Shaking up authenticated encryption

Thursday 10th October, 2024

Joan Daemen
Radboud University
Nijmegen, The Netherlands

Seth Hoffert
Nebraska, USA

Silvia Mella
Radboud University
Nijmegen, The Netherlands

Gilles Van Assche
STMicroelectronics
Diegem, Belgium

Ronny Van Keer
STMicroelectronics
Diegem, Belgium

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 the unique combination of advantages 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.**

1. Introduction

Authenticated encryption (AE) is a cryptographic mechanism that provides both confidentiality and authentication of messages under a secret key. An AE scheme *wraps* a message, composed of plaintext and of associated data, and produces a ciphertext. The ciphertext contains the encrypted plaintext and has redundancy that depends on both the associated data and the plaintext. After wrapping, the sender sends the ciphertext together with the associated data to the receiver. This can then use the AE scheme to *unwrap* the ciphertext using the secret key and the associated data as auxiliary inputs. This operation encompasses the verification of the ciphertext redundancy and, if valid, the decryption and return of the plaintext. If the redundancy does not match, it returns an error code.

Many AE schemes are built using modes of operation on top of AES. For simple AES-based schemes the security breaks down when the amount of data encrypted with a given key approaches 2^{64}, the so-called *birthday bound*. There are AES modes that do not suffer from that limitation but they tend to be more complicated and/or expensive. Our (Turbo)SHAKE-based AE schemes are simple and do not suffer from the birthday bound: They achieve security strength implied by that of the underlying XOF. With SHAKE128 for instance, the security strength is 128.

In modern applications, parties do not limit to exchange individual messages, but usually have to wrap sequences of messages in bi-directional communications. A simple example is secure messaging applications, where users exchange messages in a continuous stream.

A *session* refers 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. Session-supporting AE covers the traditional notion of authenticated encryption of a single message (i.e., a plaintext-associated data pair) as well. Each session then contains a single message.

Session-supporting AE can be implemented with a scheme that maintains a rolling state, that keeps track of the sequence of messages exchanged during the session. When a new session is started, the rolling state of the session-supporting AE scheme is initialized. As Alice sends messages to Bob, the rolling state of the AE scheme is updated with each new message. Encryption and authentication of each message are computed using the rolling state, ensuring the dependance of the intermediate tag on all previous messages in the session. When Bob receives a message from Alice, the AE scheme checks the message in the context of the rolling state to ensure that it has not been tampered with and that it is in the correct sequence. Both parties maintain their AE scheme states, ensuring that the bidirectional communication remains secure throughout the session.

The support for sessions presents a solution to the requirement to be online, as defined in [1] : the ability to do wrapping or unwrapping on the fly with a fixed memory size. A long message can be cut in short segments that are decrypted and authenticated as separate messages in a session.

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 definition 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 the widely used AE schemes AES-GCM and ChaCha20-Poly1305, one can find key-plaintext combinations that lead to equal ciphertexts [2]. 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 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, with K the key, N the nonce, AD the associated data and P the plaintext [3].

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 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.

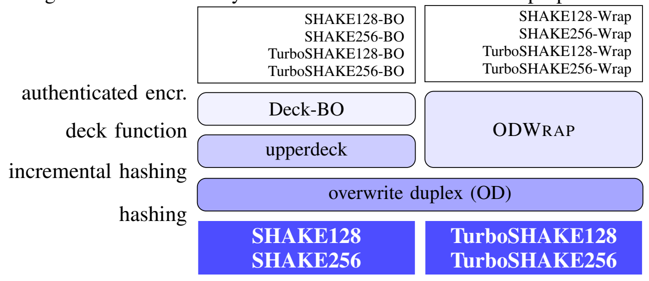
1.1. Our contribution

In this work we propose new AE schemes based on the sponge function families SHAKE and TurboSHAKE, whose collision resistance guarantees commitment security. Our schemes uniquely combine the following advantages. First, 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 [4]. Second, they make use of the standard KECCAK-p permutation that not only receives more and more dedicated hardware support (e.g., in the recent Apple™ processors), but also allows competitive software-only implementations thanks to the TurboSHAKE instances. Third, they are user-friendly in that they only require the variable-length AD of the first message in a session to be a nonce, rather than a fixed-length short data element N for each wrap call. Finally, they offer CMT-4 committing security based on collision-resistance of (Turbo)SHAKE.

Our AE schemes are session-supporting: they allow to split a long message in fragments and wrap them separately, with the ciphertext of a fragment authenticating the partial message decrypted up to that point. We build our schemes in multiple layers, as depicted in Figure 1:

- At the bottom is the sponge function layer that we instantiate with SHAKE and TurboSHAKE functions. These are reviewed in Section 2, together with preliminaries.
- Then, in Section 3, 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, meaning that it can efficiently evaluate increasingly long input by hashing its blocks one at a time, and the current output depends on all the blocks received so far. Like the OVERWRITE mode defined in [5], the given input block overwrites part of the state instead of being (bitwise) added 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 copying 40 bytes instead of 200.
- Finally, on top of these, we build committing AE schemes with two different modes. The first set of schemes, defined in Section 4, use the nonce-based authenticated encryption mode ODWRAP, that is similar to SPONGEWRAP [5]. The second set of schemes, defined in Section 6, build upon the Deck-BO mode [6], a session-supporting version of the SIV AE mode. This mode in turn relies on a deck function that we build with the upperdeck construction on top of OD, and this is presented in Section 5.

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



In an implementation, these multiple layers can be easily merged, as illustrated in Figures 3 and 5. TurboSHAKE, OD and upperdeck accept payload data in byte string inputs and accumulate domain separation bits, coming from different layers, in a separate single-byte trailer.

We prove that 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 a random oracle.

As SHAKE and TurboSHAKE functions with sufficient output length are designed to be collision-resistant and are extensively scrutinized with respect to that property, the fact that tags in our AE schemes are essentially hashes of all inputs makes them naturally CMT-4 committing. With tags of $2s$ bits, this provides s bits of collision-resistance. 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 taken equal to the security strength level s.

Finally, in Section 7, we discuss the performance of the resulting schemes and show their efficiency is competitive.

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 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 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 repetition of a bit is noted in exponent, e.g., $0^3 = \texttt{000}$. Similarly, for bytes, e.g., $\texttt{0x00}^3 = \texttt{0x00}||\texttt{0x00}||\texttt{0x00}$. 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.

In this paper we perform security analysis in the *distinguishability framework* where one bounds the advantage of an adversary $\mathcal{D}$ in distinguishing a real-world system from an ideal-world system.

Definition 1. *Let $\mathcal{O}, \mathcal{P}$ be two collections of oracles with the same interface. 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, this means we are maximizing over all adversaries that use at most these resources,

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

2.2. AE and the jammin cipher

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

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

for any tuple (K, N, AD, P). Both N and AD are inputs to the wrap function with the same effect: they influence the encryption of P to C. We argue that the separation between N and AD is historical and mostly due to the fact that in block cipher modes N and AD are processed differently and it is costly to have a nonce N longer than the block length. From a user perspective, the distinction presents an inconvenient restriction: having a requirement on uniqueness of the variable-length AD would be easier to satisfy, especially in the case of random nonces. There is simply no need for distinct AD and N as we can achieve the same goal with an AD uniqueness requirement. Expressing security of schemes that require a fixed-length nonce can still be accomodated by recasting N to the first $|N|$ bits of AD and stipulating that the first $|N|$ bits of AD shall be a nonce. In other words, the nonce data element is transformed into a *nonce requirement*, namely that the AD field, or its leading $|N|$ bits, shall be unique for each plaintext P for a given key K. In the following we will adopt this convention and omit the term N.

The typical formulation of security notion for an AE scheme requires an adversary to distinguish between two collections of oracles. The first one, *in the real world*, is the AE scheme. The second one, *in the ideal world*, behaves in an ideal way. In this work, we will use the *jammin cipher* as the ideal-world model [6]. Its oracles are stateful instances of objects that output random responses to `wrap` calls, correct plaintexts as a response to `unwrap` calls for valid ciphertexts, and errors for invalid ciphertexts. In a `wrap` query with (AD, P), a real oracle returns $\texttt{wrap}(K, AD, P)$ while an ideal oracle returns a random string of $|P| + \tau$ bytes (with $\tau = t/8$ and t the ciphertext expansion in bits).

As opposed to most other ideal-world AE schemes, the jammin cipher is *operational*: it supports all functions that a real-world scheme does, but with ideal behaviour. The jammin cipher is inherently multi-user in that it supports multiple instances that can exchange encrypted messages on the condition that they have the same ID. Any pair of such instances supports bi-directional communication with any instance able to process wrap and unwrap calls in any order. It can also serve as an ideal world scheme for AE schemes that do not support sessions by by limiting each session to a single message AD, P.

