, by using a collision resistant hash function like SHA256. An instantiation of such generic composition is deployed as part of the AWS Encryption SDK [Ama], an open source client-side encryption library [Tri]. Chan and Rogway [CR22] also propose a generic construction that makes a nonce-based AE scheme committing in the strongest sense. The additional computational cost is a hash call over the tag. Bellare and Hoang [BH22] 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. [DGRW18] 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 full 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 [ADG+22, Kra19]. However the effectiveness 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 and ChaCha20Poly1305 [ADG+22]. Moreover, Menda et al. [MLGR23] show that this padding zeros transform, while designed to achieve key-commitment, does not achieve commitment in the strongest sense. In [BH22], Bellare and Hoang also propose modifications to GCM and AES-GCM-SIV 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 AES-GCM-SIV.

1.1 Our contribution

In this work we propose two AE schemes based on the sponge functions SHAKE and TurboSHAKE, whose collision resistance guarantees commitment security. Our schemes have two unique advantages. The first is that their security is based on the security claim of a NIST standard that has received a large amount of public scrutiny: the SHA-3 standard FIPS 202 [NIS15]. The second is that they make use only of the standard Keccak-p permutation that has dedicated hardware support on more and more CPUs (e.g., the recent Apple™ processors).

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

- At the bottom is the hashing layer that we instantiate with SHAKE and TurboSHAKE.
- Then, we define a duplexing interface for SHAKE and TurboSHAKE that we call *overwrite duplex* or OD. It is a duplex object that provides incremental hashing. Like the Overwrite mode defined in [BDPV11b], the given input block overwrites part of the state instead of being XORed into it. This is more efficient when the state

needs to be cloned between calls, since the part of the state being overwritten does not need to be copied. Concretely, in the case of (Turbo)SHAKE128, this means 40 bytes instead of 200.

- Finally, on top of these, we build two committing AE schemes. The first one uses the nonce-based authenticated encryption mode that we call ODWrap and that is similar to SpongeWrap [BDPV11b]. The second one builds upon the Deck-BO mode [BDH+22], an online version of the SIV AE mode, using upperdeck on top of OD.

We prove that the confidentiality, integrity and commitment security of our two AE schemes, and the distinguishing security of the intermediate constructions, all reduce to the security of the underlying sponge functions, i.e., SHAKE and TurboSHAKE. As SHAKE and TurboSHAKE are resistant against inner collisions, the fact that tags in our AE schemes are essentially hashes of all inputs makes them naturally committing.

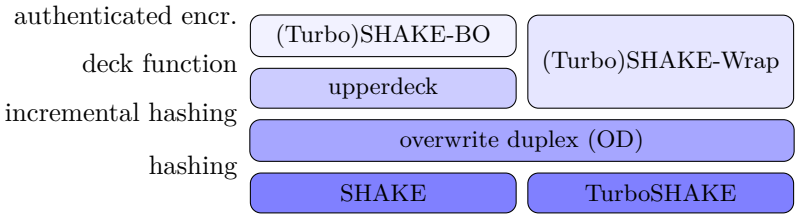

Figure 1: Hierarchy

2 Preliminaries

In this section, we first introduce our notation. Then we recall some definitions related to authenticated encryption and the jammin cipher, our security reference. Finally we recall SHAKE and TurboSHAKE and define how we encode our inputs and split them into blocks.

2.1 Notation

Most strings that we consider in this work are byte strings and we denote the empty string by ϵ. The byte length of a string X is denoted by $|X|$. The concatenation of two strings X, Y is denoted as $X||Y$ and their bitwise addition as $X + Y$, with the resulting string having length $\min(|X|, |Y|)$. Bit values are noted with a typewriter font, such as `01101`. Byte string values are noted with a typewriter font and preceded by `0x`, e.g., `0x1F`. The repetition of a bit is noted in exponent, e.g., $\texttt{0}^3 = \texttt{000}$. Similarly, for bytes, e.g., $\texttt{0x00}^3 = \texttt{0x000000}$. In a sequence of m strings, we separate the individual strings with a comma, i.e., $x_1, x_2, \ldots x_n$. Finally, $\perp$ denotes an error code.

2.2 AE and the jammin cipher

A nonce-based authenticated encryption scheme with associated data (AD) is usually specified as a pair of algorithms (`wrap`, `unwrap`). `wrap` is a deterministic function that takes as input a 4-tuple (K, N, A, P) with key K, nonce N, associated data A, and message P, and outputs a ciphertext C. The ciphertext C includes the tag and hence has size $|P| + \tau$, where τ is called *expansion* or *tag length* in bytes. `unwrap` takes a 4-tuple (K, N, A, C) and returns a plaintext P or an error $\perp$. A scheme is called correct if

$$\texttt{unwrap}(K, N, A, \texttt{wrap}(K, N, A, P)) = P$$

for any tuple (K, N, A, P). The separation between a nonce and A is for many schemes an artificial one and one may merge the nonce N into the associated data A. In this, simpler, representation the nonce data element is transformed into a *nonce requirement*, namely that the A field shall be unique for each plaintext P for a given key K.

The typical formulation of security notion for an AE scheme requires an adversary to distinguish between two oracles. The first one, *in the real world* is the AE scheme. The second one, *in the ideal world* outputs random responses. In a `wrap` query with (A, P), the real oracle returns $\texttt{wrap}(K, A, P)$ while the ideal oracle returns a random string of $|P| + \tau$ bytes (with $\tau = t/8$ and t the tag length in bits). In this work, we will use the *jammin cipher* as ideal model [BDH+22].

The aim of the *jammin cipher* is to serve as an ideal-world model when proving or claiming a distinguishing bound for AE schemes, including if they support *sessions*. In modern applications, parties do not limit to exchange individual messages, but usually have to encrypt and authenticate sequences of messages in bi-directional communications. A session deals with the authentication of such sequences of messages by intermediate tags, which ensure that a message is authenticated in the context of previously sent messages. Also, the support for sessions allows using it as an online AE scheme: a long message can be split in chunks that are encrypted separately, and each ciphertext authenticates the partial message decrypted up to that point. Note that any session-supporting AE can also naturally work in a non-session mode by limiting the sessions to a single message A, P.

