Random-Access AEAD for Fast Lightweight Online Encryption

Andrés Fábrega[1], Julia Len[2], Thomas Ristenpart[3], Gregory Rubin[4]

[1]Cornell University [2]UNC Chapel Hill [3]Cornell Tech [4]Snowflake

Abstract. We study the problem of *random-access authenticated encryption*. In this setting, one wishes to encrypt (resp., decrypt) a large payload in an *online* matter, i.e., using a limited amount of memory, while allowing for the processing of plaintext (resp., ciphertext) segments to be in a random order. Prior work has studied online AE for in-order (streaming) encryption and decryption, and later work added additional constraints to support random access decryption. The result is complicated notions that are not built from the start to account for random access.

We thus provide a new, clean-state treatment to the random-access setting. We introduce *random-access authenticated encryption (raAE) schemes*, which captures AEAD that provides random-access encryption and decryption. We introduce formal security definitions for raAE schemes that cover confidentiality, integrity, and commitment. We prove relationships with existing notions, showing that our simpler treatment does not sacrifice achievable security. Our implications also result in the first treatment of commitment security for online AEAD as well, an increasingly important security goal for AEAD.

We then exercise our formalization with a practice-motivated case study: FIPS-compliant raAE. We introduce an raAE scheme called FLOE (Fast Lightweight Online Encryption) that is FIPS compliant, compatible with existing AES-GCM APIs that mandate random nonces, and yet can provide secure, random-access, committing encryption of orders of magnitude more data than naive approaches that utilize AES-GCM. FLOE was designed in close collaboration with leading cloud data platform Snowflake, where it will soon be used in production to protect sensitive data.

1 Introduction

A ubiquitious problem in large-scale production environments is *online encryption* of large messages (e.g., databases, video streams, etc.), where encryption must be performed using a constant amount of memory. While straightforward when only (unauthenticated) encryption is required—say, by encrypting/decrypting in chunks, using a standard encryption scheme—this is more challenging in the presence of active attackers, who may drop or reorganize ciphertext segments arbitrarily. Moreover, many use cases additionally require both encryption and decryption to be *random access*, meaning that messages (resp., ciphertexts) can be encrypted (resp., decrypted) in any order. These requirements are the starting point for our work in this paper.

Existing treatments of online AE. The problem of online authenticated encryption has received formal academic treatment before (see Section 3). Most recently, this setting was formalized by the work of Hoang et al. [17], who introduce the notion of an *online authenticated encryption* scheme, which encrypts and decrypts large messages by breaking them up into smaller segments that are processed individually. They introduce an accompanying security definition, nOAE, which captures confidentiality and authenticity requirements. Notably, in addition to per-segment guarantees, nOAE also models cross-segment authenticity attacks, such as reordering or dropping segments.

Hoang et al. additionally provide a practical construction, called STREAM, which they prove satisfies nOAE. STREAM has an elegant design: encrypt each segment individually using a nonce-based authenticated encryption scheme AE with (N, i, b) as the nonce, where N is an input nonce, i is the segment number, and b is a bit indicating whether the segment is terminal or not. The inclusion of i and b in the nonce is the simple (yet powerful) design choice that allows STREAM to prevent cross-segment attacks.

While a good starting point, Hoang et al.'s formalism of online AE had an important limitation: it assumed that encryption and decryption are performed in a *streaming* fashion, i.e., that message and ciphertext segments are processed in order. Yet, various real-world use cases require decryption of segments in a random order, such as reading subsets of a large file. Motivated by this, later work by Hoang and Shen [18] extended the syntax of [17] to allow for random-access decryption. They refer to this class of schemes as *canonical* online AE schemes, which they define as a particular type of online AE scheme by adding syntactical constraints to the primitive of [17]. Similarly, they introduce nOAE2, which adapts

nOAE to allow for the decryption oracle to be used in any order, so that nOAE defaults to a special case of this new definition (in-order use of the decryption oracle). They show that STREAM remains secure under nOAE2. Then, as the primary contribution of their work, Hoang and Shen formalize and analyze the security of the streaming encryption scheme in Google's Tink library [3], which was heavily inspired by STREAM, but deviated from it in important ways [18] that required a new security analysis.

In summary, there are two important gaps in our present formal understanding of online AE. First, existing definitions are *complex and unintuitive*; see Section 3. This stems from the fact that random access is defined as an *extension* to the streaming setting, which requires molding the syntax of in-order encryption and decryption in complicated ways to fit the random-access setting. Second, and perhaps more importantly, current treatments do not consider random access *encryption*. This is an important requirement in practice, for instance, to parallelize segment encryptions. Indeed, existing constructions, such as STREAM and Tink, can be used to encrypt segments in a random order, yet existing security definitions do not model this. In particular, as we show in Section 3, nOAE2 does not imply security in the random-access encryption setting. Therefore, for example, parallelized encryption in STREAM and Tink would not benefit from the formal guarantees of nOAE2.

These limitations motivate the need for revisiting the definitional landscape of online authenticated encryption.

New definitions for random access online AE. We begin our work by providing a new formalization for online AE with random-access decryption *and* encryption. One possible starting point would be to continue the trend of prior works, extending canonical AE schemes and nOAE2 security to allow for random-access encryption. However, this would result in an even more complex set of definitions, which further deviate from the intuition and canonical usage of online random-access schemes. We therefore take a clean-slate approach, and introduce *random-access authenticated encryption (raAE) schemes*, a new primitive with random-access encryption and decryption as a first principle. Like online AE schemes, raAE schemes process large messages in terms of individual segments. Yet, our syntax makes a number of refinements to explicitly allow for encryption and decryption of segments in any order (see Section 4).