The jammin cipher is parameterized by a *ciphertext expansion* function $\text{WrapExpand}()$ that expresses the length of the ciphertext given the length of the plaintext. For the modes in this paper we have $|C| = \text{WrapExpand}(|P|) = |P| + \tau$: it simply adds τ bytes. It achieves the highest possible security, i.e., 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.

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.

The jammin cipher naturally defines a security notion. We say that an authenticated encryption scheme is a *pseudo-jammin cipher* (PJC) if the advantage in distinguishing it from the jammin cipher is negligible. If the AE scheme requires the context to be a nonce, we speak of a *nonce-based* PJC (nPJC); if there is no such restriction, we speak of a *plain* PJC. 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 AD, or its leading n bits, 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. One of the schemes that we define is a nonce-based PJC, while the other is a plain PJC. A full specification of the jammin cipher and more explanations can be found in Appendix A.

Definition 2 (PJC and nPJC security)**.** *Let* AE *be an authenticated encryption scheme with keys generated ac-*

cording to the distribution $\mathcal{K}$ and $\mathcal{J}^{+t}$ the jammin cipher with $\text{WrapExpand}(p) = p + t$. *We denote the PJC advantage of* AE *by:*

$$\text{Adv}^{\text{PJC}}_{\text{AE}[\mathcal{K}]}(\mathcal{D}_1) = \Delta_{\mathcal{D}_1}(\mathbf{K} \xleftarrow{\$} \mathcal{K}; \text{AE}_{\mathbf{K}} \parallel \mathcal{J}^{+t})\,.$$

We denote the nPJC advantage of AE *by:*

$$\text{Adv}^{\text{nPJC}}_{\text{AE}[\mathcal{K}]}(\mathcal{D}_2) = \Delta_{\mathcal{D}_2}(\mathbf{K} \xleftarrow{\$} \mathcal{K}; \text{AE}_{\mathbf{K}} \parallel \mathcal{J}^{+t})\,,$$

where all wrap queries of $\mathcal{D}_2$ have a different encryption context.

2.3. SHAKE and TurboSHAKE

EXtendable Output Functions (XOF) are hash functions with an arbitrary-length output. SHAKE128 and SHAKE256 are two XOFs standardized by NIST in [4]. They are defined on top of the KECCAK$[c]$ sponge function. Internally, both use the 24-round permutation KECCAK-$p[1600, n_\text{r} = 24]$ and they are parameterized by the capacity c, 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 $r = b - c$. Here, b is the permutation width in bits and r the (bit) rate, and we denote with $R = r/8$ the rate in bytes. In particular, we have $c = 256$ 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 presenting it to KECCAK$[c]$. In particular,

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

TurboSHAKE is a family of XOFs that was originally introduced for use in KANGAROOTWELVE [7] and later formally defined in [8]. As SHAKE, it is parameterized by the capacity c and is based on the 12-round permutation KECCAK-$p[1600, n_\text{r} = 12]$.

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 $[\texttt{0x01}, \texttt{0x7F}] = [1, 127]$. 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` to the last byte of M'.

2.3.1. Collision resistance. The authors of SHAKE and of TurboSHAKE attached a security claim to their functions, see Claim 1. Informally, it claims that for capacity c, the success probability of any attack against one of these functions is at most $N^2/2^{c+1}$ higher than the same attack against a random oracle, with N the computational complexity expressed as the number of calls to the permutation or equivalent computations. In other words, they 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.

Claim 1 (Flat sponge claim [9])**.** *The expected success probability of any attack against* KECCAK$[r, c]$ *with a workload equivalent to N calls to* KECCAK-$f[r + c]$ *or its inverse shall be smaller than or equal to that for a random oracle plus*

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

We exclude here weaknesses *due to the mere fact that* KECCAK-$f[r + c]$ *can be described compactly and can be efficiently executed, e.g., the so-called random oracle implementation impossibility [9, Section "The impossibility of implementing a random oracle"].*

This security claim is rather informal due to the absence of key. Nevertheless, the claim includes the standard notion of collision resistance as a corollary, and this is one of two security properties that we need from the primitive. (The other one is PRF security of the keyed primitive, see Section 2.3.2.) More specifically, a (Turbo)SHAKE instance F 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 [8]. Quoting FIPS 202 [4]:

> 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.

2.3.2. Multi-user PRF security. As illustrated in Figure 1, our deck function and AE schemes are all built on top of a XOF F and use a secret key that figures as a prefix of the input to F. This presence of a key means we rely on a *keyed* form of the XOF F, that we denote with $F_K(x)$. Here $F_K(x)$ denotes $F(K; x)$ with $a; b$ denoting injective coding of two strings a and b into one string. If we assume these keyed XOFs are hard to distinguish from a random oracle, we can prove the PJC and nPJC security bounds of our AE modes. In particular, we say that F_K is a *pseudo-random function* (PRF) if an adversary cannot distinguish it from a random oracle when the key is randomly and secretly chosen. In this section, we discuss the PRF security of keyed (Turbo)SHAKE, furthermore in the multi-user setting.

For multi-user security, we adopt the formalism of [10, Section 2.1], where the adversary can invoke the scheme with a key selected from an array of μ keys, each of length k bits:

$$\mathbf{K} = (K[0], \ldots, K[\mu - 1]) \in \left(\mathbb{Z}_2^k\right)^\mu\,.$$

Each position in the array corresponds to a pair of communicating users, and the index i of a key $K[i]$ can be viewed as an identifier, just like in the jammin cipher's object identifier, see Section 2.2 and Appendix A. Hence, we will abuse notation and denote a key in the array as $K[\text{ID}]$ for a given identifier ID. There are μ different identifiers and so they can be injectively mapped to $\mathbb{Z}_\mu$.

These keys are sampled according to some distribution $\mathcal{K}$. This distribution is very general: key values do not have to be independent (e.g., if they are drawn without replacement) and they can depend on their identifier.

The key distribution impacts the expression of our bounds via two metrics that we will now discuss. First, we define its *multi-target min-entropy* as

$$H_{\text{mtmin}}(\mathcal{K}) = -\log_2 \max_{K \in \mathbb{Z}_2^k} \Pr(\exists \text{ ID } : K[\text{ID}] = K).$$

Second, we define its *collision entropy* as

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

Definition 3 (PRF security). *Let $F_{\mathbf{K}}$ a collection of μ keyed instances of a XOF, with keys sampled according to distribution $\mathcal{K}$ and $\mathcal{RO}^{\mu}$ a collection of μ different random oracles. The multi-user PRF advantage of $F_{\mathbf{K}}$ with key distribution $\mathcal{K}$ is defined as:*

$$\text{Adv}^{\text{PRF}}_{F[\mathcal{K}]}(N, M, \mu) = \Delta_{N,M}(\mathbf{K} \xleftarrow{\$} \mathcal{K}; F_{\mathbf{K}} \parallel \mathcal{RO}^{\mu}) \,,$$

with the adversarial resources defined as:

- *N: computational complexity, expressed as the number of evaluations of F's underlying permutation or equivalent computations, and*
- *M: data complexity, expressed in total number of input and output blocks in queries to $F_{\mathbf{K}}$,*
- *μ: the number of target keys.*

In the following we will will omit the qualifier multi-user when speaking about multi-user PRF security.

PRF security claims 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 PRF security of (Turbo)SHAKE, see [8] for references. Still, we can prove PRF security bounds for (Turbo)SHAKE if we assume it stands by it security claim.

Theorem 1. *On the condition that (Turbo)SHAKE stands by its claimed security [8], [11], the PRF advantage of keyed (Turbo)SHAKE with key distribution $\mathcal{K}$ is upper bounded as*

$$\text{Adv}^{\text{PRF}}_{F[\mathcal{K}]}(N, M, \mu) \leq \frac{N}{2^{H_{\text{mtmin}}(\mathcal{K})}} + \frac{\binom{\mu}{2}}{2^{H_{\text{coll}}(\mathcal{K})}} + \frac{M^2}{2^{c+1}} \,, \tag{2}$$

where c is the capacity of F and N, M and μ the adversarial resources as defined in Definition 3.

The first term of the righthand side of (2) is due to key guessing, the second is due to key collisions and the third is due to the security claim of (Turbo)SHAKE.

The proof can be found in Appendix B.1.

We will consider two particular distributions:

- $\mathcal{U}$: each key in $\mathbf{K}$ is drawn independently and uniformly from $\mathbb{Z}_2^k$;
- $\mathcal{I}$: each key in $\mathbf{K}$ the concatenation of a independently and uniformly drawn key of k bits and an i-bit representation of a unique ID.