The jammin cipher is parameterized by a *ciphertext expansion* function WrapExpand(), which here we specify as always adding $t = 8\tau$ bits as tag, hence $\text{WrapExpand}(p) = p + t$. It supports bi-directional communication with wrap and unwrap calls in any order, and it achieves the highest possible security, i.e., the cryptograms it produces are as random as injectivity allows, while behaving deterministically, meaning equal inputs give same output. More details about the jammin cipher can be found in [BDH+22] and in Appendix A.

In the jammin cipher, the *encryption context* of a `wrap` query is the sequence composed of the (A, P) inputs received during the previous `wrap` and `unwrap` queries and of the A 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.

For some AE modes a strong bound on the distinguishing advantage from the jammin cipher can be proven without a nonce requirement. Such modes will leak information due to the fact that equal ciphertexts with equal encryption contexts indicate equal plaintexts. Other AE modes require the associated data A of the first message of a session to be a nonce for a provable strong bound on the distinguishing advantage from the jammin cipher. They are usually more efficient but the consequences of nonce violation are more serious. In the sequel, we define one scheme on which we put a nonce requirement, while for the other we do not.

2.3 SHAKE and TurboSHAKE

SHAKE128 and SHAKE256 are two eXtendable Output Functions (XOF) standardized by NIST in [NIS15]. They are defined on top of the Keccak$[c]$ sponge function. Both internally use the permutation Keccak-$p[1600, n_\text{r} = 24]$ and are parameterized by the capacity c. The capacity determines the security strength level as well as the efficiency since the number of bits a sponge function can absorb or squeeze per call to the underlying permutation is $r = b - c$. Here, b is the permutation width and r the (bit) rate, and we denote with $R = r/8$ the rate in bytes. In particular, this is $c = 256$ for SHAKE128 and $c = 512$ for SHAKE256, giving (byte) rates of $R = 136$ and $R = 168$, respectively.

An instance of SHAKE takes as input a variable length string M and an output length

d and appends four bits to M before processing it. In particular,

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

TurboSHAKE is a family of XOFs that was originally introduced for use in KangarooTwelve [BDP+18] and later formally defined in [BDH+23]. As SHAKE, it is parameterized by the capacity c and is based on the permutation Keccak-p[1600, n_r] but with $n_r = 12$ rounds.

It was introduced with the aim of having a more efficient version of Keccak. We consider two instances: TurboSHAKE128 with $c = 256$ and TurboSHAKE256 with $c = 512$.

An instance of TurboSHAKE takes as input a message M, that is a byte string of variable length and a domain separator parameter D, a byte with value in the range $[\texttt{0x01}, \ldots, \texttt{0x7F}]$. The function processes these two inputs as follows. It appends the byte D to M and pads the resulting string with the minimum number of bytes `0x00` until $M' = M||D||\texttt{0x00}^*$ has length a multiple of the rate R. Then it bitwise adds the byte `0x80` is to the last byte of M'.

2.4 Byte strings and trailers

In most real-world use cases keys, plaintexts, tags or associated data that are strings of bytes. Nevertheless, as we go down the stack of our constructions as depicted in Figure 1, the need for domain separation may require us to go down to the bit level and consider strings that have a length that is not a multiple of 8.

In this paper, we distinguish between payload strings, made of bytes, and *trailers* that are bit strings of length at most 7 bits; We encode trailers in single bytes and compactly specify constant trailers as an integer value, similarly to the approach taken in the definition of TurboSHAKE's domain separation byte D [BDH+23]. For a n-bit trailer $e = (e_0, e_1, \ldots, e_{n-1})$, we define its integer equivalent $E = \text{padint}(e)$, with

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

For instance, $\text{padint}(\epsilon) = 1$ and $\text{padint}(\texttt{011}) = 14$. The inverse function, $\text{unpad}(S)$, interprets the binary representation of $S \geq 1$ as a string of bits, from the least to the most significant bit, and removes the last bits '`0`' of S, if any, then the last bit '`1`'.

Representing a trailer with an integer value, sufficiently small to fit in a byte, makes descriptions match implementations closely. A byte representing a trailer can be easily integrated into a byte string. The length of the trailer is unambigous: Using the padint function works like padding with the pattern $\texttt{10}^*$, where the padding bit '`1`' comes from the 2^n term. In some cases, this padding may even coincide with padding requirements in lower levels, thereby simplifying layered descriptions.

Two layers may need to add trailers for domain separation. An easy case is when the lower layer prepends a bit e to a trailer, represented as the integer E, that come from the upper layer. The resulting trailer is represented as $E' = \text{padint}(e||\text{unpad}(E))$, which simplifies to $E' = e + 2E$. For simplicity, in the sequel, we do not distinguish between a trailer as a short string of bits and its integer representation.

2.5 Parsing into blocks

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

Algorithm 1 Functions to parse byte strings into block sequences

Definition of parse(X, ℓ_1, ℓ_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. The last block may be shorter than ℓ_1 if $|\mathbf{x}| = 1$, and shorter than ℓ_2 if $|\mathbf{x}| > 1$.

3 Duplexing (Turbo)SHAKE

In this section, we define the *overwrite duplex* (OD) interface to sponge functions, and in particular for the SHAKE and TurboSHAKE XOFs. It is the lower layer on top of which we will build our authenticated encryption schemes. In a nutshell, the OD object is stateful and provides incremental hashing.

The OD construction combines the ideas of the duplex and overwrite constructions, both introduced in [BDPV11b]. Similarly to the duplex construction, the caller can invoke it as many times as needed with a length-bounded string, and each time the object produces a digest of the sequence of strings received so far. Unlike the duplex construction, however, the (payload) input block overwrites the state instead of being XORed into it.

We define the OD construction in terms of the permutation underlying the corresponding sponge function and prove that the security strength of OD and of sponge are equal.

3.1 Specification of the OD interface

The OD object is parameterized with a permutation f, a block length ρ (in bytes) and a capacity c (in bits). 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, $\rho = (1600 - c - 64)/8$.

An OD object can be created by initializing it with a secret key K or by cloning another OD object. There are two cloning methods, one that discards the outer part and one that does not. It supports incremental hashing by means of duplexing calls, where each call takes as input a block and a trailer (B, E), with $|B| \leq \rho$ and $E \in \{1, \ldots, 63\}$, and returns up to ρ bytes of output. The output of duplexing call depends on the sequence of blocks and trailers (B_i, E_i) received so far. The OD object keeps track of what it returned and the squeezeMore method allows returning more output in between duplexing calls. See Figure 2 for an illustration.