We then introduce an all-in-one, simulation-based confidentiality and authenticity security definition for raAE schemes, which we call ra-ROR. Our definition is an extension of the standard multi-user real-or-random security definition for AEAD schemes, but on a per-segment basis. In particular, ra-ROR allows the adversary to adaptively interleave encryption and decryption of segments for different messages and ciphertexts, in any order, and across different users. The result is a simple definition, which captures the intuition of how online AE schemes are used in practice.

We formally prove that ra-ROR implies nOAE2, and thus our simplified treatment does not degrade our understanding of security. However, the converse implication does not hold: we show that schemes can be nOAE2-secure but not ra-ROR-secure, and so our definition is strictly stronger than existing ones. In light of this gap, we proceed to show that STREAM does satisfy ra-ROR security, and thus remains secure even if segments are encrypted at random, say, during parallelized encryption.

After revisiting existing notions, we expand the definitional landscape with additional security goals not yet considered in the online setting. We define *context commitment* for raAE schemes, called ra-CMT, which requires that an adversary should not be able to produce two different decryption contexts (key, nonce, and associated data) that successfully decrypt a single ciphertext. An important and (by now) standard security requirement for conventional AEAD schemes, commitment notions are notably absent from the online AE landscape, and thus we address this gap. We analyze existing online AE constructions through the lens of ra-CMT, and show that their security reduces to the commitment security of the underlying AEAD they use.

Practical challenges with existing constructions. At this point, it may seem like ours is a solved problem, since existing constructions like STREAM satisfy ra-ROR security. However, existing schemes face a number of practical challenges that limit their applicability in many settings. We describe these at a high-level below, and discuss them in more detail in Section 6. Our understanding of these limitations was informed by close collaboration with Snowflake, which, requiring online encryption of large files, deemed both STREAM and Tink unsuitable solutions.

First, and most notably, existing schemes *do not comply* with NIST's cryptographic standards. Specifically, Section C.H of FIPS 140-3 [27] allows a limited number of ways for selecting IVs (e.g, 96-bit random nonces), none of which are met by STREAM and Tink. As mentioned earlier, IV selection is critical for the security of these schemes, and thus they cannot be naïvely adapted to comply with NIST's requirements. FIPS compliance is important in many production environments, for example, because customers of an encryption-using product require it.

A second limitation is that, as mentioned above, the commitment security of existing schemes reduces to the commitment security of the underlying AEAD. In particular, use of non-committing AEADs (e.g., AES-GCM, ChaCha20/Poly1305, etc.) would mean losing commitment properties. It would instead be desirable to decouple commitment of the raAE scheme from commitment of the AEAD, thereby enabling the use of generic AEADs while maintaining security.

Third, an inherent requirement of the online setting is encryption of very large messages, which require many AEAD segment encryptions. This can lead to rapid wearout of AEAD keys, i.e., the amount of data that can be safely encrypted. As we unpack in Section 6, existing schemes require key rotations that are more frequent than desirable.

Our scheme: Fast Lightweight Online Encryption (FLOE). This leads to our final contribution: the design of *Fast Lightweight Online Encryption (FLOE)*, a new raAE scheme motivated by practical production requirements. At a high-level, FLOE encrypts each segment individually using an underlying AEAD, with a number of key design choices along the way that address the practical requirements mentioned above. The result is an efficient, FIPS-compliant raAE scheme, with explicit support for random-access encryption and decryption. FLOE was developed in conjunction with Snowflake, where it is slated to be deployed in production, and thus is designed with large-scale, real-world usage in mind.

We prove that FLOE satisfies ra-ROR security, showing both general security bounds that reduce to the security of underlying algorithm choices, and concrete bounds for instantiations that use AES-GCM. Furthermore, we show that FLOE's ra-CMT security is independent of the commitment of the underlying AEAD, thus preserving commitment security irrespective of AEAD choice.

Snowflake has released an open-source reference implementation of FLOE-GCM (i.e., FLOE instantiated with AES-GCM) in various programming languages, which can be found at `https://github.com/Snowflake-Labs/floe-specification`.

2 Preliminaries

Notation. We denote the length of a bitstring X by $|X|$. For a finite set S, we denote by $X \leftarrow\!\!\$\ S$ the process of sampling from S uniformly at random and assigning this value to X. For two strings X and Y, we denote concatenation by $X \| Y$. For a tuple T consisting of t bitstrings, we denote by $(X_1, ..., X_t) \leftarrow T$ parsing T into its individual components. We let $\{0,1\}^n$ be the set of all n-bit strings, 0^n be the n-bit string consisting of all zeroes, and ε the empty string.

Block ciphers. An n-bit block cipher is a deterministic function $\mathsf{E} : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$, which defines a family of permutations. The (multi-user) security of E is defined by game mu-PRP, which consists of oracles New and Eval. The behavior of these oracles is determined by a challenge bit b sampled randomly at the start of the game. The New oracle takes no inputs, and generates new random κ-bit key instances if $b = 1$, or a new random permutation if $b = 0$. The Eval oracle takes as input a key index and n-bit input, and returns the output of E under the corresponding key if $b = 1$, or the output of the random permutation for the key index if $b = 0$.