For $\mathcal{U}$, we have that $\Pr(\exists \text{ ID } : K[\text{ID}] = K) \leq \mu \Pr(K[0] = K)$, and $H_{\text{mtmin}}(\mathcal{U}) \geq k - \log_2 \mu$. This shows the degradation of the security by $\log_2 \mu$ bits, and the probability of guessing one of the keys is $\mu N / 2^k$ in this case. Also, the probability of collision between two given keys is 2^{-k}, hence $H_{\text{coll}}(\mathcal{U}) = k$, and among μ keys this becomes $\binom{\mu}{2} 2^{-k}$.

Corollary 1. *Under the same conditions as in Theorem 1, the multi-user PRF advantage of $F_{\mathbf{K}}$ with key distribution $\mathcal{U}$ is upper bounded as*

$$\text{Adv}^{\text{PRF}}_{F[\mathcal{U}]}(N, M, \mu) \leq \frac{\mu N}{2^k} + \frac{\binom{\mu}{2}}{2^k} + \frac{M^2}{2^{c+1}} \,.$$

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

Corollary 2. *Under the same conditions as in Theorem 1, the multi-user PRF advantage of $F_{\mathbf{K}}$ with key distribution $\mathcal{I}$ is upper bounded as*

$$\text{Adv}^{\text{PRF}}_{F[\mathcal{I}]}(N, M, \mu) \leq \frac{N}{2^k} + \frac{M^2}{2^{c+1}} \,.$$

2.4. Byte strings and trailers

In most 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 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*.

So, in the modes in this paper, functions take as input payload byte strings and single-byte *trailers* that encode strings of domain separation bits of length at most 7 bits. We specify constant trailers as an integer value, similarly to the approach taken in the definition of TurboSHAKE's domain separation byte D [8]. For a n-bit trailer $e = (e_0, e_1, \ldots, e_{n-1})$, we define its integer value $E = \text{padint}(e)$, with

$$\text{padint}(e) = 2^n + \sum_{i=0}^{n-1} 2^i e_i \ . \tag{3}$$

For instance, $\text{padint}(\epsilon) = 1$ and $\text{padint}(\texttt{011}) = 14$. The inverse function, $\text{unpad}(E)$, converts an integer $E \geq 1$ to a bit string e. The string 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 '$\texttt{1}$' since $E \geq 1$).

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 unambiguous: Using the padint function works like padding with the pattern $\texttt{10}^*$, where the padding bit '$\texttt{1}$' comes from the 2^n term in Equation 3. In some cases, this padding coincides with padding requirements in lower levels, thereby simplifying layered descriptions.

This allows us to present simple integrated specifications without the layers, as in Figure 4 and Figure 5.

Multiple layers may need to add domain separation bits. A typical case is when the lower layer appends a bit p to a trailer 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, in the sequel we use the term trailer for the short string of bits and its integer encoded in a byte interchangeably and with abuse of notation we will write above expression as $E' = E||p$.

2.5. 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 our algorithm to do that in Algorithm 1.

Algorithm 1 Functions to parse byte strings into block sequences

```
Definition of parse($X, \ell_1, \ell_2$)
Input: Byte string $X$, length $\ell_1$, and length $\ell_2$
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 $\ell_1$ bytes and remaining blocks of $\ell_2$ 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 $\ell_2$.
```

3. 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, in the OD construction the user accesses the sponge function as a stateful object that supports incremental hashing.

OD combines the ideas of the duplex and overwrite constructions, both introduced in [5]. After initialization, an OD object can be called an arbitrary number of times. In a call it takes as input a length-bounded payload byte string and a single-byte trailer and returns a digest of the sequence of inputs received since initialization and it updates its state.

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 underlying sponge function.

3.1. Specification of the OD construction

The OD construction is parameterized with a permutation f, a payload block length ρ (in bytes) and a capacity c (in bits).

An OD object can be created by initializing it or by cloning another OD object. There are two cloning methods, one that clones the full state and another one that discards the first ρ bytes, i.e., it clones only the last $b - 8\rho$ bits.

An OD object supports incremental hashing by means of duplexing calls, where each call takes as input a string B with $|B| \leq \rho$, a trailer $E \in \{1, \ldots, 63\}$ and an index $I \in \mathcal{I} \subseteq \mathbb{Z}$, and returns up to ρ bytes of output. The output of a duplexing call depends on the sequence of strings, trailers and indexes (B_i, E_i, I_i) received so far. The OD object keeps track of how many bytes it returned and the squeezeMore method allows returning more output in between duplexing calls. See Figure 2 for an illustration.

The index is an integer, and its intended purpose is to provide domain separation in addition to the trailers. However, in the scope of this paper, we will only use it with the value $I = 0$; the other values are reserved for future use.

Algorithm 2 defines the OD construction and uses the following conventions. While a duplexing call overwrites the state with the (padded) payload input string, it (bitwise) adds the input trailer after applying to it an encoding function $\text{trailenc}()$. 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 $\text{pad10}^*(B)$ and we have $\text{pad10}^*(B) = B||\texttt{0x01}||\texttt{0x00}^*$. 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 $\text{trailenc}(D)$ with $D = E||1$ in the former case and $D = E||0$ in the latter.

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

```
Parameters: $b$-bit permutation $f$, payload byte rate $\rho$ 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($(B, E, I), \ell$) with $|B| \leq \rho$, $E \in \{1, \ldots, 63\}$, $I \in \mathcal{I}$ and $\ell \leq \rho$
if $|B| = \rho$ then
    Replace the first $\rho$ bytes of $s$ with $B$
    XOR the next bytes of $s$ with $\text{trailenc}(E||1, I)$
else
    Replace the first $\rho$ bytes of $s$ with $\text{pad10}^*(B)$
    XOR the next bytes of $s$ with $\text{trailenc}(E||0, I)$
$s \leftarrow f(s)$
return the first $\ell$ bytes of $s$, then set $o \leftarrow \ell$

Interface OD.squeezeMore($\ell$) with $\ell \leq \rho - o$
return $\ell$ 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 with $(s, o) = (\text{OD}.s, \text{OD}.o)$

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

3.2. 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

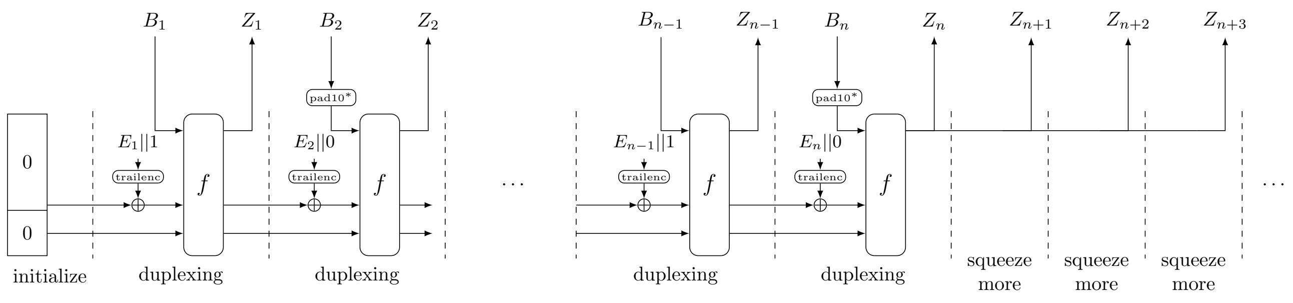


Figure 2. Illustration of the OD construction.

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 define it as

$$\text{trailenc}(D, I) = \text{enc}_{40}(I)||D||\texttt{0x00}||\texttt{0x80}\,,$$

where $\mathcal{I} = \mathbb{Z}_{2^{40}}$ and $\text{enc}_{40}(I)$ converts the index I to a string of 5 bytes in little-endian format.

The format of $\text{trailenc}(D, I)$ is chosen so as to match that of TurboSHAKE's domain separation byte and padding as shown in Section 2.3. In this way, the output of the first duplexing call $\text{OD.duplexing}((B, E, I), \ell)$ equals the output of $\text{TurboSHAKE}[c](B||\text{enc}_{40}(I), E||1)$ or $\text{TurboSHAKE}[c](\text{pad10}^*(B)||\text{enc}_{40}(I), E||0)$ truncated to ℓ bytes. See Lemma 1 for more details.

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

$$\text{SHAKE:}\quad \text{trailenc}(D, I) = \text{enc}_{40}(I)||D||\texttt{0x1F}||\texttt{0x80}\,,$$

since $\text{padint}(\texttt{1111}) = \texttt{0x1F}$.

We now show that an $\text{OD}[f, \rho, c]$ object can be seen as a restricted interface to a (Turbo)SHAKE instance F with permutation f and capacity c. Consequently, $\text{OD}[f, \rho, c]$ inherits, e.g., F's collision resistance and PRF security. This is formalized in Lemma 1.

The output of an OD object to the n-th duplexing call is fully determined by the sequence of inputs $(B_1, E_1, I_1, \ldots, B_n, E_n, I_n)$ that it received in the n duplexing calls $\text{OD.duplexing}((B_i, E_i, I_i), \ell_i)$ since its initialization with $\text{OD.initialize}()$. The output of any intermediate $\text{OD.squeezeMore}(\ell)$ call can be seen as the delayed output of the most recent duplexing calls. Moreover, the output lengths $\ell_i \le \rho$ 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.

Lemma 1. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. The full-block output of the n-th duplexing call to $\text{OD}[f, \rho, c]$ is the ρ-byte output of F applied to an input that is an injective mapping of $(B_1, E_1, I_1, \ldots, B_n, E_n, I_n)$. In addition, the input to F has B_1 as a prefix.*

The proof can be found in Appendix B.2.

Algorithm 3 Definition of $\text{IDAHO}[\mathcal{IN}]$