Algorithm 2 defines the OD construction and uses the following conventions. For an input block B shorter than ρ bytes, let pad10*(B) be padding of B to the block length ρ bytes: pad10*$(B) = B||\texttt{0x01}||\texttt{0x00}^*$ such that the resulting string has ρ bytes. We do not pad an input block B of exactly ρ bytes, but we distinguish between padded and not-padded blocks in the domain separator byte D, namely, $D = 2E$ in the former case and $D = 1 + 2E$ in the latter.

While a duplexing call overwrites the state with the (padded) input block, it XORs the input trailer after applying to it an encoding function trailenc(). The trailer encoding function is specific for the underlying sponge functions. For TurboSHAKE, we define it as

$$\text{TurboSHAKE:} \quad \text{trailenc}(D) = D||\texttt{0x00}^6||\texttt{0x80}\,.$$

The format of trailenc(D) is chosen so as to match that TurboSHAKE's domain separation byte and padding, see Theorem 1 for more details. For SHAKE, this is slightly different to account for the suffix `1111` that FIPS 202 appends to the input string: We define the encoding function as

$$\text{SHAKE:} \quad \text{trailenc}(D) = D||\texttt{0x1F}||\texttt{0x00}^5||\texttt{0x80}\,,$$

since padint(1111) = 0x1F.

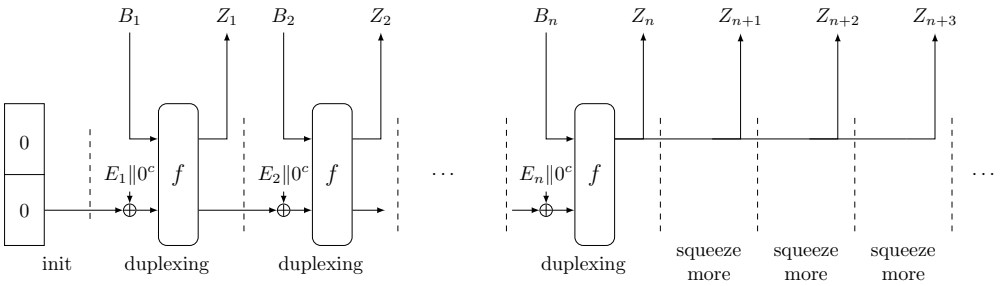

Figure 2: Illustration of the OD construction. Note that neither the padding of the input blocks B_i nor the encoding of the trailer E_i is explicitly depicted in this figure, please see Algorithm 2 for more details on this.

Algorithm 2 Definition of $\text{OD}[f, \rho, c]$

Parameters: b-bit permutation f, payload byte rate ρ and capacity c

Interface OD.initialize()
Initialize OD's attributes $s \leftarrow 0^b = \texttt{0x00}^{b/8}$ and $o \leftarrow \rho$

Interface OD.duplexing(B, E, ℓ) with $|B| \leq \rho$, $E \in \{1, \ldots, 63\}$ and $\ell \leq \rho$
if $|B| = \rho$ **then**
  Replace the first ρ bytes of s with B
  XOR the next bytes of s with trailenc$(1 + 2E)$
else
  Replace the first ρ bytes of s with pad10*(B)
  XOR the next bytes of s with trailenc$(0 + 2E)$
$s \leftarrow f(s)$
return the first ℓ bytes of s, then set $o \leftarrow \ell$

Interface OD.squeezeMore(ℓ) with $\ell \leq \rho - o$
return ℓ bytes of s starting from offset o, then update $o \leftarrow o + \ell$

Interface OD.clone()
return a new $\text{OD}[f, \rho, c]$ object initialized with $(s, o) = (\text{OD}.s, \text{OD}.o)$

Interface OD.cloneCompact()
return a new $\text{OD}[f, \rho, c]$ object initialized with $s = \text{OD}.s$ except the first ρ bytes that are set to 0x00 and with $o = \rho$

3.2 Equivalence to (Turbo)SHAKE

We here show that any output of an $\text{OD}[f, \rho, c]$ object with $f = \text{Keccak-}p[1600, n_\text{r} = 12]$ can be obtained by calls to TurboSHAKE[c] with an input that can be formed from the strings absorbed and squeezed by the OD object. This means that **any attack on the OD object translates to an attack on the TurboSHAKE instance and hence they are security-equivalent**. We focus on TurboSHAKE128, but the proofs for TurboSHAKE256, SHAKE128 and SHAKE256 are essentially the same.

The output of an OD object 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 OD.duplexing(B_i, E_i, ℓ_i) since its initialization with OD.initialize(). The output of any intermediate OD.squeezeMore(ℓ) calls can be seen as the delayed output of the most recent duplexing calls. Moreover, the output lengths $\ell_i \leq \rho$ for $i < n$ do not influence the output of the n-th duplexing call. Without loss of generality, we therefore treat the case of *full-block outputs*, that is, output blocks of ρ bytes.

Theorem 1. *The full-block output of the n-th duplexing call to OD is the ρ-byte output of TurboSHAKE128 applied to an input that is an injective mapping of $(B_1, E_1, \ldots, B_n, E_n)$.*

Proof. 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 $|B_i| < \rho$, $\beta_i \leftarrow \text{pad10}^*(B_i)$ and $D_i = 0 + 2E_i$. Otherwise $\beta_i \leftarrow B_i$ and $D_i = 1 + 2E_i$. As the parity of D_i indicates whether padding was applied and the padding itself is injective, this mapping is injective.

We denote by $\text{TS}(M, D)$ the output of TurboSHAKE128 with byte string M and trailer D as inputs, truncated to its first ρ bytes, and by $\text{OD}(\beta_1, D_1, \ldots, \beta_n, D_n)$ 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 $n = 1$: we express $\text{OD}(\beta_1, D_1)$ 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 $\text{OD}(\beta_1, D_1) = \text{TS}(\beta_1, D_1)$. Clearly, the mapping from (β_1, D_1) to the TurboSHAKE128 input is injective.