In this work, we will analyze security based on the *ideal cipher model* [12], where, in game mu-PRP, E is replaced with a family of random permutations, one per key. That is, we replace each n-bit function $\mathsf{E}(K, \cdot)$ with a permutation sampled uniformly at random from the space of all n-bit permutations, denoted by $\mathrm{Perm}(n)$. Moving forward, all security definitions assume that adversaries have oracle access to these permutations and their inverses.

Authenticated encryption. An AEAD scheme AE is a triple of algorithms $\mathsf{AE} = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$. The randomized key generation algorithm $\mathsf{AE.Kg}$ takes no inputs and outputs a key K of a predefined length. The deterministic encryption algorithm $\mathsf{AE.Enc}(K, N, A, M)$ takes as input a key K, nonce N, associated data A, and message M, and outputs a ciphertext C. The deterministic decryption algorithm $\mathsf{AE.Dec}(K, N, A, C)$ takes as input a key, nonce, associated data, and ciphertext, and outputs a message or error symbol $\perp$. Deterministic function $\mathsf{clen}(|M|)$ indicates the ciphertext length for message M. As usual, we require that AEAD schemes are correct, meaning that decryption always reverses encryption: for any K, N, A, M, C, it follows that $\mathsf{Enc}(K, N, A, M) = C \implies \mathsf{Dec}(K, N, A, C) = M$. We will sometimes make use of AEAD schemes that use random nonces, which we denote by $C \leftarrow\!\!\$\ \mathsf{AE.Enc}(K, A, M)$; note that parameter N is omitted in this case. The generated nonce is returned as part of C.

Confidentiality and authenticity: Our security notion for confidentiality and authenticity is *multi-user real-or-random* security, mu-ROR, defined by the games shown in Figure 1. The adversary has

mu-AE-Real$^{\mathcal{A}}_{\mathsf{AE}}$:	mu-AE-Rand$^{\mathcal{A}}$:
$u \leftarrow 0$	$u \leftarrow 0$
$b \leftarrow\!\!\$\ \mathcal{A}^{\text{New},\text{Enc},\text{Dec}}$	$b \leftarrow\!\!\$\ \mathcal{A}^{\text{New},\text{Enc},\text{Dec}}$
return b	**return** b
New :	New :
$u \leftarrow u + 1$	$u \leftarrow u + 1$
$K_u \leftarrow\!\!\$\ \mathsf{AE.Kg}$	**return**
return	
Enc(i, N, A, M) :	Enc(i, N, A, M) :
if $i > u$: **return** $\perp$	**if** $i > u$: **return** $\perp$
return $\mathsf{AE.Enc}(K_i, N, A, M)$	$C \leftarrow\!\!\$\ \{0,1\}^{\mathsf{clen}(\lvert M \rvert)}$
Dec(i, N, A, C) :	**return** C
if $i > u$: **return** $\perp$	Dec(i, N, A, C) :
return $\mathsf{AE.Dec}(K_i, N, A, M)$	**return** $\perp$

Fig. 1. Games for mu-ROR security of AEAD schemes.

CMT$^{\mathcal{A}}_{\mathsf{AE}}$:
$\big(C, (K_1, N_1, A_1), (K_2, N_2, A_2)\big) \leftarrow \mathcal{A}$
$M_1 \leftarrow \Pi.\mathsf{Dec}(K_1, N_1, A_1, C)$; $M_2 \leftarrow \Pi.\mathsf{Dec}(K_2, N_2, A_2, C)$
if $M_1 = \perp$ or $M_2 = \perp$: **return** 0
return $(K_1, N_1, A_1) \neq (K_2, N_2, A_2)$

Fig. 2. Game for CMT security of AEAD schemes.

access to encryption and decryption oracles, which they can call with any key index of their choosing. In the real world, these return ciphertexts and messages computed using the input scheme; and in the ideal world, these return random strings of the appropriate length for encryptions, and always return $\perp$ for decryptions. We denote the randomized variant of mu-ROR by mu-ROR\$, which simply omits parameter N in the encryption oracle, and internally samples the nonce at random instead. We assume that adversaries are nonce-respecting, and so they never reuse nonces for the same user. We define the advantage of mu-ROR adversary $\mathcal{A}$ over AEAD scheme AE as

$$\mathrm{Adv}^{\text{mu-ROR}}_{\mathsf{AE}}(\mathcal{A}) = \left|\Pr\left[\,\text{mu-AE-Real}^{\mathcal{A}}_{\mathsf{AE}} = 1\,\right] - \Pr\left[\,\text{mu-AE-Rand}^{\mathcal{A}}_{\mathsf{AE}} = 1\,\right]\right|$$

and respectively for mu-ROR\$.

Commitment security: We will also rely on the *context commitment* [7,10] of an AEAD, which requires that any ciphertext can only be decrypted under a single decryption context (K, N, A). This property is formalized by security notion CMT, defined by the game shown in Figure 2. Hereafter, whenever we use the broad term "commitment", we are referring to full context commitment, as opposed to weaker commitment variants like (just) key commitment. We define the advantage of CMT adversary $\mathcal{A}$ over AEAD scheme AE as

$$\mathrm{Adv}^{\text{cmt}}_{\mathsf{AE}}(\mathcal{A}) = \Pr\left[\,\text{CMT}^{\mathcal{A}}_{\mathsf{AE}} = 1\,\right].$$