```
Parameters: input set $\mathcal{IN}$

Interface I.initialize()
Initialize I's attributes path $\leftarrow$ empty and $o \leftarrow \infty$

Interface I.duplexing(in, $\ell$) with in $\in \mathcal{IN}$
path $\leftarrow$ path; in
return the first $\ell$ bytes of $\mathcal{RO}$(path), then set $o \leftarrow \ell$

Interface I.squeezeMore($\ell$) with $o < \infty$
return $\ell$ bytes of $\mathcal{RO}$(path) starting from offset $o$,
then update $o \leftarrow o + \ell$

Interface I.clone()
return a new $\text{IDAHO}[\mathcal{IN}]$ object with $(\text{path}, o)$ =
(I.path, I.$o$)

Interface I.cloneCompact()
return a new $\text{IDAHO}[\mathcal{IN}]$ object with $(\text{path}, o)$ =
(I.path, $\infty$)
```

3.3. Security of keyed OD

We introduce the ideal counterpart of the OD object in Algorithm 3 and call it an *ideal incremental hashing object* (idaho). We define the security of a keyed OD scheme by the advantage of distinguishing a collection of its OD objects from a collection of idaho objects. Furthermore, we prove that this advantage is upper bound by the PRF security of the keyed sponge function underlying the idaho objects.

An idaho object simply remembers the sequence of inputs received so far, called the *path*, and produces outputs by calling a random oracle $\mathcal{RO}$ with the path as input. It takes as a parameter $\mathcal{IN}$, which by default is the set of triplets that OD accepts. It has the same interface as an OD object, except that duplexing calls to an idaho object have no limited input or output lengths.

In the sequel, we will use the OD object with a secret key that is input in the first duplexing call, and denote this as $(\text{OD.duplexing}((K[\text{ID}], 1, 0), 0); \text{OD})$ in Definition 4. Theorem 2 then expresses the PRF security of the OD object in such a case.

Definition 4. *The PRF advantage of a keyed duplex object with key distribution $\mathcal{K}$ is defined as the advantage of distinguishing a collection of keyed duplex objects with key* *distribution $\mathcal{K}$ from a collection of μ independent idaho objects, that is,*

$$\begin{aligned}&\mathrm{Adv}^{\mathrm{PRF}}_{\mathrm{OD}[\mathcal{K}]}(N, M, \mu) \\ &= \Delta_{N,M,\mu}(\mathbf{K} \xleftarrow{\$} \mathcal{K}; (\mathrm{OD.duplexing}(K[\mathrm{ID}], 1, 0); \mathrm{OD})^{\mu} \\ &\qquad \parallel \mathrm{IDAHO}^{\mu})\,,\end{aligned}$$

with N the computational complexity, M the data complexity and μ the number of target keys, and where the adversary makes only queries that would be valid on a OD object (i.e., input and output blocks restricted to ρ bytes).

Theorem 2. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. The PRF advantage of* $\mathrm{OD}[f, \rho, c]$ *with key distribution $\mathcal{K}$ is upper bounded as*

$$\mathrm{Adv}^{PRF}_{\mathrm{OD}[f,\rho,c][\mathcal{K}]}(N, M, \mu) \leq \mathrm{Adv}^{\mathrm{PRF}}_{F[\mathcal{K}]}(N, M, \mu)\,.$$

with resources N, M and μ defined as in Definition 4.

The proof can be found in Appendix B.3.

3.4. Collision resistance of OD

The following theorem expresses the collision resistance of the OD object in terms of the collision resistance of the corresponding sponge function. A collision for an OD object means two different sequences of inputs $S = (B_1, E_1, I_1, \ldots, B_n, E_n, I_n)$ and $S' = (B'_1, E'_1, I'_1 \ldots, B'_m, E'_m, I'_m)$ such that the output of the n-th duplexing call to $\mathrm{OD}[f, \rho, c]$ with S is the same as the m-th duplexing call to $\mathrm{OD}[f, \rho, c]$ with S'.

Theorem 3. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. If an adversary $\mathcal{A}$ outputs a collision for* $\mathrm{OD}[f, \rho, c]$*, then one can efficiently transform it into an adversary $\mathcal{A}'$ that outputs a collision for F.*

The proof can be found in Appendix B.4.

4. The ODWrap mode

In this section, we specify the ODWrap mode that builds a nonce-based authenticated encryption scheme from a sponge function via the OD construction. The schemes are named 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.

4.1. Specification of ODWrap

In Algorithm 4, we specify ODWrap. This mode is a refinement of spongeWrap defined in [5] and is illustrated in Figure 3.

ODWrap objects make use of an underlying OD object. Upon initialization, the underlying OD object is loaded with a secret key K. A wrap call takes as input associated data AD 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 AD 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 ciphertext.

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 length τ.

A wrap call first splits the AD 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 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 τ bytes of that output were already used for tag generation, this block will be at most $\rho - \tau$ bytes long.

4.2. nPJC security of (Turbo)SHAKE-Wrap

(Turbo)SHAKE-Wrap is ODWrap on top of OD applied to keyed (Turbo)SHAKE. We will prove an upper bound on its nPJC advantage $\mathrm{Adv}^{\mathrm{nPJC}}$ (see Definition 2) in terms of the PRF advantage of keyed (Turbo)SHAKE.

When an adversary can start multiple sessions with a ODWrap instance with first wrap calls that have the same associated data AD 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 AD of the first wrap call of all sessions with the same key K to be unique, hence a nonce.

Theorem 4. *Let F be a (Turbo)SHAKE instance, with permutation f and capacity c, that per assumption stands by its claimed security. For a fixed ciphertext expansion t, let there be μ instances of* $\mathrm{ODWrap}[\mathrm{OD}[f, \rho, c], t]$ *(or F-Wrap for short) keyed according to distribution $\mathcal{K}$. Assuming the encryption context is a nonce, then the PJC advantage of F-Wrap is upper bounded as*

$$\begin{aligned}\mathrm{Adv}^{\mathrm{nPJC}}_{F\text{-Wrap}[\mathcal{K}]}(N, M, \mu, q_{\mathrm{forge}}) &\leq \frac{q_{\mathrm{forge}}}{2^t} + \mathrm{Adv}^{\mathrm{PRF}}_{F[\mathcal{K}]}(N, M, \mu) \\ &\leq \frac{q_{\mathrm{forge}}}{2^t} + \frac{N}{2^{H_{\mathrm{mtmin}}(\mathcal{K})}} + \frac{\binom{\mu}{2}}{2^{H_{\mathrm{coll}}(\mathcal{K})}} + \frac{M^2}{2^{c+1}}\,,\end{aligned}$$

with q_{forge} the number of forgery attempts and N, M and μ as in Definition 3.

If k-bit keys are combined with globally unique identifiers, this becomes

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

The proof can be found in Appendix B.5.

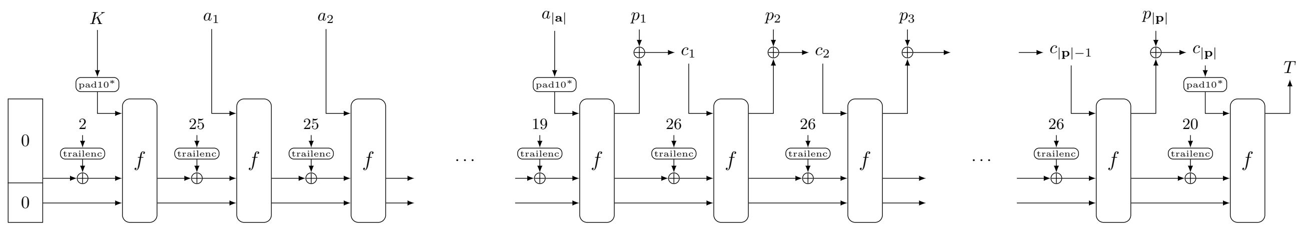


Figure 3. Illustration of the ODWRAP mode merged with the underlying OD construction. This figure shows a first call W.initialize(K) and then W.wrap(AD, P). For the sake of representation, we assume that the last block of each input string has length smaller than ρ and therefore that pad10* is applied.

4.3. Committing security of (Turbo)SHAKE-Wrap

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 $t/2$ bits for a tag length of t bits, as long as $t \leq c$ with c the capacity of the underlying sponge function. Therefore, for tag length $t = 256$, the schemes guarantee a committing security strength of 128 bits. For (Turbo)SHAKE256-Wrap, a tag length of $t = 512$ 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 $t = 160$ for 80 bits of security.

This is expressed more formally in the following theorem.

Theorem 5. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. If an adversary $\mathcal{A}$ outputs a tag collision for* ODWRAP[OD[f, ρ, c], t], *then one can efficiently transform it into an adversary $\mathcal{A}'$ that outputs a collision for F.*

The proof can be found in Appendix B.6.

5. The upperdeck mode

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

5.1. Stateful deck objects

A *doubly-extendable cryptographic keyed*, or *deck*, function is a keyed primitive that natively supports variable input and output lengths [12]. Examples of deck functions are KRAVATTE and XOOFFF, two instances of the Farfalle construction based on the KECCAK-p and XOODOO permutations, respectively [12], [13]. 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. The extendability property of deck functions allows to support sessions in a natural way.

A *deck object* is an object that holds the state that is necessary to evaluate the deck function on an increasing number of input strings. A deck object is equivalent to a deck function but better reflects its implementation and typical usage. For instance, the description of Deck-BO in [6] 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:

$$\begin{aligned} \text{history} &\leftarrow \text{history}, X \\ Y &\leftarrow 0^\ell + D_K(\text{history}) \end{aligned} \tag{4}$$

In a concrete implementation, however, the history is not materialized as an actual sequence of strings. Instead, a deck object keeps state and offers a function that replaces (4).

With a deck object F, the sequence in (4) is replaced with $Y \leftarrow D.\text{absorbAndSqueeze}(X, \ell)$. In addition, we replace bit strings with pairs of byte strings and trailers. So, most generally, the following sequence:

$$\begin{aligned} &D.\text{initialize}(K) \\ &D.\text{absorbAndSqueeze}(X_1, E_1, \ell_1) \\ &\ldots \\ Y \leftarrow &D.\text{absorbAndSqueeze}(X_n, E_n, \ell_n) \end{aligned}$$

is equivalent to

$$Y \leftarrow 0^{\ell_n} + D_K(X_1||\text{unpad}(E_1), \ldots, X_n||\text{unpad}(E_n)) \,.$$

5.2. Specification of upperdeck

In Algorithm 5, we build a deck function on top of OD, using the incremental interface described above. The deck function has an OD object as parameter, that we indicate by 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 consists of a byte string X and a trailer E. 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 length 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 $E' = \text{padint}(0) = 2$ for $i < n$ and $E' = E||1$ for the last block. When the last block

Algorithm 4 Definition of ODWRAP[OD object with ρ, t].

