Parallelizable modes for duplex-based authenticated encryption

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

[1] Radboud University, Nijmegen, The Netherlands
[2] STMicroelectronics, Diegem, Belgium

Abstract

This note enriches the SHAKE and TurboSHAKE-based authenticated encryption schemes defined by Daemen et al. at EuroS&P 2025 with *parallelizable* ones, while keeping their security properties. The main contribution is a new deck cipher mode, called *FlightDeck*, that has the ability of processing input strings similarly to KangarooTwelve and producing output strings as in counter mode, so both parallelizable. We then instantiate FlightDeck on top of the overwrite duplex (OD) construction, in turn on top of a Keccak-p permutation. Combined with a deck cipher-based mode like Deck-BO, the security of the resulting authenticated encryption schemes reduces to the PRF security of keyed (Turbo)SHAKE and to the collision resistance of (Turbo)SHAKE. Compared to the schemes defined at EuroS&P 2025, replacing UpperDeck with FlightDeck greatly improves their software efficiency as one can implement them with Keccak-p permutations evaluated in parallel.

Keywords: Committing authenticated encryption, sponge/duplex construction, duplex cipher, deck cipher, SHA-3, TurboSHAKE

1 Introduction

In [DHM+25], we defined a number of authenticated encryption (AE) schemes based on the Keccak-p permutation and whose security reduces to that of SHAKE or TurboSHAKE. We followed two different approaches. In the first one, the DWrap mode gives AE schemes that directly exploit the efficiency of the duplex construction, where one can absorb and squeeze one block per permutation call. This yields a nonce-based AE scheme that can be described in a relative straightforward way on top of the chosen permutation. The AE schemes following the second approach rely on Deck-BO, a mode that provides nonce-misuse resistance [BDH+22] and that is closely related to the SIV approach by Rogaway and Shrimpton [RS06]. This mode in turn relies on a deck cipher, which is implemented with the UpperDeck construction.

A disadvantage of these schemes is that they can only be implemented in a serial way, that is, calls to the permutation cannot be parallelized. This prevents one from using multiple cores or taking full advantage of the single-instruction-multiple-data (SIMD) instructions of modern processors, such as AVX2, AVX-512 and even ARM's SHA-3 instructions.

Figure 1 illustrates the organization of the different layers, from the permutation to the AE scheme. This note aims at proposing FlightDeck, a parallelizable alternative to

Emails: joan.daemen@ru.nl (Joan Daemen), silvia.mella@ru.nl (Silvia Mella), gilles.vanassche@st.com (Gilles Van Assche)

This work is licensed under a "CC BY 4.0" license.
Date of this document: 2026-11-06.

UpperDeck, on which Deck-BO and potentially other deck cipher-based modes [BDH+22] can rely. Hence, FlightDeck allows one to make the deck-based AE schemes benefit from the parallelism of the platform on which they are implemented.

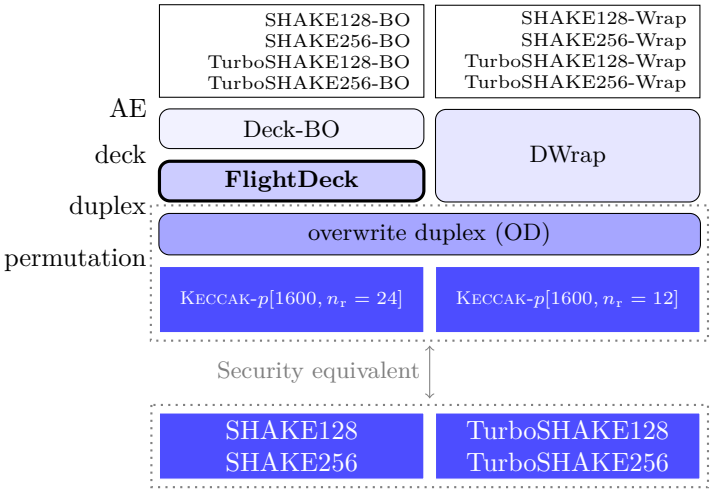

Figure 1: The hierarchy of modes and schemes that we propose. Compared to [DHM+25], we replace UpperDeck with FlightDeck.

This note follows the syntax of stateful primitives as defined in [DHM+25, Section 7.1] and accordingly uses the terminology proposed in [DHM+25, Section 7.3]. For clarity, we briefly recall the central function of deck and duplex ciphers, namely, *duplexing*: It consists of absorbing a string into the state and subsequently squeezing out a string that depends in a complicated way on all the strings absorbed up to that point. We call it a *deck cipher* if the strings can have arbitrary length and a *duplex cipher* if the length of the strings is upper bounded, per definition to the size of a *block*.