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 [BDPV11b], 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: $\text{OD}(\beta_1, D_1, \beta_2, D_2) = \text{TS}(\beta_1 || \text{trailenc}(D_1) || (\beta_2 \oplus \text{OD}(\beta_1, D_1)), D_2)$.

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

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

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

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 $(\beta_1, D_1, \ldots, \beta_{n-1}, D_{n-1}) \rightarrow (O(\beta_1, D_1, \ldots, \beta_{n-1}), D_{n-1})$ is injective, then $(\beta_1, D_1, \ldots, \beta_n, D_n) \rightarrow (O(\beta_1, D_1, \ldots, \beta_n), D_n)$ 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 $\text{OD}(\beta_1, D_1, \ldots, \beta_{n-1}, D_{n-1})$ is fixed, and XORing β_n with it preserves the injectivity. □

3.3 PRF security

In our constructions, we always input a key in the first duplexing call of an OD object and use it as a PRF of the subsequent inputs. Referring to Theorem 1, having a key in the first duplexing call is equivalent to calling the underlying sponge function with the key as a prefix. Distinguishing the keyed OD from a random oracle is therefore upper bounded by the sum of two distinguishing advantages:

- guessing the key correctly—this is upper bounded by $N/2^k$ with N the time complexity expressed in the (equivalent) number of calls to the permutation and k the length of the key in bits;

- distinguishing the keyed sponge function from a random oracle—following the claimed security of (Turbo)SHAKE, this is upper bounded by $M^2/2^{c+1}$ with M the data complexity expressed in input and output blocks.

4 (Turbo)SHAKE-Wrap

In this section we specify nonce-based authenticated encryption schemes based on the ODWrap mode. We first specify it and discuss its distinguishing advantage from the jammin cipher and its committing security.

4.1 Specification of ODWrap

In Algorithm 3, we specify the nonce-based session-supporting authenticated encryption mode ODWrap on top of OD. This mode is inspired by spongeWrap defined in [BDPV11b] and is illustrated in Figure 3.

Upon initialization, the underlying OD object is loaded with a secret key K. A wrap call takes as input associated data A and plaintext P and returns a ciphertext C of $|P| + \tau$ bytes, with τ the tag length in bytes. An unwrap call takes as input associated data A and ciphertext C with $|C| \geq \tau$ and returns a plaintext P of $|C| - \tau$ bytes or an error $\perp$ in case the ciphertext is not authentic. Before unwrapping, a clone is made of the OD object, allowing a roll-back in case of an invalid cryptogram.

Each ciphertext authenticates all previous messages in the session since initialization. Both A and P can be empty, leading to 4 cases. If P is empty, the *ciphertext* is basically a tag of length τ. For confidentiality it is important that the sequence of A strings before the first plaintext P is each time different for a given key K.

A wrap call first splits the A and P in sequences of blocks of ρ or less and absorbs them in a number of serial duplexing calls of the underlying OD object, where the trailer is used to indicate type of block and the purpose of the corresponding OD 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 τ bytes of OD output serves as tag.

If there is no A 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 that was already used for tag generation, this block will be at most $\rho - \tau$ bytes long.

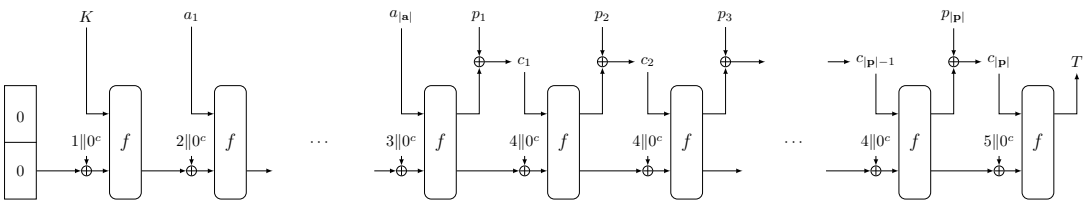

Figure 3: Illustration of the ODWrap mode merged with the underlying OD construction. This figure shows a first call W.initialize(K) and then W.wrap(A, P). Neither the padding of the input blocks nor the encoding of the trailers is explicitly depicted in this figure.

Algorithm 3 Definition of ODWrap[OD object with ρ, t].

Require: OD = OD$[f, \rho, c]$

Interface: W.initialize(K) with $K \in \mathbb{Z}_2^*$
OD.initialize()
OD.duplexing($K, 1, 0$)
$\tau = t/8$

Interface: $C \leftarrow W$.wrap(A, P)
$\mathbf{a} \leftarrow$ parse(A, ρ, ρ)
if ($|A| > 0$) AND ($|P| > 0$) **then**
  $\mathbf{p} \leftarrow$ parse(P, ρ, ρ)
  for $i = 1$ to $|\mathbf{a}| - 1$ **do** OD.duplexing($a_i, 2, 0$)
  $c_1 \leftarrow p_1 +$ OD.duplexing($a_{|\mathbf{a}|}, 3, |p_1|$)
  for $i = 2$ to $|\mathbf{p}|$ **do** $c_i \leftarrow p_i +$ OD.duplexing($c_{i-1}, 4, |p_i|$)
  $T \leftarrow$ OD.duplexing($c_{|\mathbf{p}|}, 5, \tau$)
else if ($|A| = 0$) AND ($|P| > 0$) **then**
  $\mathbf{p} \leftarrow$ parse($P, \rho - \tau, \rho$)
  $c_1 \leftarrow p_1 +$ OD.squeezeMore($|p_1|$)
  for $i = 2$ to $|\mathbf{p}|$ **do** $c_i \leftarrow p_i +$ OD.duplexing($c_{i-1}, 4, |p_i|$)
  $T \leftarrow$ OD.duplexing($c_{|\mathbf{p}|}, 5, \tau$)
else
  for $i = 1$ to $|\mathbf{a}| - 1$ **do** OD.duplexing($a_i, 2, 0$)
  $T \leftarrow$ OD.duplexing($a_{|\mathbf{a}|}, 6, \tau$)
return C, the concatenation of $\mathbf{c}$ (empty if $|P| = 0$) and T