Pseudorandom functions. A pseudorandom function (PRF) F is a deterministic function $F : \{0,1\}^{\kappa} \times \{0,1\}^{i} \to \{0,1\}^{o}$. We define the multi-user security of a PRF, mu-PRF, with the games shown in Figure 3, where the adversary is tasked with differentiating the real F from a random function with the same domain and range as F. We define the advantage of mu-PRF adversary $\mathcal{A}$ over PRF F as

$$\mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{A}) = \left|\Pr\left[\,\text{mu-PRF-Real}^{\mathcal{A}}_{\mathsf{F}} = 1\,\right] - \Pr\left[\,\text{mu-PRF-Rand}^{\mathcal{A}}_{\mathsf{F}} = 1\,\right]\right|.$$

We will also rely on the *collision resistance* CR of F. Simply, the adversary must produce a tuple (K', K'', X', X'') such that $F(K', X') = F(K'', X'')$, subject to the constraints that $(K', X') \neq (K'', X'')$,

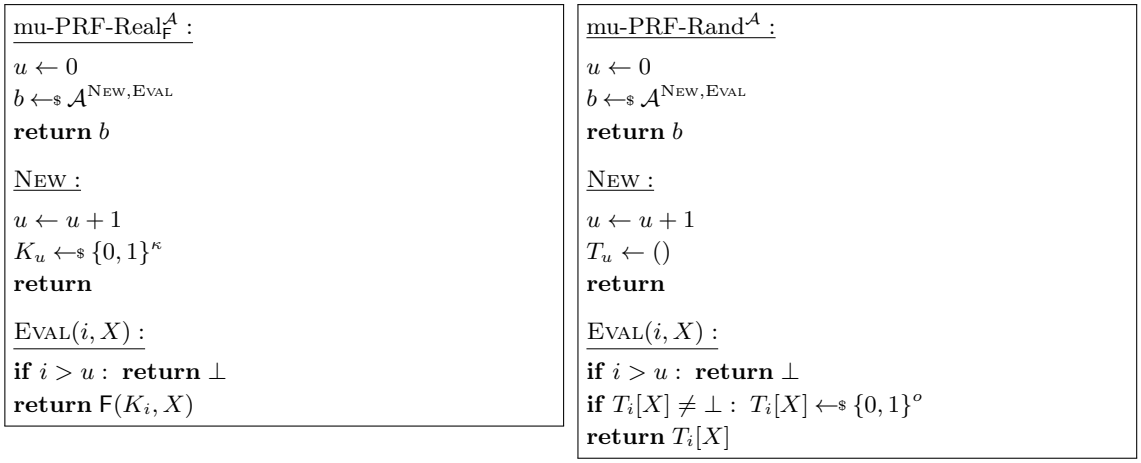

Fig. 3. Games for mu-PRF security of PRFs.

$F(K', X') \neq \perp$, and $F(K'', X'') \neq \perp$. We define the advantage of CR adversary $\mathcal{A}$ over F as

$$\text{Adv}^{\text{cr}}_{\mathsf{F}}(\mathcal{A}) = \left| \Pr\left[\text{CR}^{\mathcal{A}}_{\mathsf{F}} = 1 \right] \right| .$$

We will sometimes be interested in PRFs that support *variable output lengths*. Such a PRF accepts as third parameter some positive integer ℓ, specifying the desired output length. For our purposes, it is sufficient for variable-output length PRFs to be *extendable-output functions (XOFs)* [4], meaning that, for any K, X, ℓ_1, and $\ell_2 \geq \ell_1$, $\mathsf{F}(K, X, \ell_1)$ is a prefix of $\mathsf{F}(K, X, \ell_2)$. In terms of security, for simplicity, we model a variable output-length PRF F as simply truncating the output of a standard PRF F'. That is, F has some maximum allowable output size ℓ_{max}, and is defined by $\mathsf{F}(K, X, \ell) = \mathsf{Prefix}(\mathsf{F}'(K, X), \ell)$, where F' has output space $o = \ell_{\mathsf{max}}$ and $\mathsf{Prefix}(Y, \ell)$ returns the first ℓ bits of Y if $\ell \leq \ell_{\mathsf{max}}$ and $\perp$ otherwise. Notice that this definition implies that F achieves the same security guarantees as F' if and only if F is only queried on a single ℓ for each (K, X); this is different from the stronger security goal of requiring that each output length produces independent outputs [4]. Looking ahead to Section 6, we will never query an input to a variable-output length PRF on more than one output length, and thus this definition is sufficient for our purposes.

3 Online Authenticated Encryption

We begin by recalling the syntax and security definitions for online authenticated encryption with random-access decryption from Hoang et al. [17] and Hoang and Shen [18]. We do so for two reasons: first, to highlight the complexity of the current definitional landscape and, second, to allow formally comparing these prior treatments to our new, simpler one.

Syntax for online AE. An online authenticated encryption (oAE[1]) scheme $\Pi = (\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$ is a triple of algorithms. The randomized key generation algorithm Kg outputs a secret key K from the key space $\mathcal{K}$ associated with Π. The encryption and decryption algorithms differ from those of traditional AEAD schemes in two ways. First, messages, associated data, and ciphertexts consist of *vectors* of bitstrings; full vectors of messages are referred to as *segmented* messages, and the individual bitstrings as message *segments* (and, respectively, for associated data and ciphertexts). Second, encryption and decryption are *stateful* processes. Concretely, the encryption algorithm Enc consists of three subroutines:

- $S \leftarrow \mathsf{Enc.init}(K, N)$: A deterministic algorithm that takes as input a key K and nonce N, and returns a state S. This initiates state for encrypting a segmented message.
- $(C, \tilde{S}) \leftarrow \mathsf{Enc.next}(S, A, M)$: A deterministic algorithm that takes as input a state S, an associated data segment A, and a message segment M, and returns a ciphertext segment C and the updated state $\tilde{S}$.