2 The FlightDeck mode

In this section, we define a mode to build a deck cipher on top of a duplex cipher, called FlightDeck, and express and prove its security.

The mode FLIGHTDECK[DUPLEX, α, c] is specified in Algorithm 3, using auxiliary Algorithms 1 and 2, and relies on a duplex cipher D of type DUPLEX. In the scope of this note, D is of type overwrite duplex (OD), but we nevertheless define it in full generality. It is also parameterized by the chunk size α in bytes and by the chaining value size c in bits.

In a nutshell, FlightDeck works as follows. The input string is absorbed in a kangaroo-hopping fashion similar to KANGAROOTWELVE [BDP+18, VWV+25]. It is first cut into chunks of α bytes, with α a parameter of the construction. Then, the first chunk is absorbed as is, but starting from the second one, the chunks are processed in parallel. The output string is squeezed in a counter-mode fashion. The first block of ρ bytes is squeezed

Algorithm 1 Definition of parse(X, ℓ_1, ℓ_2) [DHM+25, Algorithm 1]

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

Algorithm 2 Definition of string(n, ℓ)

Input: an integer $n > 0$, granularity ℓ in bytes
Output: a byte string whose length is a multiple of ℓ
Let $N = \lceil \log_{2^{8\ell}} n + 1 \rceil \, \ell$.
Output n as a string of N bytes in little-endian representation, that is, starting with the least significant byte.

from the main deck instance, and then the subsequent blocks are squeezed after duplexing a counter.

When duplexing a string X and its trailer E, the $\mathsf{duplexing}(X, E)$ function processes X by cutting it into blocks of DUPLEX.ρ bytes (denoted ρ in the sequel for simplicity) and calling D.$\mathsf{duplexing}()$ on each of these blocks. This is at least what happens to the first α bytes of X. When the size of X exceeds α bytes, FlightDeck cuts X into chunks of α bytes, $X = x_1||x_2||x_3 \ldots$. The main duplex cipher D processes the first chunk x_1 as explained above, but the subsequent chunks are treated by D$'$, and D only processes their chaining values. More specifically, FlightDeck relies on clones D$'$ of D as it was when starting $\mathsf{duplexing}(X, E)$, and each D$'$ yields the chaining value that D then processes. The same clone is taken for each chunk, so that the different chunks can be absorbed in parallel.

In Algorithm 3, the string R is the concatenation of the first chunk x_1 and of the chaining values, and formally represents what D processes. In practice, the implementation does not have to buffer x_1 but can start processing it immediately. In that case, it is important to save the state of D at the beginning of $\mathsf{duplexing}(X, E)$, and actually only the part that is needed for doing a $\mathsf{cloneCompact}()$.

When requested to produce output data, FlightDeck starts with squeezing out up to ρ bytes from the main duplex cipher D. If more bytes are requested, it follows the counter mode approach: FlightDeck clones D into $\mathrm{D}_{\mathsf{squeeze}}$, makes it absorb a counter and squeezes ρ additional bytes. This can then be repeated as many times as necessary and can also be performed in parallel.

The different types of output are domain separated:

- The chaining values are obtained by duplexing with trailer 4
- The first output block is obtained by duplexing with trailer $E||\mathtt{1}$. Whatever the value of E, this does not collide with the other cases, as we append a bit `1`.
- The next output blocks are obtained by duplexing a counter and trailer 2

The duplexing calls that do not produce output bits use trailer 1, equivalent to the empty string.

2.1 Security of FlightDeck

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

Theorem 1. *The idaho advantage [DHM+25, Definitions 2 and 10] of the FlightDeck mode is upper bounded as*

$$\mathrm{Adv}^{\mathrm{idaho}}_{\textsc{FlightDeck}[\cdot,\alpha,c]}(\sigma) \leq \frac{\rho^2}{\alpha^2} \frac{\sigma^2}{2^{c+1}},$$

with ρ the block length of the underlying duplex cipher and σ the data complexity expressed in number of duplexing queries to the underlying duplex cipher.