```
Parameters: overwrite duplex object $\text{OD} = \text{OD}[f, \rho, c]$

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

Interface: $C \leftarrow W.\text{wrap}(AD, P)$
$\mathbf{a} \leftarrow \text{parse}(AD, \rho, \rho)$
if $(|AD| > 0)$ AND $(|P| > 0)$ then
  $\mathbf{p} \leftarrow \text{parse}(P, \rho, \rho)$
  for $i = 1$ to $|\mathbf{a}| - 1$ do OD.duplexing($(a_i, 9, 0), 0$)
  $c_1 \leftarrow p_1 + \text{OD.duplexing}((a_{|\mathbf{a}|}, 11, 0), |p_1|)$
  for $i = 2$ to $|\mathbf{p}|$ do
    $c_i \leftarrow p_i + \text{OD.duplexing}((c_{i-1}, 10, 0), |p_i|)$
  $T \leftarrow \text{OD.duplexing}((c_{|\mathbf{p}|}, 12, 0), \tau)$
else if $(|AD| = 0)$ AND $(|P| > 0)$ then
  $\mathbf{p} \leftarrow \text{parse}(P, \rho - \tau, \rho)$
  $c_1 \leftarrow p_1 + \text{OD.squeezeMore}(|p_1|)$
  for $i = 2$ to $|\mathbf{p}|$ do
    $c_i \leftarrow p_i + \text{OD.duplexing}((c_{i-1}, 10, 0), |p_i|)$
  $T \leftarrow \text{OD.duplexing}((c_{|\mathbf{p}|}, 12, 0), \tau)$
else
  for $i = 1$ to $|\mathbf{a}| - 1$ do OD.duplexing($(a_i, 9, 0), 0$)
  $T \leftarrow \text{OD.duplexing}((a_{|\mathbf{a}|}, 13, 0), \tau)$
return $C$, the concatenation of $\mathbf{c}$ (empty if $|P| = 0$) and $T$

Interface: $P \leftarrow W.\text{unwrap}(AD, C)$, may return $\perp$
if $(|C| < \tau)$ then return $\perp$
$\text{OD}' \leftarrow \text{OD.clone()}$
$(C'||T) \leftarrow C$ with $|T| = \tau$
$\mathbf{a} \leftarrow \text{parse}(AD, \rho, \rho)$
if $(|AD| > 0)$ AND $(|C| > \tau)$ then
  $\mathbf{c} \leftarrow \text{parse}(C', \rho, \rho)$
  for $i = 1$ to $|\mathbf{a}| - 1$ do OD.duplexing($(a_i, 9, 0), 0$)
  $p_1 \leftarrow c_1 + \text{OD.duplexing}((a_{|\mathbf{a}|}, 11, 0), |c_1|)$
  for $i = 2$ to $|\mathbf{c}|$ do
    $p_i \leftarrow c_i + \text{OD.duplexing}((c_{i-1}, 10, 0), |c_i|)$
  $T' \leftarrow \text{OD.duplexing}((c_{|\mathbf{c}|}, 12, 0), \tau)$
else if $(|AD| = 0)$ AND $(|C| > \tau)$ then
  $\mathbf{c} \leftarrow \text{parse}(C', \rho - \tau, \rho)$
  $p_1 \leftarrow c_1 + \text{OD.squeezeMore}(|c_1|)$
  for $i = 2$ to $|\mathbf{c}|$ do
    $p_i \leftarrow c_i + \text{OD.duplexing}((c_{i-1}, 10, 0), |c_i|)$
  $T' \leftarrow \text{OD.duplexing}((c_{|\mathbf{c}|}, 12, 0), \tau)$
else
  for $i = 1$ to $|\mathbf{a}| - 1$ do OD.duplexing($(a_i, 9, 0), 0$)
  $T' \leftarrow \text{OD.duplexing}((a_{|\mathbf{a}|}, 13, 0), \tau)$
if $T = T'$ then
  return $P$, the concatenation of $\mathbf{p}$ (empty if $|C| = \tau$)
OD $\leftarrow$ OD$'$
return $\perp$
```

is absorbed, at most ρ bytes are squeezed. More output bits can be obtained via duplexing calls with empty input blocks and trailer $E' = \text{padint}(0) = 2$, although this is done on a cloned state to make the state of the upperdeck object independent of the output length ℓ.

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

Algorithm 5 Definition of UPPERDECK[OD$[f, \rho, c]$]

```
Parameters: overwrite duplex object $\text{OD} = \text{OD}[f, \rho, c]$

Interface: $D$.initialize($K$)
OD.initialize()
OD.duplexing($(K, 1, 0), 0$) // 1 encodes $\epsilon$

Interface: $D$.absorbAndSqueeze($X, E, \ell$) returns $Y$
$\mathbf{x} \leftarrow \text{parse}(X, \rho, \rho)$
for $i = 1$ to $|\mathbf{x}| - 1$ do
  OD.duplexing($(x_i, 2, 0), 0$)
$Y \leftarrow \text{OD.duplexing}((x_{|\mathbf{x}|}, E||1, 0), \min(\ell, \rho))$
if $|Y| < \ell$ then $\text{OD}' \leftarrow \text{OD.cloneCompact()}$
while $|Y| < \ell$ do
  $Y \leftarrow Y||\text{OD}'.\text{duplexing}((\epsilon, 2, 0), \min(\ell - |Y|, \rho))$
return Y