Interface: $P \leftarrow W$.unwrap(A, C), may return $\perp$
if ($|C| < \tau$) **then return** $\perp$
OD$' \leftarrow$ OD.clone()
$(C'||T) \leftarrow C$ with $|T| = \tau$
$\mathbf{a} \leftarrow$ parse(A, ρ, ρ)
if ($|A| > 0$) AND ($|C| > \tau$) **then**
  $\mathbf{c} \leftarrow$ parse(C', ρ, ρ)
  for $i = 1$ to $|\mathbf{a}| - 1$ **do** OD.duplexing($a_i, 2, 0$)
  $p_1 \leftarrow c_1 +$ OD.duplexing($a_{|\mathbf{a}|}, 3, |c_1|$)
  for $i = 2$ to $|\mathbf{c}|$ **do** $p_i \leftarrow c_i +$ OD.duplexing($c_{i-1}, 4, |c_i|$)
  $T' \leftarrow$ OD.duplexing($c_{|\mathbf{c}|}, 5, \tau$)
else if ($|A| = 0$) AND ($|C| > \tau$) **then**
  $\mathbf{c} \leftarrow$ parse($C', \rho - \tau, \rho$)
  $p_1 \leftarrow c_1 +$ OD.squeezeMore($|c_1|$)
  for $i = 2$ to $|\mathbf{c}|$ **do** $p_i \leftarrow c_i +$ OD.duplexing($c_{i-1}, 4, |c_i|$)
  $T' \leftarrow$ OD.duplexing($c_{|\mathbf{c}|}, 5, \tau$)
else
  for $i = 1$ to $|\mathbf{a}| - 1$ **do** OD.duplexing($a_i, 2, 0$)
  $T' \leftarrow$ OD.duplexing($a_{|\mathbf{a}|}, 6, \tau$)
if $T = T'$ **then**
  return P, the concatenation of $\mathbf{p}$ (empty if $|C| = \tau$)
OD $\leftarrow$ OD$'$
return $\perp$

4.2 Indistinguishability from the jammin cipher

The security strength of ODWrap is defined by the advantage of distinguishing it from the jammin cipher when keyed with an unknown uniformly selected k-bit key. When an adversary can start multiple sessions with first wrap calls that have the same associcated data A but different single-block plaintexts P, ODWrap can be immediately distinguished from the jammin cipher as the keystream used for encrypting the different plaintexts P is the same, and therefore $P + C$ is the same for all these messages. For the jammin cipher this is extremely unlikely to happen. Therefore, for its security, ODWrap requires the A of the first wrap call of all sessions with the same K to be unique, hence a nonce.

The distinguishing advantage is upper bound by the sum of two distinguishing advantages:

- between ODWrap when instantiated with a random oracle instead of (Turbo)SHAKE and the jammin cipher;
- between (Turbo)SHAKE where the input starts with a secret key K and a random oracle.

It is easy to see that the distinguishing advantage between ODWrap when instantiated with a random oracle and the jammin cipher is upper bound by $q_{\text{forge}}/2^t$ with q_{forge} the number of forgery attempts. In short, each call to the underlying random oracle has a different input string thanks to the domain separation bits and the fact that the A of the first wrap call is a nonce. Therefore all keystreams and tags are uniformly random and therefore also all ciphertexts C. The only way to distinguish it from the jammin cipher is by a successful forgery: attempting to unwrap a cryptogram that was not generated in a call to wrap. As the tag has t bits and all tags are uniformly random, the success probability for each attempt is 2^{-t}. After q_{forge} attempts, this is upper bound by $q_{\text{forge}}/2^t$. We summarize this in Theorem 2.

Theorem 2. *Let $\mathcal{D}$ be any fixed deterministic adversary whose goal is to distinguish (Turbo)SHAKE-Wrap keyed with an unknown uniformly selected k-bit key from $\mathcal{J}^{+t}$, the jammin cipher with* $\text{WrapExpand}(p) = p + t$*. If in the queries of $\mathcal{D}$ the encryption context is a nonce, then the advantage is*

$$\Delta_{\mathcal{D}}(\textit{(Turbo)SHAKE-Wrap}\ ;\ \mathcal{J}^{+t}) \leq \frac{q_{\mathsf{unwrap}}}{2^t} + \frac{N}{2^k} + \frac{M^2}{2^{c+1}}$$

with q_{unwrap} the number of unwrap calls $\mathcal{D}$ makes, N the time complexity expressed in the (equivalent) number of calls to the permutation, and M the data complexity expressed in input and output blocks.

4.3 Committing security

In (Turbo)SHAKE-Wrap the tag is the result of hashing an injective encoding of 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, which is $t/2$ bits for a tag length of t bits. Therefore, for tag length $t = 256$, the scheme guarantees a committing security strength of 128 bits and for tag length $t = 512$, the scheme guarantees committing security strength of 256 bits. If for a given application less committing security strength is considered sufficient, a smaller tag length can be chosen, say $t = 160$ for 80 bits of security.

Note that considering the collisions resistance of the underlying hash function gives us the strongest notion of committing AE. There may be cases where an attacker must find a second interpretation of a given, fixed, ciphertext. In that case, the requirement for the underlying hash function is rather second preimage resistance, and the tag length can be equal to the security strength level.

5 The upperdeck construction

In this section, we define a construction to build a deck function on top of OD called upperdeck and discuss its security.

5.1 Specification of upperdeck

A *doubly-extendable cryptographic keyed (deck)* function is a keyed primitive that natively supports variable input and output lengths [DHVV18]. Examples of deck functions are KRAVATTE [BDH+17] and XOOFFF [DHVV18], two instances of the Farfalle construction [BDH+17] based on the KECCAK-p and XOODOO permutations, respectively. One of the main properties of deck functions is incrementality: the cost of computing $F(X, Y)$ depends only on the processing of Y if $F(X)$ was previously computed. The incrementality property of deck functions allows to support sessions in a natural way.

In Algorithm 4, we define an interface for a deck function built on top of OD. The deck function is parametrized with the OD object OD. Upon initialization the key K is absorbed with a duplexing call with block $B = K$ and trailer $E = \text{padint}(\epsilon) = 1$. Then, the user can absorb an arbitrarily long string and squeeze as many bits as needed, via a sequence of calls to the underlying duplex object. This is illustrated in Figure 4.