Algorithm 3 Definition of FLIGHTDECK[DUPLEX, α, c]

Parameters:
duplex cipher DUPLEX with DUPLEX.ρ, that we denote as ρ here
chunk size α in bytes
chaining value size c in bits

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

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

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

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

Method $Z \leftarrow$ FLIGHTDECK.duplexing$(X, E)[\ell]$
$\mathbf{x} \leftarrow \mathrm{parse}(X, \alpha, \alpha)$
$R \leftarrow x_1$
for $i = 2$ to $|\mathbf{x}|$ **do**
  $\mathbf{y} \leftarrow \mathrm{parse}(x_i, \rho, \rho)$
  $\mathrm{D}' \leftarrow \mathrm{D.cloneCompact}()$
  for $j = 1$ to $|\mathbf{y}| - 1$ **do** D'.duplexing$(y_j, 1)$
  D'.duplexing$(y_{|\mathbf{y}|}, 4)$
  $R \leftarrow R || (0^c + \mathrm{D}'.\mathsf{squeezing}())$
$\mathbf{r} \leftarrow \mathrm{parse}(R, \rho, \rho)$
for $i = 1$ to $|\mathbf{r}| - 1$ **do** D.duplexing$(r_i, 1)$
D.duplexing$(r_{|\mathbf{r}|}, E||\mathtt{1})$
$o \leftarrow 0$
$\mathrm{D}_{\mathsf{squeeze}} \leftarrow \perp$
return $Z \leftarrow 0^{8\ell} +$ squeezing()

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

Proof. Following the definitions in [DHM+25, Definitions 2 and 10], we have that

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

The FlightDeck cipher converts the input string to each duplexing call injectively to a sequence of input blocks and trailers it presents to the underlying duplex cipher Idaho[ρ]. If all the input strings have a length of at most α bytes, the deck mode maps different paths to different paths to its underlying idaho object and equal paths to equal paths. It follows that the two worlds are indistinguishable, in that case.

When the input string is longer than α bytes, the chunks, after the first one, are replaced with chaining values. Each such chaining value is the result of feeding the chunk, injectively, into a clone of the underlying duplex cipher Idaho[ρ], yet in a different domain thanks to the proper use of trailers. If no collision occurs in chaining values, the input string is therefore fed injectively to the underlying duplex cipher, and the two worlds are indistinguishable.

There remains to compute the probability that two chaining values collide. We upper bound the number of chaining values by lower bounding the number of duplexing queries to the underlying deck cipher per chaining value. A chaining value is produced from the input of a chunk that takes $\lceil \frac{\alpha}{\rho} \rceil$ blocks, and the chaining value itself contributes to a duplexing query when processing R. Hence, there are no more than $\frac{\rho\sigma}{\alpha}$ chaining values for a total cost of σ duplexing queries. As the chaining values have c bits, we obtain the upper bound stated in the theorem. □

2.2 (Turbo)SHAKE-FlightDeck

We now define four concrete deck ciphers $\{\text{TurboSHAKE}, \text{SHAKE}\} \times \{128, 256\} \times \{\text{-FlightDeck}\}$ by putting FlightDeck on top of OD on top of a Keccak-p permutation,

$$\textsc{FlightDeck}[\mathrm{OD}[f,\rho,\mathrm{trailenc}],\alpha,c],$$

and setting the four combinations of parameters as follows:

- f is Keccak-$p[1600, n_\mathrm{r} = 12]$ (TurboSHAKE) or Keccak-$p[1600, n_\mathrm{r} = 24]$ (SHAKE),
- ρ is 160 bytes (128-bit secure variant) or 128 bytes (256-bit secure variant),
- α is 5120 bytes (128-bit secure variant) or 8192 bytes (256-bit secure variant), and
- c is 256 bits (128-bit secure variant) or 512 bits (256-bit secure variant).

The performance of the different instances are given in Table 1 and discussed in Section 4. Their collision resistance is formalized in the following theorem.

Theorem 2. *Let F be a (Turbo)SHAKE function with permutation f and capacity c. If an adversary $\mathcal{D}$ outputs a collision for FlightDeck$[OD[f,\rho,\mathrm{trailenc}],\alpha,c]$, then one can efficiently transform it into an adversary $\mathcal{D}'$ that outputs a collision for F.*

Proof. An adversary can find a collision either if D′ processes different chunks and produces identical chaining values, or if D processes different first chunks and/or chaining values. In both cases, the adversary can translate such a collision into a collision in the input of $\mathrm{OD}[f,\rho,\mathrm{trailenc}]$. The conclusion follows from the application of [DHM+25, Theorem 6]. □

3 A new (Turbo)SHAKE-BO

Deck-BO is the simplest of the four robust modes presented in [BDH+22]. It is inspired by the SIV approach in [RS06] and supports sessions. In Algorithm 7 of [DHM+25], the Deck-BO mode is rewritten in terms of a deck cipher with the idaho interface. Its security is expressed in Theorem 3 below.