Interface: $D$.clone() returns new upperdeck object $D'$

return $D'$ with $D'.\text{OD} = \text{OD.cloneCompact()}$
```

5.3. PRF security of (Turbo)SHAKE-Upperdeck

In the following theorem, we express that the PRF security of upperdeck can be, again, reduced to that of the underlying (Turbo)SHAKE instance. Note that we speak about "PRF" as we view upperdeck in its original interface.

Theorem 6. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. The multi-user PRF advantage of* UPPERDECK[OD$[f, \rho, c]$] *with key distribution $\mathcal{K}$ is upper bounded as*

$$\text{Adv}^{\text{PRF}}_{\text{UPPERDECK}[\text{OD}[f,\rho,c]][\mathcal{K}]}(N, M, \mu) \leq \text{Adv}^{\text{PRF}}_{F[\mathcal{K}]}(N, M, \mu)\,.$$

The proof can be found in Appendix B.7.

6. The Deck-BO mode

In this section, we specify AE schemes based on the Deck-BO mode, adapted to work on top of an upperdeck object. We prove an upper bound on their (plain) PJC advantage in terms of the PRF advantage of the underlying (Turbo)SHAKE instance. The schemes are named by adding "-BO" to the underlying (Turbo)SHAKE instance name. We first specify the mode and then discuss the PJC advantage and the committing security of the schemes.

6.1. Specification of Deck-BO

Deck-BO is the simplest of the four robust modes presented in [6]. It is based on the Synthetic Initial

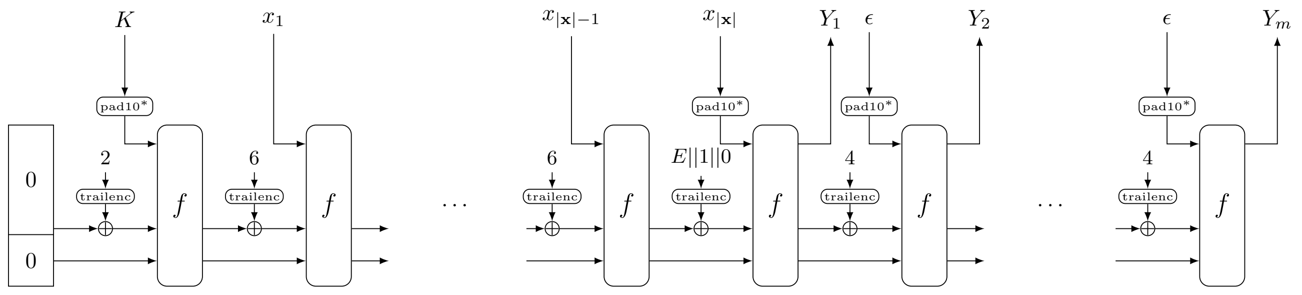


Figure 4. Illustration of the upperdeck mode merged with the underlying OD construction. This figure shows the call to D.initialize(K), followed by F.absorbAndSqueeze(X, E, ℓ) that returns Y. Again, we assume K and the last block of X have size smaller than ρ.

Value (SIV) approach in [14] and supports sessions. In Algorithm 6, we adapt the Deck-BO mode to work on top of the upperdeck construction to support trailers, as required by the upperdeck interface (see Section 2.4). This is illustrated in Figure 5.

An instance of Deck-BO is parameterized with the deck object F and the tag length in bytes τ. Upon initialization, the deck object F is initialized with the key K. A wrap call takes as input (possibly empty) associated data AD 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 AD and P, if non-empty, and by squeezing the first τ bytes of the state. The tag T is thus a pseudorandom function of AD 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 of the deck object is cloned but, taking advantage of the overwrite property of the duplex object underlying the deck object, only $b - 8\rho$ bits of the state must be copied. Upon unwrap, a copy of the state is used to be able to roll back to the original state in case of failure. If the procedure succeeds then the state is updated with the working copy.

Algorithm 6 Definition of Deck-BO adapted to the upperdeck interface

```
Parameters: deck function $F$, expansion length $\tau$

Interface: $Q$.initialize($K$)
$D$.initialize($K$)

Interface: $Q$.wrap($AD, P$) returning $C$
if $|P| = 0$ then
  return  $C \leftarrow F$.absorbAndSqueeze($AD, 4, \tau$)
if $|AD| \neq 0$ then
  $F$.absorbAndSqueeze($AD, 5, 0$)
$F' \leftarrow F$.clone()
$T \leftarrow F$.absorbAndSqueeze($P, 14, \tau$)
$Z \leftarrow P + F'$.absorbAndSqueeze($T, 13, |P|$)
return  $C \leftarrow Z||T$

Interface: $Q$.unwrap($AD, C$) returning $P$ or $\perp$
if $|C| = \tau$ then
  $F' \leftarrow F$.clone()
  $P \leftarrow \epsilon$
  $C' \leftarrow F'$.absorbAndSqueeze($AD, 4, \tau$)
  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 $|AD| \neq 0$ then
    $F'$.absorbAndSqueeze($AD, 5, 0$)
  $F'' \leftarrow F'$.clone()
  $P \leftarrow Z + F''$.absorbAndSqueeze($T, 13, |Z|$)
  $T' \leftarrow F'$.absorbAndSqueeze($P, 14, \tau$)
  if $T' \neq T$ then return $\perp$
else return $\perp$
$F \leftarrow F'$
return  $P$
```

6.2. PJC security of (Turbo)SHAKE-BO

(Turbo)SHAKE-BO is Deck-BO on top of upperdeck on top of OD applied to keyed (Turbo)SHAKE. As the following theorem shows, it is a PJC with its (multi-user) security strength defined by Adv^{PJC}, i.e., the advantage of distinguishing μ keyed instances of it from the jammin cipher, see Definition 2.

Theorem 7. *Let F be a (Turbo)SHAKE instance, with permutation f and capacity c, that per assumption stands by its claimed security. For a fixed ciphertext expansion t, let there be μ instances of (Turbo)SHAKE-BO (or F-BO for short) keyed according to distribution $\mathcal{K}$. Then the PJC advantage of F-BO is upper bounded as*

$$\text{Adv}^{\text{PJC}}_{F\text{-BO}[\mathcal{K}]}(N, M, q_{\text{forge}})$$
$$\leq \frac{q_{\text{forge}}}{2^t} + \sum_{\text{context}} \frac{\binom{\sigma(\text{context})}{2}}{2^t} + \text{Adv}^{\text{PRF}}_{F[\mathcal{K}]}(N, M, \mu)$$
$$\leq \frac{q_{\text{forge}}}{2^t} + \sum_{\text{context}} \frac{\binom{\sigma(\text{context})}{2}}{2^t}$$
$$+ \frac{N}{2^{H_{\text{mtmin}}(\mathcal{K})}} + \frac{\binom{\mu}{2}}{2^{H_{\text{coll}}(\mathcal{K})}} + \frac{M^2}{2^{c+1}},$$

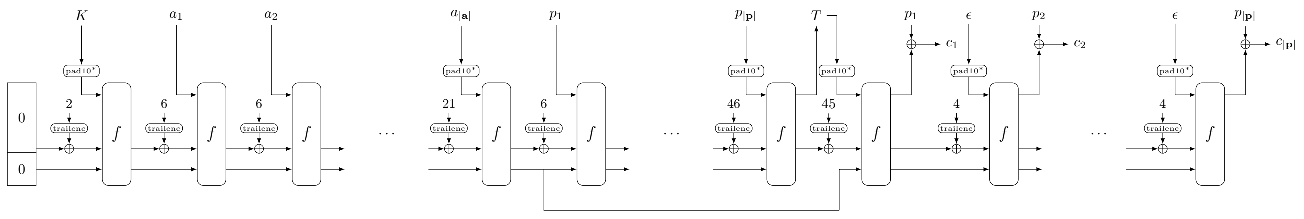


Figure 5. Illustration of the Deck-BO mode merged with the underlying upperdeck mode and the OD construction. This figure shows the call to Q.initialize(K), followed by Q.wrap(AD, P) that returns C. For the sake of visualization, we assume that K, T, and the last blocks of AD and P have length smaller than ρ.

with q_{forge} the number of forgery attempts, $\sigma(\text{context})$ the number of wrap queries with $P \neq \epsilon$ for a given context *value, and N, M and μ as in Definition 3.*

If k-bit keys are combined with globally unique identifiers, this becomes

$$\begin{aligned}\mathrm{Adv}^{\mathrm{PJC}}_{F\text{-BO}[\mathcal{K}]}(N, M, \mu, q_{\text{forge}}) \\ \leq \frac{q_{\text{forge}}}{2^t} + \sum_{\text{context}} \frac{\binom{\sigma(\text{context})}{2}}{2^t} + \frac{N}{2^k} + \frac{M^2}{2^{c+1}}.\end{aligned}$$

The proof can be found in Appendix B.8.

6.3. 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 $t/2$, half the tag length in bits 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.

This is expressed more formally in the following theorem, that can be proved following the same approach used to prove Theorem 5.

Theorem 8. *Let F be a (Turbo)SHAKE instance with permutation f and capacity c. If an adversary $\mathcal{A}$ outputs a tag collision for (Turbo)SHAKE-BO, then one can efficiently transform it into an adversary $\mathcal{A}'$ that outputs a collision for F.*

7. Performance

In this section, we discuss the performance of the different schemes, $\{\text{TurboSHAKE}, \text{SHAKE}\} \times \{128, 256\} \times \{\text{-Wrap}, \text{-BO}\}$. In a first step, we show the performance on an Raspberry Pi 4 equipped with an ARM™ Cortex-A72 processor running at 1.5 GHz. Our goal was to use a popular platform, yet one that does not have any dedicated cryptographic acceleration, to be able to compare relevant algorithms on an equal footing. Then, in a second step, we discuss the performance of our schemes relative to that of hashing with the standard function SHAKE128.

We implemented the algorithms on top of the code provided in XKCP [15]. Table 1 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

TABLE 1. 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 2. 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

of associated data, plaintext or ciphertext. Table 1 also gives the cost of ChaCha20-Poly1305, AES128-GCM and AES256-GCM on the same platform using the implementation in OpenSSL 3.0.2 [16].

On this platform, we can see that all the schemes defined in this paper outperform AES-based ones. Of course, on a platform with dedicated AES support, the picture would be different. ChaCha20-Poly1305 is a particularly efficient alternative that does not rely on hardware acceleration. Yet, TurboSHAKE128-Wrap outperforms it.

As a second step we discuss the cost relative to that of hashing with the standard function SHAKE128. Table 2 evaluates the cost of the different schemes under the assumption that the evaluation of the KECCAK-p permutation dominates, 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 $\rho = 128$ 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, 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.

Table 1 is consistent with Table 2 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.

8. 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 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

[1] 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

[2] 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

[3] 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

[4] 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.

[5] 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

[6] 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

[7] 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

[8] 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

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

[10] 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

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

[12] 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

[13] 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

[14] 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

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

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

[17] 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

[18] 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

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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

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

[30] 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

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

Algorithm 7 The jammin cipher $\mathcal{J}^{\mathrm{WrapExpand}(p)}$