The input string is given as a byte string X and a trailer E, which typically contains domain separation bits coming from the higher level. The string X is first split into blocks $x_1, x_2, \ldots, x_n$ such that $X = x_1||x_2||\ldots||x_n$. The size of each block is ρ except for the last block that can be shorter. Therefore, $n = \lceil |X|/\rho \rceil$. Each block is processed by a duplexing call, with block $B_i = x_i$ and with a domain separation bit that indicates whether the current block is the last one of the input string (bit `1`) or not (bit `0`). Together with the last block, we also absorb the trailer E. More formally, we make duplexing calls with $E' = \text{padint}(\texttt{0}) = 2$ for $i < n$ and $E' = \text{padint}(\texttt{1}||\text{unpad}(E)) = 1 + 2E$ for the last block. When the last block is absorbed, at most ρ bytes are squeezed. More output bits can be obtained via duplexing calls with empty blocks and trailer $E' = \text{padint}(\texttt{0}) = 2$.

We also specify a clone function that limits the copy to the last $b/8 - \rho$ bytes of the state via a call to OD.cloneCompact().

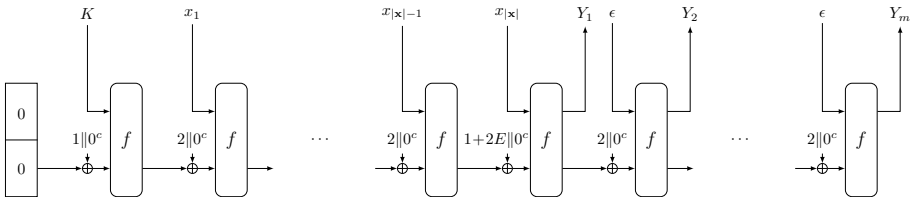

Figure 4: Illustration of the upperdeck mode merged with the underlying OD construction. This figure shows the call to F.initialize(K), followed by F.absorbAndSqueeze(X, E, ℓ) that returns Y. The value T is $1 + 2E$. Again, neither the padding of the input blocks nor the encoding of the trailers is explicitly depicted in this figure.

5.2 Security analysis

We consider adversaries that aim at distinguishing between the instantiation of a construction using a fixed underlying function (e.g., (Turbo)SHAKE), keyed with a secret and uniformly chosen k-bit key, and a random oracle. The distinguishing advantage between an upperdeck instance and a random oracle can be reduced to that of distinguishing an OD from a random oracle, which is given in Section 3.3. This follows from the fact that

Algorithm 4 Definition of upperdeck

Parameters: overwrite duplex object OD = OD$[f, \rho, c]$

Interface: F.initialize(K)
OD.initialize()
OD.duplexing($K, 1, 0$) // 1 encodes ϵ

Interface: F.absorbAndSqueeze(X, E, ℓ) returns Y
$\mathbf{x} \leftarrow$ parse(X, ρ, ρ)
for $i = 1$ to $|\mathbf{x}| - 1$ **do**
  OD.duplexing($x_i, 2, 0$)
$Y \leftarrow$ OD.duplexing($x_{|\mathbf{x}|}, 1 + 2E, \min(\ell, \rho)$)
while $|Y| < \ell$ **do**
  $Y \leftarrow Y$||OD.duplexing($\epsilon, 2, \min(\ell - |Y|, \rho)$)
return Y

Interface: F.clone() returns F'
return a new *deck* object F' initialized with OD.cloneCompact()

in upperdeck the sequence of input strings (i.e., byte strings and trailers) is injectively converted to a sequence of input blocks and trailers.

6 (Turbo)SHAKE-BO

In this section, we define the (Turbo)SHAKE-BO session-supporting AE scheme on top of an upperdeck instance, and provide security reductions for it.

6.1 Specification of (Turbo)SHAKE-BO

In Algorithm 5, we specify (Turbo)SHAKE-BO on top of the upperdeck construction instanciated with (TurboSHAKE). This scheme follows the Deck-BO mode, the simplest of the four robust modes presented in [BDH+22]. It is based on the Synthetic Initial Value (SIV) approach [RS06] and supports sessions.

Algorithm 5 follows the specifications of Deck-BO given in [BDH+22], although we make explicit the steps of the Feistel network inside it, we use the interface of upperdeck and we put the domain separation bits in the trailers (see Section 2.4). This is illustrated in Figure 5.

An instance of Deck-BO is parameterized with the deck function F and the tag length in bytes τ. Upon initialization, the deck function F is initialized with the key K. A wrap call takes as input (possibly empty) associated data A and plaintext P. As output, it gives a ciphertext Z, that encrypts P, and an authentication tag T of τ bytes. The tag is generated by absorbing A and P, if non-empty, and by squeezing the first τ bytes of the state. The tag T is thus a pseudorandom function of A and P (and all previous messages) 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, the state is cloned but, taking advantage of the overwrite property of the duplex object, only $b/8 - \rho$ bytes of the state must be copied. Upon unwrap, a copy of the state is used to be able to go back to the original state in case of failure. If the procedure succeeds then the state is updated with the working copy.

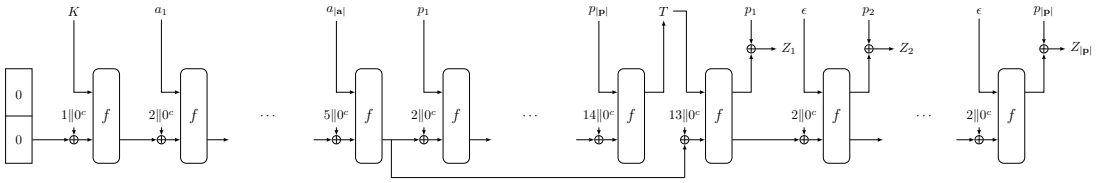

Figure 5: Illustration of the Deck-BO mode merged with the underlying upperdeck mode and the OD construction. This figure shows the call to Q.initialize(K), followed by Q.wrap(A, P) that returns C. Again, neither the padding of the input blocks nor the encoding of the trailers is explicitly depicted in this figure.

6.2 Indistinguishability from the jammin cipher

The security strength of (Turbo)SHAKE-BO is defined by the advantage of distinguishing it from the jammin cipher when keyed with an unknown uniformly selected k-bit key. This distinguishing advantage is upper bound by the sum of two distinguishing advantages:

- between Deck-BO when instantiated with a random oracle as deck function and the jammin cipher;
- between upperdeck instantiated with (Turbo)SHAKE whose first block is a secret key K and a random oracle.