Theorem 3. *[BDH+22, Theorem 3] The pseudo jammin cipher (PJC) advantage of the* $\textsc{DeckBO}[\cdot,\tau]$ *mode is given by*

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

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

We override the naming convention of [DHM+25] and redefine (Turbo)SHAKE-BO as a family of four AE schemes obtained by putting Deck-BO on top of FlightDeck on top of OD on top of the corresponding Keccak-p permutation with corresponding capacity value, that is,

$$\textsc{DeckBO}[\textsc{FlightDeck}[\mathrm{OD}[f, \rho, \mathrm{trailenc}], \alpha, c], \tau],$$

where, unlike in [DHM+25], FlightDeck is used instead of UpperDeck (see also Figure 1), f, ρ, α and c are as in Section 2.2, and τ is the tag length in bits, chosen by the user depending on the maximum number of queries (see Theorem 3).

The following theorem gives an upper bound on its PJC advantage, see [DHM+25, Definition 7].

Theorem 4. *Let* F *be a (Turbo)SHAKE instance with capacity* c*, that per assumption stands by its claimed security and let*

$$F\text{-BO} = \textsc{DeckBO}[\textsc{FlightDeck}[\mathrm{OD}[f, \rho, \mathrm{trailenc}], \alpha, c], \tau].$$

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

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

with q_{forge} *and* q_{w} *as defined in Theorem 3 and* μ, t *and* σ *as in [DHM+25, Definition 6].*

Proof. Rewriting yields:

$$\mathrm{Adv}^{\mathrm{PJC}}_{\textsc{DeckBO}[\textsc{FlightDeck}[\mathrm{OD}[f,\rho,\mathrm{trailenc}],\alpha,c],\tau][\mathcal{K}]}(\mu, t, \sigma, q_{\mathrm{forge}}, q_{\mathrm{w}}) \leq \mathrm{Adv}^{\mathrm{PJC}}_{\textsc{DeckBO}[\cdot,\tau]}(q_{\mathrm{forge}}, q_{\mathrm{w}}) + \mathrm{Adv}^{\mathrm{idaho}}_{\textsc{FlightDeck}[\cdot,\alpha,c]}(\sigma) + \mathrm{Adv}^{\mathrm{idaho}}_{\mathrm{OD}[f,\rho,\mathrm{trailenc}][\mathcal{K}]}(\mu, t, \sigma).$$

Applying Theorem 3 to the first term, Theorem 1 to the second and [DHM+25, Theorem 5] to the third finishes the proof. □

The committing strength of (Turbo)SHAKE-BO is given by the infeasibility of generating tag collisions. By construction, the tag is computed as the hash of all input data via (Turbo)SHAKE. Therefore, the committing security strength is given by the minimum of $c/2$ and $\tau/2$, half the tag length in bits τ. In (Turbo)SHAKE, $c = 256$ or 512, and if we choose $\tau \geq c$ this yields a committing security strength of 128 and 256 bits, respectively. This is expressed more formally in the following theorem, and it can be proved with the same approach as for [DHM+25, Theorem 8].

Theorem 5. *Let* F *be a (Turbo)SHAKE instance with permutation* f *and capacity* c*. If an adversary* $\mathcal{D}$ *outputs a tag collision for (Turbo)SHAKE-BO, then one can efficiently transform it into an adversary* $\mathcal{D}'$ *that outputs a collision for* F*.*

4 Performance of FlightDeck

In this section, we report briefly on the performance of FlightDeck on its own (see Table 1), as well as that of the new schemes $\{\text{TurboSHAKE}, \text{SHAKE}\} \times \{128, 256\} \times \{\text{-BO}\}$, for which FlightDeck takes the place of UpperDeck (see Tables 2 and 3). Our code is (or soon will be) integrated to the XKCP [VVC26].

Table 1: Performance on an Intel Core i7-7800X CPU (AVX-512) and on an Intel Core i5-6500 (AVX2) with the TurboBoost disabled. All values are in cycles/byte for long inputs and outputs. The "output" column includes the time needed to XOR the output with an input.