[1] The shorthand "oAE" for online AE (and, later, "coAE" for canonical online AE) is not part of the definitions from prior work, but we introduce it here for convenience.

- $C \leftarrow \mathsf{Enc.last}(S, A, M)$: A deterministic algorithm that takes as input a state S, an associated data segment A, and a message segment M, and returns a ciphertext segment C. This concludes the encryption of the segmented message.

These three algorithms work in conjunction to encrypt a segmented message. First, Enc.init is called with a key and nonce, followed by Enc.next on every non-terminal message segment. Then, Enc.last is called on the terminal segment. The final ciphertext is composed of the resulting ciphertext segments.

The decryption algorithm Dec consists of three analogous subroutines (Dec.init, Dec.next, Dec.last) with the same interface and behavior as the encryption algorithms (but with ciphertext segments instead of message segments). Notably, when using all three algorithms together, if any call to Dec.next or Dec.last fails and returns $\perp$, the process immediately aborts and returns the message segments that have been decrypted thus far.

The use of separate algorithms for encrypting/decrypting terminal and non-terminal segments may seem surprising. While (confusingly) the reason for this is never stated explicitly in [17,18], this enables preventing *truncation attacks*, where the adversary can trick a recipient into closing the decryption stream before all ciphertext segments have been processed. Attacks of this type have led to vulnerabilities in various other contexts [6]. Thus, this formalization enables schemes to process terminal segments differently (e.g., cryptographically binding a flag indicating that a ciphertext block is terminal).

Online AE with random-access decryption. Hoang and Shen [18] refine the definition of oAE schemes above to allow for *random-access decryption*, meaning that decryption of segments can be performed in any order. They refer to such schemes as *canonical* online AE (coAE) schemes, a subset of the entire space of oAE schemes, which they define by constraining the structure of decryption states: Π is canonical if (1) for any key K and nonce N, we have that $S \leftarrow \mathsf{Dec.init}(K, N)$ returns a state of the form $S = (1, \sigma)$ for some σ; and (2) calling $(M, \tilde{S}) \leftarrow \mathsf{Dec.next}\big((i, \sigma), A, C\big)$ returns a state of the form $\tilde{S} = (i + 1, \sigma)$.

In other words, coAE schemes use S to encode the index of the input ciphertext segment, which guides the behavior of Dec.next and Dec.last accordingly. Also note that, importantly, σ is the same across all states, and so the only variation in S is the ciphertext segment identifier. Therefore, we can decrypt any segment by directly instantiating S with the segment identifier, which is independent of other segment decryption calls.

nOAE2 **security.** Hoang and Shen [18] introduced the nOAE2 notion, which generalizes the nOAE definition [17] for oAE schemes to coAE schemes. We show the security game for nOAE2 in Figure 4 (slightly simplifying the syntax of [18] to an equivalent one). Using the encryption oracles, adversary $\mathcal{A}$ can arbitrarily interleave encryptions of message segments corresponding to different encryption streams under different keys. Note, however, that segment encryptions for a single segmented message must be performed in order. The decryption oracle requires $\mathcal{A}$ to specify a segmented message and associated data as inputs, and the subset and order of segments they want to decrypt (via variable $\mathbf{I}$). Additionally, the adversary must specify whether the input segmented message and associated data are full via input bit a, which indicates whether decryption of the last segment should use the Dec.last algorithm. The advantage of adversary $\mathcal{A}$ over coAE scheme Π is defined by

$$\mathrm{Adv}^{\mathrm{noae2}}_{\Pi}(\mathcal{A}) = \left|\frac{1}{2} - \Pr[\mathrm{nOAE2}^{\mathcal{A}}_{\Pi} = 1]\right|$$

where $\mathcal{A}$ is assumed to be nonce-respecting. Further, $\mathcal{A}$ is prevented from the following behaviors to disallow trivial wins:

(i) Querying $q \leftarrow \textsc{InitEnc}(i, N)$ and $C_k \leftarrow \textsc{NextEnc}(i, q, A_k, M_k)$ for $k \in [1, m]$ (for some positive integer m), followed by $\textsc{Dec}(i, N, \mathbf{A}, \mathbf{C}, \mathbf{I}, 0)$ where $\mathbf{A}[j] = A_j$ and $\mathbf{C}[j] = C_j$ for every $j \in \mathbf{I}$.

(ii) Querying $q \leftarrow \textsc{InitEnc}(i, N)$ and $C_k \leftarrow \textsc{NextEnc}(i, q, A_k, M_k)$ for $k \in [1, m]$ (for some positive integer m), followed by $\textsc{Dec}(i, N, \mathbf{A}, \mathbf{C}, \mathbf{I}, 1)$ where $|\mathbf{C}| \notin \mathbf{I}$, and $\mathbf{A}[j] = A_j$ and $\mathbf{C}[j] = C_j$ for every $j \in \mathbf{I}$.

(iii) Querying $q \leftarrow \textsc{InitEnc}(i, N)$, $C_k \leftarrow \textsc{NextEnc}(i, q, A_k, M_k)$ for $k \in [1, m-1]$ (for some positive integer m), and $C_m \leftarrow \textsc{LastEnc}(i, q, A_m, M_m)$, followed by $\textsc{Dec}(i, N, \mathbf{A}, \mathbf{C}, \mathbf{I}, 1)$ where $\mathbf{A}[j] = A_j$ and $\mathbf{C}[j] = C_j$ for every $j \in \mathbf{I}$.