The first is covered by the proven bound for Deck-BO given in [BDH+22, Theorem 3] which is the probability of a successful forgery plus the probability of tags colliding. In a forgery attempt, for any unwrap call that the adversary makes, the tag is compared with a tag generated with the underlying random oracle. This is a uniformly generated string, and the probability that is equal to the tag is 2^{-t}. For q_{unwrap} unwrap calls, this gives $\frac{q_{\mathsf{unwrap}}}{2^t}$. Tags collision happens with probability $2^{-t} \cdot \binom{q}{2}$ for q wrap calls. If we consider at most $\sigma(\text{context})$ wrap queries with the same context, this gives $\sum_{\text{context}} \frac{\binom{\sigma(\text{context})}{2}}{2^t}$.

Combining the reasoning in Sections 3.3 and 5.2, the second is $\frac{N}{2^k} + \frac{M^2}{2^{c+1}}$, with N the number of offline calls to the permutation and M the data complexity expressed in input and output blocks.

We capture the security of (Turbo)SHAKE-BO in Theorem 3 by adding these two advantages.

Theorem 3. *Let $\mathcal{D}$ be any fixed deterministic adversary whose goal is to distinguish (Turbo)SHAKE-BO keyed with an unknown uniformly selected k-bit key from $\mathcal{J}^{+t}$ (the jammin cipher with* WrapExpand$(p) = p + t$*). Assume that the (Turbo)SHAKE instance stands by its claimed security [BDPV11a, BDH+23]. Then its advantage is*

$$\Delta_{\mathcal{D}}(\textit{(Turbo)SHAKE-BO}\,;\,\mathcal{J}^{+t}) \leq \frac{q_{\mathsf{unwrap}}}{2^t} + \sum_{\text{context}} \frac{\binom{\sigma(\text{context})}{2}}{2^t} + \frac{N}{2^k} + \frac{M^2}{2^{c+1}}$$

with q_{unwrap} the number of unwrap calls that $\mathcal{D}$ makes, σ(context) the number of wrap queries with $P \neq \epsilon$ for a given context *value, N the number of offline calls to* Keccak-p[1600]*, and M the data complexity expressed in input and output blocks.*

6.3 Committing security

The committing strength of (Turbo)SHAKE-BO is given by the (in)ability of generating tag collisions. By construction, the tag is computed as the hash of all input data via

Algorithm 5 Definition of (Turbo)SHAKE-BO

Parameters: deck function F, expansion length τ

Interface: Q.initialize(K)
F.initialize(K)

Interface: Q.wrap(A, P) returning C
if $|P| = 0$ **then**
  return $C \leftarrow F$.absorbAndSqueeze($A, 4, \tau$) // 4 encodes `00`
if $|A| \neq 0$ **then**
  F.absorbAndSqueeze($A, 5, 0$) // 5 encodes `10`
$F' \leftarrow F$.clone()
$T \leftarrow F$.absorbAndSqueeze($P, 14, \tau$) // 14 encodes `011`
$Z \leftarrow P + F'$.absorbAndSqueeze($T, 13, |P|$) // 13 encodes `101`
return $C \leftarrow Z||T$

Interface: Q.unwrap(A, C) returning P or $\perp$
if $|C| = \tau$ **then**
  $F' \leftarrow F$.clone()
  $P \leftarrow \epsilon$
  $C' \leftarrow F'$.absorbAndSqueeze($A, 4, \tau$) // 4 encodes `00`
  if $C' \neq C$ **then return** $\perp$
else if $|C| > \tau$ **then**
  parse C as $Z||T$ with $|T| = \tau$
  $F' \leftarrow F$.clone()
  if $|A| \neq 0$ **then**
    F'.absorbAndSqueeze($A, 5, 0$) // 5 encodes `10`
  $F'' \leftarrow F'$.clone()
  $P \leftarrow Z + F''$.absorbAndSqueeze($T, 13, |Z|$) // 13 encodes `101`
  $T' \leftarrow F'$.absorbAndSqueeze($P, 14, \tau$) // 14 encodes `011`
  if $T' \neq T$ **then return** $\perp$
else return $\perp$
$F \leftarrow F'$
return P

(Turbo)SHAKE. Therefore, the committing security strength is given by the minimum of $c/2$ and $t/2$, half the tag length t. In (Turbo)SHAKE, $c = 256$ or 512, and if we choose $t \geq c$ this guarantees a committing security strength of 128 and 256 bits, respectively.

7 Performance

In this section, we discuss the performance of the different schemes, {TurboSHAKE, SHAKE} $\times$ $\{128, 256\}$ $\times$ {-Wrap, -BO}. Giving performance for various platforms would never be exhaustive, so instead we express the cost relative to that of hashing with the standard function SHAKE128. We focus on the cost for long messages, i.e., the slope for increasing sizes of associated data, plaintext or ciphertext. Also, we make the assumption that the evaluation of the permutation dominates the cost.

Table 1 evaluates the cost of the different schemes under these assumptions, and we now explain where the values come from.

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 $\rho = 128$

Table 1: Cost of the different schemes relative to SHAKE128.

	...-Wrap	...-BO	
		A	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

bytes for OD on top of (Turbo)SHAKE128 and (Turbo)SHAKE256, respectively. Due to OD's smaller payload block size, 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, so the cost is divided by 2 in these cases.

Next, we discuss the relative cost of ODWrap. 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 ODWrap 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.

8 Conclusions

Recent works have highlighted the importance of committing security guarantees for authenticated encryption schemes in some real-world settings and applications. Widely used schemes, including AES-GCM and ChaCha20Poly1305, have been proved to not commit to the key or the other inputs.

In this work we introduce session-supporting authenticated encryption schemes with inherent committing properties. Our schemes are in fact based on incremental hashing 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, the proposed schemes 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 number of hardware support for it. The definition of the overwrite duplex object 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.

References

[ABN10] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In Daniele Micciancio, editor, *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010.*

Proceedings, volume 5978 of *Lecture Notes in Computer Science*, pages 480–497. Springer, 2010.

[ADG+22] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 3291–3308. USENIX Association, 2022.

[Alb] Jim Albrecht. Introducing subscribe with Google. https://blog.google/outreach-initiatives/google-news-initiative/introducing-subscribe-google/.