```
1: Parameter: $\mathrm{WrapExpand}$, a $t$-expanding function
2: Global variables: codebook initially set to $\perp$ for all, taboo initially set to empty

3: Instance constructor: init($\mathrm{ID}$)
4: return new instance inst with attribute inst.history $= \mathrm{ID}$

5: Instance cloner: inst.clone()
6: return new instance inst$'$ with the history attribute copied from inst

7: Interface: inst.wrap($AD, P$) returns $C$
8: context $\leftarrow$ inst.history; $AD$
9: if codebook(context; $P$) $= \perp$ then
10:     $\mathcal{C} = \mathbb{Z}_2^{\mathrm{WrapExpand}(|P|)} \setminus$ (codebook(context; $*$) $\cup$ taboo(context))
11:     if $\mathcal{C} = \varnothing$ then return $\perp$
12:     codebook(context; $P$) $\xleftarrow{\$} \mathcal{C}$
13: inst.history $\leftarrow$ inst.history; $AD$; $P$
14: return codebook(context; $P$)

15: Interface: inst.unwrap($AD, C$) returns $P$ or $\perp$
16: context $\leftarrow$ inst.history; $AD$
17: if $\exists! P$ : codebook(context; $P$) $= C$ then
18:     inst.history $\leftarrow$ inst.history; $AD$; $P$
19:     return $P$
20: else
21:     taboo(context) $\leftarrow C$
22:     return $\perp$
```

In Algorithm 7, we recall the definition of the *jammin cipher* [6]. 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 ciphertext C from a message that has a plaintext P and associated data AD, both arbitrary bit strings. With the unwrap function 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 $\mathrm{WrapExpand}(p) = p + t$ 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 $\mathrm{WrapExpand}$ to satisfy this property, defined below.

Definition 5. *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 (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 functions 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.

A.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 6. *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 B. Deferred proofs

B.1. Proof of Theorem 1

Proof. In the distinguishing experiment $\Delta_{\mathcal{D}}(\mathbf{K} \xleftarrow{\$} \mathcal{K}; F_{\mathbf{K}} \parallel \mathcal{RO}^{\mu})$, we can replace the μ random oracles $\mathcal{RO}^{\mu}$ with one that takes the ID as input. The adversary $\mathcal{D}$ is therefore left with the problem of telling $F(K[\mathrm{ID}]||x)$ and $\mathcal{RO}(\mathrm{ID}, x)$ apart, with (ID, x) pairs of its choice.

We now consider a public random oracle $\mathcal{RO}'$ that can be queried offline by the adversary and switch to the problem of telling $\mathcal{RO}'(K[\mathrm{ID}]||x)$ and $\mathcal{RO}(\mathrm{ID}, x)$ apart.

We define two generic bad events and argue that, if they do not occur, $\mathcal{RO}'(K[\mathrm{ID}]||x)$ and $\mathcal{RO}(\mathrm{ID}, x)$ cannot be distinguished. The two bad events are as follows.

- The first is *key guessing*, that is, the event that the adversary queries $\mathcal{RO}'(K||x)$ offline, with K one of the μ keys in $\mathbf{K}$ and some string x. For a given key candidate K^*, the probability that there exists one key in $\mathbf{K}$ with this value is upper bounded by $\leq 2^{-H_{\mathrm{mtmin}}(\mathcal{K})}$. This probability of guessing one of the μ keys correctly after N attemps is at most $N2^{-H_{\mathrm{mtmin}}(\mathcal{K})}$.
- The second is *key collision*, that is, the event that two keys in the array are equal, i.e., $K[\mathrm{ID}] = K[\mathrm{ID}']$ with $\mathrm{ID} \neq \mathrm{ID}'$. The probability of such a collision among the μ keys is at most $\binom{\mu}{2}2^{-H_{\mathrm{coll}}(\mathcal{K})}$.

On the condition that these bad events do not occur, the absence of key collisions implies that the encoding of the ID into the input of $\mathcal{RO}'$ is injective, and the adversary cannot exploit colliding inputs to $\mathcal{RO}'$. Therefore,

$$\Delta_{\mathcal{D}}(\mathbf{K} \xleftarrow{\$} \mathcal{K}; \mathcal{RO}'_{\mathbf{K}} \parallel \mathcal{RO}^{\mu}) \leq \frac{N}{2^{H_{\text{mtmin}}(\mathcal{K})}} + \frac{\binom{\mu}{2}}{2^{H_{\text{coll}}(\mathcal{K})}} .$$

Following the security claim of (Turbo)SHAKE, the original distinguishing problem (with $F_{\mathbf{K}}$) shall not have a success probability greater than that of the latter (with $\mathcal{RO}'_{\mathbf{K}}$) plus the term in (1). Note that distinguishing $F_{\mathbf{K}}$ from $\mathcal{RO}'_{\mathbf{K}}$ can only be done with online queries, so the resource measure in (1) is the online cost M (instead of N). Finally, we upper bound (1) with $M(M+1)/2^{c+1}$ and approximate it as $M^2/2^{c+1}$. □

B.2. Proof of Lemma 1

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 $(B_1, E_1, I_1, \dots, B_n, E_n, I_n)$ 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 $(\beta_1, D_1, I_1, \dots, \beta_n, D_n, I_n)$. More precisely, if $|B_i| < \rho$, $\beta_i \leftarrow \text{pad10}^*(B_i)$ and $D_i = E_i||0$. Otherwise $\beta_i \leftarrow B_i$ and $D_i = E_i||1$. As the last bit of $\text{unpad}(D_i)$ 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 $\overline{\text{OD}}(\beta_1, D_1, I_1, \dots, \beta_n, D_n, I_n)$ the full-block output of OD to the preprocessed input sequence $(\beta_1, D_1, I_1, \dots, \beta_n, D_n, I_n)$.

We first prove the theorem for $n = 1$ by expressing $\overline{\text{OD}}(\beta_1, D_1, I_1)$ as TurboSHAKE128 applied to an input that is an injective mapping of (β_1, D_1, I_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 $\beta_1||\text{enc}_{40}(I_1)||D_1$, in total $\rho + 5$ bytes, that fits in a single $b - c$-bit block. From the TurboSHAKE128 specifications, we see that for a single-block $\overline{\text{OD}}(\beta_1, D_1, I_1) = \text{TS}(\beta_1||\text{enc}_{40}(I_1), D_1)$. Clearly, the mapping from (β_1, D_1, I_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 [5], 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:

$$\begin{aligned}&\overline{\text{OD}}(\beta_1, D_1, I_1, \beta_2, D_2, I_2) = \\ &\quad \text{TS}(\beta_1||\text{trailenc}(D_1, I_1)||(\beta_2 \oplus \overline{\text{OD}}(\beta_1, D_1, I_1)) \\ &\qquad\qquad ||\text{enc}_{40}(I_2), D_2).\end{aligned}$$

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

$$\begin{aligned}O(\beta_1, D_1, I_1, \dots, \beta_n) &= O(\beta_1, D_1, I_1, \dots, \beta_{n-1}) \\ &\quad ||\text{trailenc}(D_{n-1}, I_{n-1}) \\ &\quad ||(\beta_n \oplus \overline{\text{OD}}(\beta_1, \dots, \beta_{n-1}, D_{n-1}, I_{n-1})).\end{aligned}$$

Then,

$$\begin{aligned}&\overline{\text{OD}}(\beta_1, D_1, I_1, \dots, \beta_n, D_n, I_n) \\ &\quad = \text{TS}(O(\beta_1, D_1, \dots, \beta_n)||\text{enc}_{40}(I_n), D_n).\end{aligned}$$

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 $(\beta_1, D_1, I_1, \dots, \beta_{n-1}, D_{n-1}, I_{n-1}) \rightarrow (O(\beta_1, D_1, \dots, \beta_{n-1})||\text{enc}_{40}(I_{n-1}), D_{n-1})$ is injective, then $(\beta_1, D_1, I_1, \dots, \beta_n, D_n, I_n) \rightarrow (O(\beta_1, D_1, I_1, \dots, \beta_n)||\text{enc}_{40}(I_n), D_n)$ 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 $\overline{\text{OD}}(\beta_1, D_1, I_1, \dots, \beta_{n-1}, D_{n-1}, I_{n-1})$ is fixed, and XORing β_n with it preserves the injectivity. □

B.3. Proof of Theorem 2