Intuitively, the conditions above prevent $\mathcal{A}$ from computing Dec.next (resp., Dec.last) with a ciphertext obtained from Enc.next (resp., Enc.last) and with the same parameters as that corresponding encryption query. Concretely, condition (i) disallows decrypting segmented ciphertexts where each segment (as specified by $\mathbf{I}$) is the output of some previous NextEnc call, and in the same position as that NextEnc call.

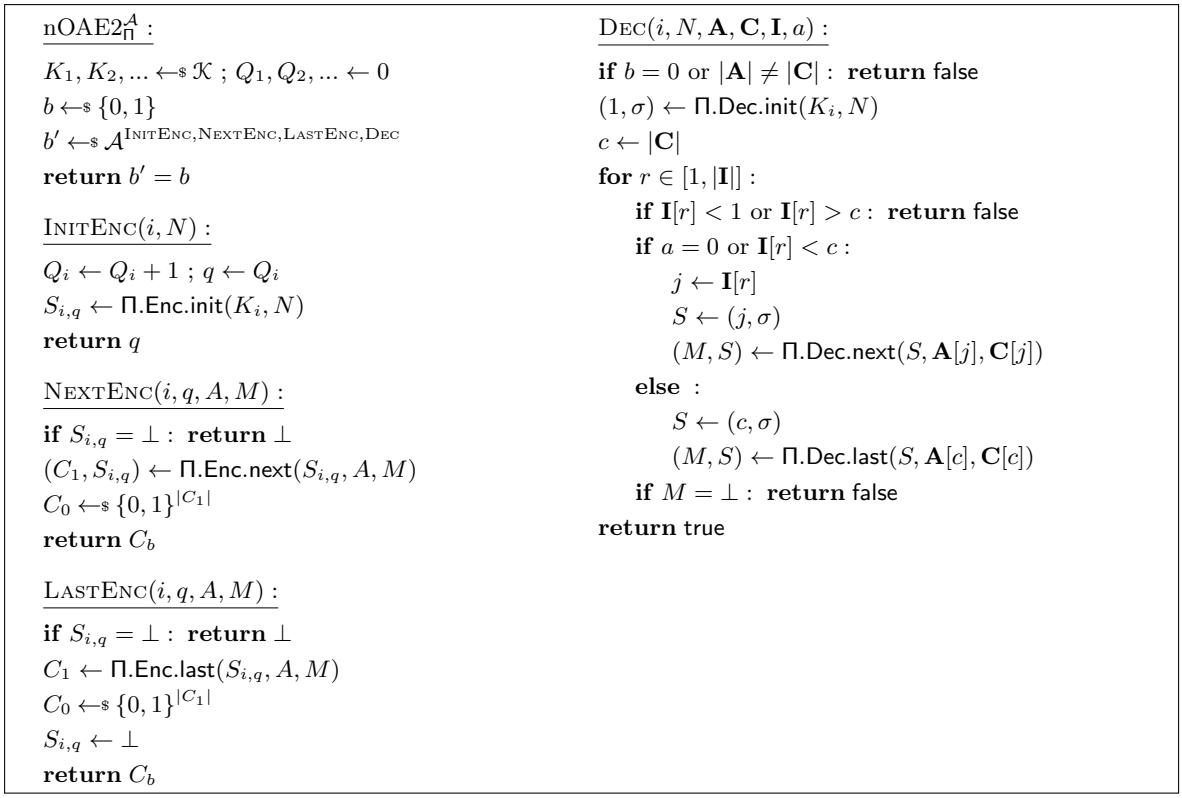

Fig. 4. Game for nOAE2 security of canonical online AE schemes.

For example, say that $\mathcal{A}$ calls $C_1 \leftarrow \text{NextEnc}(1, 1, A_k, M_k)$ followed by $C_2 \leftarrow \text{NextEnc}(1, 2, A_k, M_k)$. Then, $\mathbf{C} = (C_1, C_2)$ with $\mathbf{I} = (1)$ would not be allowed, but $\mathbf{C} = (C_2, C_1)$ with $\mathbf{I} = (1)$ would: when $b = 1$, the former would call $\Pi.\mathsf{Dec.next}$ with input $((1, \sigma), A_1, C_1)$, and the latter with input $((1, \sigma), A_2, C_2)$. Condition (ii) is essentially the same as condition (i) but for the case when $\mathcal{A}$ sets $a = 1$ without including the ciphertext's terminal segment number in $\mathbf{I}$ (which would thus be ignored, degrading to condition (i)). Lastly, condition (iii) is the same as condition (i), but extending the constraints to terminal segments.

STREAM and SE1 constructions. The most promiment coAE scheme is the STREAM construction of Hoang et al. [17], shown in Figure 5. At a high-level, STREAM encrypts each message segment independently using a standard AEAD AE (a parameter to the scheme) with key K and nonce (N, i, b), where i is the segment number and b is a bit indicating whether the segment is terminal or not. The nOAE2 security of STREAM was analyzed in [18], showing a direct reduction to the multi-user real-or-random security of AE.