[Ama] Amazon Web Services. AWS encryption SDK. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/introduction.html.

[BDH+17] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Farfalle: parallel permutation-based cryptography. *IACR Transactions on Symmetric Cryptology*, 2017(4):1–38, 2017.

[BDH+22] Norica Băcuieți, Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. Jammin' on the deck. In Shweta Agrawal and Dongdai Lin, editors, *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*, volume 13792 of *Lecture Notes in Computer Science*, pages 555–584. Springer, 2022.

[BDH+23] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. TurboSHAKE. *IACR Cryptol. ePrint Arch.*, page 342, 2023.

[BDP+18] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer, and B. Viguier. KangarooTwelve: Fast hashing based on Keccak-p. In B. Preneel and F. Vercauteren, editors, *Applied Cryptography and Network Security, ACNS 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 400–418. Springer, 2018.

[BDPV11a] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak reference, January 2011. https://keccak.team/papers.html.

[BDPV11b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - SAC 2011, Revised Selected Papers*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.

[BH22] Mihir Bellare and Viet Tung Hoang. Efficient schemes for committing authenticated encryption. In Orr Dunkelman and Stefan Dziembowski, editors, *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*, volume 13276 of *Lecture Notes in Computer Science*, pages 845–875. Springer, 2022.

[CR22] John Chan and Phillip Rogaway. On committing authenticated-encryption. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark,*

September 26-30, 2022, Proceedings, Part II, volume 13555 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2022.

[DGRW18] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 155–186. Springer, 2018.

[DHVV18] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer. The design of Xoodoo and Xoofff. *IACR Trans. Symmetric Cryptol.*, 2018(4):1–38, 2018.

[FLPQ13] Pooya Farshim, Benoît Libert, Kenneth G. Paterson, and Elizabeth A. Quaglia. Robust encryption, revisited. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, volume 7778 of *Lecture Notes in Computer Science*, pages 352–368. Springer, 2013.

[FOR17] Pooya Farshim, Claudio Orlandi, and Razvan Rosie. Security of symmetric primitives under incorrect usage of keys. *IACR Trans. Symmetric Cryptol.*, 2017(1):449–473, 2017.

[GH03] Yitchak Gertner and Amir Herzberg. Committing encryption and publicly-verifiable signcryption. *IACR Cryptol. ePrint Arch.*, page 254, 2003.

[GLR17] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 66–97. Springer, 2017.

[Kra19] Hugo Krawczyk. The opaque asymmetric pake protocol. Internet-Draft draft-krawczyk-cfrgopaque-03, Internet Engineering Task Force, 2019.

[LGR21] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning oracle attacks. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 195–212. USENIX Association, 2021.

[MLGR23] Sanketh Menda, Julia Len, Paul Grubbs, and Thomas Ristenpart. Context discovery and commitment attacks - how to break ccm, eax, siv, and more. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part IV*, volume 14007 of *Lecture Notes in Computer Science*, pages 379–407. Springer, 2023.

[NIS15] NIST. Federal information processing standard 202, SHA-3 standard: Permutation-based hash and extendable-output functions, August 2015. http://dx.doi.org/10.6028/NIST.FIPS.202.

[RS06] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2006.

[Tri] Alex Tribble. Improved client-side encryption: Explicit keyids and key commitment. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/introduction.html.

A The jammin cipher, an ideal-world AE scheme

Algorithm 6 The jammin cipher $\mathcal{J}^{\mathrm{WrapExpand}(p)}$

Parameter: WrapExpand, a t-expanding function
Global variables: codebook initially set to $\perp$ for all, taboo initially set to *empty*

Instance constructor: init(ID)
return new instance inst with attribute inst.history = ID

Instance cloner: inst.clone()
return new instance inst′ with the history attribute copied from inst

Interface: inst.wrap(A, P) returns C
context $\leftarrow$ inst.history; A
if codebook(context; P) $= \perp$ **then**
  $\mathcal{C} = \mathbb{Z}_2^{\mathrm{WrapExpand}(|P|)} \setminus (\mathsf{codebook}(\mathsf{context}; *) \cup \mathsf{taboo}(\mathsf{context}))$
  if $\mathcal{C} = \varnothing$ **then return** $\perp$
  codebook(context; P) $\xleftarrow{\$}$ $\mathcal{C}$
inst.history $\leftarrow$ inst.history; A; P
return codebook(context; P)

Interface: inst.unwrap(A, C) returns P or $\perp$
context $\leftarrow$ inst.history; A
if $\exists! P$: codebook(context; P) $= C$ **then**
  inst.history $\leftarrow$ inst.history; A; P
  return P
else
  taboo(context) $\leftarrow C$
  return $\perp$

In Algorithm 6, we recall the definition of the *jammin cipher* [BDH+22]. 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 instances that share the same identifier $\mathrm{ID}_{\text{Alice and Bob}}$, while Edward and Emma use instances initialized with $\mathrm{ID}_{\text{Edward and Emma}}$. We will informally call an *object* the set of instances sharing the same object identifier. This way, all the instances of the same object have indistinguishable behavior, and this justifies that we collectively call them an object, whereas instances of different objects are completely independent.

The scheme supports two functions: wrap and unwrap. With the wrap function the object computes a cryptogram C from a message that has a plaintext P and associated data A, both arbitrary bit strings. With the unwrap function the object computes the plaintext P from the cryptogram C and A again. The cryptogram C is the encryption of P for a given A.

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

Definition 1. A function $f\colon \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$ is *t-expanding* iff (i) $\forall \ell > 0\colon f(\ell) > f(0)$ and (ii) $\forall \ell\colon f(\ell) \geq \ell + t$.

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 (A, P). So history is a sequence with ID followed by zero, one or more messages (A, P).

A *session* is the process in which the history grows with the messages exchanged so far. The wrap and unwrap functions make the history act as associated data, so that a cryptogram authenticates not only the message (A, 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.

A.1 Properties

The jammin cipher enjoys the following properties:

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

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

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

Deterministic unwrapping: In a given context, an object unwraps equal cryptograms to equal responses. It achieves this by tracking in taboo cryptogram 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 cryptograms 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 2. 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 cryptograms 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 in use cases where the encryption context is a nonce, 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.