	AVX-512		AVX2	
	input	output	input	output
TurboSHAKE128-FlightDeck	0.569	0.497	1.434	1.370
TurboSHAKE256-FlightDeck	0.629	0.602	1.765	1.651
SHAKE128-FlightDeck	1.057	0.957	2.854	2.865
SHAKE256-FlightDeck	1.243	1.188	3.560	3.411

Table 2: Performance on an Intel Core i7-7800X CPU (AVX-512) with the TurboBoost disabled. The "old" and "new" attributes refer to the version with UpperDeck and with FlightDeck, respectively. All values are in cycles/byte for long messages.

	...-Wrap	...-BO (old)		...-BO (new)	
		AD	P or C	AD	P or C
TurboSHAKE128	3.01	2.81	5.81	0.57	1.07
TurboSHAKE256	3.66	3.50	7.15	0.63	1.23
SHAKE128	5.33	5.40	10.70	1.06	2.02
SHAKE256	6.57	6.70	13.26	1.24	2.43
ChaCha20-Poly1305	1.34				
AES128-GCM	0.70				
AES256-GCM	0.94				

Table 3: Performance on an Intel Core i5-6500 (AVX2) with the TurboBoost disabled. The "old" and "new" attributes refer to the version with UpperDeck and with FlightDeck, respectively. All values are in cycles/byte for long messages.

	...-Wrap	...-BO (old)		...-BO (new)	
		AD	P or C	AD	P or C
TurboSHAKE128	3.99	3.71	7.74	1.43	2.80
TurboSHAKE256	4.87	4.57	9.48	1.77	3.42
SHAKE128	7.18	6.91	14.14	2.86	5.72
SHAKE256	8.87	8.57	17.55	3.56	6.96
ChaCha20-Poly1305	1.67				
AES128-GCM	0.70				
AES256-GCM	0.95				

5 Conclusions

In this note, we presented an improvement to the authenticated encryption schemes we defined in [DHM+25]. By allowing parallelism, and specifically the use of SIMD

instructions, we showed a significant performance boost.

As future work, we plan on making the DWrap approach parallelizable. However, a nice advantage of DWrap is its simplicity, and we do not wish to trade it in for parallelism, especially when implementing on smaller platforms that cannot exploit parallelism anyway. So, the strategy we are considering is to generalize DWrap and parameterize it with a chunk size α in a way that it includes both a straightforward serial option when $\alpha = \infty$ (identical to the original DWrap) and a parallelizable one.

Finally, we notice that, by omitting the key or viewing it as a regular input, the FlightDeck algorithm can be transformed into an unkeyed doubly extendable cryptographic (dec) engine [Van18]. A dec engine is the generalization of an extendable output function (XOF), where the input is a sequence of strings, and where a digest of the partial sequence of the string absorbed so far can be requested. "TurboSHAKE-FlightDec" is expected to have a cost similar to that of KangarooTwelve, with the benefits of a parallelizable squeezing phase and with the flexibility of a dec engine.

Acknowledgements

The authors wish to thank Alex Bataille (Université Libre de Bruxelles) for parallelizing the implementation of FlightDeck and writing the code at the basis of the implementation in the XKCP.

References

[BDH+22] Norica Băcuieți, Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. Jammin' on the deck. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 555–584. Springer, 2022. doi:10.1007/978-3-031-22966-4_19.

[BDP+18] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. KangarooTwelve: Fast hashing based on Keccak-p. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 400–418. Springer, 2018. doi:10.1007/978-3-319-93387-0_21.

[DHM+25] Joan Daemen, Seth Hoffert, Silvia Mella, Gilles Van Assche, and Ronny Van Keer. Shaking up authenticated encryption. In *10th IEEE European Symposium on Security and Privacy, EuroS&P 2025, Venice, Italy, June 30 - July 4, 2025*, pages 861–882. IEEE, 2025. doi:10.1109/EuroSP63326.2025.00054.

[RS06] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2006. doi:10.1007/11761679_23.

[Van18] Gilles Van Assche. On dec(k) functions. INDOCRYPT 2018, December 2018. https://keccak.team/files/OnDecAndDeckFunctions-Indocrypt2018.pdf. doi:10.1007/978-3-030-05378-9.

[VVC26] Gilles Van Assche, Ronny Van Keer, and Contributors. Extended Keccak code package, 2026. https://github.com/XKCP/XKCP.

[VWV+25] Benoît Viguier, David Wong, Gilles Van Assche, Quynh Dang, and Joan Daemen. KangarooTwelve and TurboSHAKE. *RFC*, 9861:1–28, 2025. doi:10.17487/RFC9861.