Proof. Lemma 1 tells us that all the outputs of $\text{OD}[f, \rho, c]$ can be simulated by calls to F with an injective coding. We focus on the case where the OD object is keyed with OD.duplexing$((K[\text{ID}], 1, 0), 0)$ just after initialization, and Lemma 1 tells us also that $K[\text{ID}]$ is a prefix of F's input. Hence, the subsequent outputs of $\text{OD}[f, \rho, c]$ can be simulated by calls to $F_{K[\text{ID}]}$ instead. We can therefore view the keyed OD object hybridly as an idaho object where the random oracle $\mathcal{RO}$ has been replaced with $F_{K[\text{ID}]}$, which we will denote as $\text{IDAHO}[F_{K[\text{ID}]}]$ or collectively as $\text{IDAHO}[F_{\mathbf{K}}]$ for the μ instances. The adversary then has to distinguish $\text{IDAHO}[F_{\mathbf{K}}]$ from IDAHO, which is not easier than distinguishing $F_{\mathbf{K}}$ from μ independent random oracles, and this the multi-user PRF security of F. □

B.4. Proof of Theorem 3

Proof. Let assume that an adversary $\mathcal{A}$ gives two colliding sequences $S \neq S'$. As shown in Lemma 1, there is an injective mapping from a sequence S to a string X such that the output of $\text{OD}[f, \rho, c]$ 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. □

B.5. Proof of Theorem 4

Proof. Thanks to the triangle inequality, the PJC advantage is upper bounded by the sum of two distinguishing advantages:

1) between μ instances of F-Wrap keyed according to $\mathcal{K}$ and μ independent instances of ODWRAP on top of $\text{IDAHO}[\rho]$;

2) between μ instances of ODWRAP on top of IDAHO$[\rho]$, and the jammin cipher,

$$\begin{aligned}&\mathrm{Adv}^{\mathrm{nPJC}}_{F\text{-Wrap}[\mathcal{K}]}(\mathcal{D})\\ &\leq \Delta_{\mathcal{D}'}(\mathbf{K} \xleftarrow{\$} \mathcal{K}; F\text{-Wrap}_{\mathbf{K}} \parallel (\mathrm{ODWRAP}[\mathrm{IDAHO}[\rho], t])^{\mu})\\ &+ \Delta_{\mathcal{D}''}((\mathrm{ODWRAP}[\mathrm{IDAHO}[\rho], t])^{\mu} \parallel \mathcal{J}^{+t}),\end{aligned}$$

where both $\mathcal{D}$ and $\mathcal{D}''$ must ensure that the encryption context is a nonce.

The first term is upper bounded by the multi-user PRF advantage of F. This follows from the fact that an adversary with direct query access to $F_{\mathbf{K}}$ or the RO can simulate any attack by adversary $\mathcal{D}'$ through the ODWrap and idaho layers. So, owing to Theorems 1 and 2, we have

$$\Delta_{\mathcal{D}'}(\dots) \leq \mathrm{Adv}^{\mathrm{PRF}}_{F[\mathcal{K}]}(N, M) \leq \frac{N}{2^{H_{\mathrm{mtmin}}(\mathcal{K})}} + \frac{\binom{\mu}{2}}{2^{H_{\mathrm{coll}}(\mathcal{K})}} + \frac{M^2}{2^{c+1}}$$

The second term is upper bounded by $q_{\mathrm{forge}}/2^t$ 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 AD 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 F-Wrap 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 t bits and all tags are uniformly random, the success probability for each attempt is 2^{-t}. After q_{forge} attempts, this is upper bounded by $q_{\mathrm{forge}}/2^t$.

□

B.6. Proof of Theorem 5

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 $(K, AD_1, P_1, \dots, AD_n, P_n)$ to a sequence of inputs $(B_1, E_1, \dots, B_m, E_m)$ to the underlying OD object is injective. The conclusion then follows from Theorem 3.

We start with $n = 1$. We can injectively map the tuple (K, AD, P) to a sequence of the general form $S = (K, 1, 0), (a_1, 9, 0), \dots, (a_{|\mathbf{a}|}, 11, 0), (c_1, 10, 0), \dots, (c_{|\mathbf{p}|}, 12, 0)$ such that the tag output by ODWRAP$[\mathrm{OD}[f, \rho, c], t]$ is equal to the output of OD$[f, \rho, c]$ after the input sequence S. Here, $\mathbf{a} = \mathrm{parse}(AD, \rho, \rho)$ and $\mathbf{p} = \mathrm{parse}(P, \rho, \rho)$ or $\mathrm{parse}(P, \rho - \tau, \rho)$, 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. □

B.7. Proof of Theorem 6

Proof. The upperdeck object converts the sequence of input strings (i.e., byte strings and trailers) injectively to a sequence of input blocks and trailers it presents to OD. Therefore, we have

$$\mathrm{Adv}^{\mathrm{PRF}}_{\mathrm{UPPERDECK}[\mathrm{OD}[f,\rho,c]][\mathcal{K}]}(N, M) \leq \mathrm{Adv}^{\text{private-idaho}}_{\mathrm{OD}[f,\rho,c][\mathcal{K}]}(N, M).$$

The conclusion follows from Theorem 2. □

B.8. Proof of Theorem 7

Proof. Thanks to the triangle inequality, the PJC advantage is upper bounded by the sum of two distinguishing advantages:

1) between μ instances of F-BO keyed according to $\mathcal{K}$ and μ instances of Deck-BO on top of independent random oracles;
2) between μ instances of Deck-BO on top of independent random oracles and the jammin cipher.

The first term is upper bounded by the advantage of distinguishing μ instances of UPPERDECK$[\mathrm{OD}[f, \rho, c]]$ keyed according to $\mathcal{K}$ from μ different random oracle, and this is covered by Theorem 6.

The second term is covered by the bound for Deck-BO proven in [6, Theorem 3], which is the probability of a successful forgery plus the probability of tags colliding under the same encryption context. In any forgery attempt that the adversary makes, the tag received in the unwrap call is compared with a uniformly random string generated by the underlying random oracle, hence the probability they are equal is 2^{-t}. For q_{forge} forgery attempts, this gives $\frac{q_{\mathrm{forge}}}{2^t}$.

Tag collisions happens with probability $\binom{q}{2}2^{-t}$ for q wrap calls. If we consider at most $\sigma(\text{context})$ wrap queries with the same context, this gives $\sum_{\text{context}} \frac{\binom{\sigma(\text{context})}{2}}{2^t}$. □

Appendix C. Committing AE

Certain settings or applications require AE with committing property, as shown in the following examples. Dodis et al. [17] and Grubbs et al. [2] showed how to exploit non-committing AE schemes in old versions of Facebook's end-to-end encrypted message service. In [18], Albertini et al. study weaknesses of key rotation in key management services, envelope encryption, and "Subscribe with Google" [19], due to the lack of key commitment. They first introduce new theoretical attacks against commonly used AE schemes, such as AES-GCM [20], AES-GCM-SIV [21], [22], ChaCha20-Poly1305 [23], and AES-OCB3 [24], which they turn into practical ones by creating *binary polyglots* (i.e., files which are valid in two different file formats). In [25], 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 [26], Len et al. show

how Shadowsocks proxy servers and the OPAQUE [27] 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* [28]. Later, different definitions have been introduced. Bellare and Hoang [3] and Chan and Rogaway [25] 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. [28] propose to apply a collision-resistant pseudorandom function (PRF) to the entire message or ciphertext, to achieve key commitment. Grubbs et al. [2] presented *compactly committing AE*, requiring a collision-resistant hash function in HMAC mode and a stream cipher such as AES-CTR or ChaCha20. In [18], 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 [25] 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 [3] 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. [17] 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 [18], [29]. 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 [18] and Ascon [30]. In [3], 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.