Hoang and Shen [18] later introduce coAE scheme SE1, shown in Figure 6, which formalizes the streaming encryption scheme of Google's Tink encryption library [3]. Tink's scheme was inspired by STREAM, but, due to usability issues, introduced several changes. At a high-level, like STREAM, SE1 encrypts each message segment individually using AE. However, instead of using the key K directly, the encryption key is derived using a KDF F applied to K and a string R that is part of the input nonce $N = (P, R)$. Nonces for the underlying AEAD calls are of the form (P, i, b), where i and b are defined as in STREAM. The nOAE2 security of SE1, analyzed in [18], reduces to the multi-user real-or-random security of AE and the PRF security of F.

Tink's streaming encryption scheme can be cast as a particular instantiation of SE1, where F is HMAC-SHA256 and AE can be either AES-GCM or AES-CRT-HMAC. Furthermore, Tink does not support per-segment associated data, and instead lets users specify a header H at the start of encryption, which can be modelled as SE1 when A must always be the empty string. Lastly, Tink does not allow users to specify nonces, which are instead internally computed as the concatenation of a random 7-byte

$\underline{\mathsf{Enc.init}(K, N)}$:	$\underline{\mathsf{Dec.init}(K, N)}$:
return $(1, K, N)$	**return** $(1, K, N)$
$\underline{\mathsf{Enc.next}(S, A, M)}$:	$\underline{\mathsf{Dec.next}(S, A, C)}$:
$(i, K, N) \leftarrow S$	$(i, K, N) \leftarrow S$
$S \leftarrow (i + 1, K, N)$	$S \leftarrow (i + 1, K, N)$
$C \leftarrow \mathsf{AE.Enc}\big(K, (N, i, 0), A, M\big)$	$M \leftarrow \mathsf{AE.Dec}\big(K, (N, i, 0), A, C\big)$
return (C, S)	**return** (M, S)
$\underline{\mathsf{Enc.last}(S, A, M)}$:	$\underline{\mathsf{Dec.last}(S, A, C)}$:
$(i, K, N) \leftarrow S$	$(i, K, N) \leftarrow S$
$C \leftarrow \mathsf{AE.Enc}\big(K, (N, i, 1), A, M\big)$	$M \leftarrow \mathsf{AE.Dec}\big(K, (N, i, 1), A, C\big)$
return C	**return** M

Fig. 5. Specification of coAE scheme $\mathsf{STREAM}[\mathsf{AE}]$ from Hoang et al [17].

$\underline{\mathsf{Enc.init}(K, N)}$:	$\underline{\mathsf{Dec.init}(K, N)}$:
$(R, P) \leftarrow N$	$(R, P) \leftarrow N$
$\tilde{K} \leftarrow \mathsf{F}(K, R)$	$\tilde{K} \leftarrow \mathsf{F}(K, R)$
return $(1, \tilde{K}, N)$	**return** $(1, \tilde{K}, N)$
$\underline{\mathsf{Enc.next}(S, A, M)}$:	$\underline{\mathsf{Dec.next}(S, A, C)}$:
$(i, \tilde{K}, N) \leftarrow S$; $(R, P) \leftarrow N$	$(i, \tilde{K}, N) \leftarrow S$; $(R, P) \leftarrow N$
$C \leftarrow \mathsf{AE.Enc}\big(\tilde{K}, (P, i, 0), A, M\big)$	$M \leftarrow \mathsf{AE.Dec}\big(\tilde{K}, (P, i, 0), A, C\big)$
$S \leftarrow (i + 1, \tilde{K}, N)$	**if** $M = \perp$: **return** $(\perp, \perp)$
return (C, S)	$S \leftarrow (i + 1, \tilde{K}, N)$
$\underline{\mathsf{Enc.last}(S, A, M)}$:	**return** (M, S)
$(i, \tilde{K}, N) \leftarrow S$; $(R, P) \leftarrow N$	$\underline{\mathsf{Dec.last}(S, A, C)}$:
$C \leftarrow \mathsf{AE.Enc}\big(\tilde{K}, (P, i, 1), A, M\big)$	$(i, \tilde{K}, N) \leftarrow S$; $(R, P) \leftarrow N$
return C	$M \leftarrow \mathsf{AE.Dec}\big(\tilde{K}, (P, i, 1), A, C\big)$
	return M

Fig. 6. Specification of coAE scheme $\mathsf{SE1}[\mathsf{F}, \mathsf{AE}]$ from Hoang and Shen [18], which closely resembles STREAM [17], and formalizes the streaming encryption scheme of Google's Tink library [3].

prefix P, a random 16-byte salt S, and the user-specified header H. Through the framework of $\mathsf{SE1}$, this corresponds to $N = (P, R)$ where $R = S \parallel H$.

Towards random access encryption. A glaring limitation of the current treatment of online AE is that it only supports random-access decryption (i.e., coAE schemes), while ignoring random-access *encryption*. Yet, in practice, many applications demand random-access encryption as well, most notably, to parallelize encryption of large messages. It is thus important to extend our formal treatment of online AE to account for this setting.

A natural starting point would be to mirror the extensions from oAE to coAE, which added support for random-access decryption. That is, to define a new class of coAE schemes, call it dcoAE (*doubly* canonical oAE), which restricts encryption states to be of the form (i, σ), where i is the segment number, matching the structure of decryption states. However, this would require us to define yet another extension to nOAE2, since this definition is fundamentally sequential in encryption, which does not match how real-world adversaries would observe ciphertexts. In particular, this added flexibility raises new security challenges not captured by nOAE2 (which, recall, only allows encrypting message segments in order).

To make this concrete, say we adapt nOAE2 into a new definition, call it nOAE3, that replaces the InitEnc, NextEnc and LastEnc oracles with a single Enc oracle, analogous to the complex Dec oracle, thereby allowing random-access encryption. (This mirrors the extensions from nOAE to nOAE2, which replaced the three decryption oracles with one that allows random-access decryption.) Then, consider the toy dcoAE scheme shown in Figure 7. It is similar to STREAM, but adds a *backdoor*

$\underline{\mathsf{Enc.init}(K, N):}$
$K_1^* \parallel K_2^* \parallel N^* \leftarrow \mathsf{F}(K, N)$
return $(1, K_1^*, K_2^*, N^*)$

$\underline{\mathsf{Enc.next}(S, A, M):}$
$(i, K_1^*, K_2^*, N^*) \leftarrow S$
$S \leftarrow (i+1, K_1^*, K_2^*, N^*)$
if $M = F(K_1^*, i+1)$:
  $R \leftarrow 0^{|M|}$
else :
  $R \leftarrow\!\!\$\ \{0,1\}^{|M|}$
$B \leftarrow F(K_1^*, i) \parallel R$
$C \leftarrow B \parallel \mathsf{AE.Enc}(K_2^*, (N^*, i, 0), (R, i), M)$
return (C, S)

$\underline{\mathsf{Dec.init}(K, N):}$
$K_1^* \parallel K_2^* \parallel N^* \leftarrow \mathsf{F}(K, N)$
return $(1, K_1^*, K_2^*, N^*)$

$\underline{\mathsf{Dec.next}(S, A, C):}$
$(i, K_1^*, K_2^*, N^*) \leftarrow S$
$S \leftarrow (i+1, K_1^*, K_2^*, N^*)$
$(H, R, C^*) \leftarrow C$
$M \leftarrow \mathsf{AE.Dec}(K_2^*, (N^*, i, 0), (R, i), C^*)$
return (M, S)

Fig. 7. coAE scheme that is nOAE2 secure, but not secure if adversaries can encrypt segments in random order. Functions F and AE are a PRF and AEAD, respectively. Algorithms Enc.last and Dec.last are analogous to their next counterparts, but without returning a state, and setting the terminal bit to 1.

by prepending $\mathsf{F}(K_1^*, i) \parallel R$ to the ciphertext, where F is a PRF; key K_1^* is generated during encryption start, and is not known by the adversary; and string R is equal to the all-zero string if $M = F(K_1^*, i+1)$, and random otherwise. Therefore, finding an M that satisfies this relationship would allow distinguishing real encryptions (i.e., containing the all-zero string) from random ones.

The key idea is that triggering this backdoor requires encrypting position $i+1$ before position i. In nOAE2, however, since encryption is in order, the adversary cannot learn $F(K_1^*, i+1)$ until *after* it has encrypted the i-th segment, and so the backdoor is not useful, meaning that the scheme is equivalent to STREAM, which is nOAE2-secure. It may seem tempting to suggest that the adversary can (1) jump ahead to the $i+1$ position by making i "dummy" encryption calls, which set the correct $i+1$ state for the next segment encryption; (2) perform the $i+1$-th encryption call; and (3) go back and re-encrypt from the first until the i-th segment as usual, leveraging the backdoor. Critically, however, this would require starting a fresh encryption stream between steps (2) and (3). Since the adversary is nonce-respecting, K_1^* would be different in both encryption streams, and thus the $F(K_1^*, i+1)$ from step (2) would not trigger the backdoor in step (3). Conversely, in nOAE3, since the adversary can encrypt segments in any order, they can simply request an encryption of $\mathsf{Enc.next}((i+1, \sigma), \varepsilon, M)$ for any M to learn $F(K_1^*, i+1)$, followed by an encryption of $\mathsf{Enc.next}((i, \sigma), \varepsilon, F(K_1^*, i+1))$, which clearly breaks security.

As this example shows, existing definitions do not imply security when random-access encryption is allowed—similar to how nOAE2 is not implied by nOAE [18]—and thus extensions to the definitional landscape are required.

4 New Definitions for Random-Access AEAD

We provide a new AEAD formalization for random access in both encryption and decryption. One approach would be to continue the progression discussed at the end of the last section: defining dcoAE schemes as coAE schemes with restrictions on encryption states; and defining nOAE3 as an extension to nOAE2 with an oracle for random-access encryption and an expanded set of trivial win conditions. However, the result would be an even more complex definitional landscape, impeding their utility. The root cause of this complexity is that the original formalism of oAE assumed in-order encryption and decryption, so syntactical complexity is required to mold this rigid syntax to support random access.

Instead, we take a clean-slate approach—motivated by random-access encryption and decryption from the start—that achieves simpler definitions while simultaneously supporting the features required by our motivating applications.

Random-access AEAD. We introduce random-access AEAD, a new primitive with syntax directly motivated by random access encryption and decryption. We then introduce a simulation-based, all-in-one security definition for confidentiality and authenticity, followed by a new commitment security notion.

Formally, a *random-access authenticated encryption (raAE) scheme* consists of a tuple of five algorithms $\Pi = (\mathsf{Kg}, \mathsf{StartEnc}, \mathsf{EncSeg}, \mathsf{StartDec}, \mathsf{DecSeg})$, defined as follows:

- $K \xleftarrow{\$} \Pi.\mathsf{Kg}$: A randomized algorithm that takes no input, and outputs a secret key K.
- $(T_g, S) \leftarrow \Pi.\mathsf{StartEnc}(K, N, G)$: A deterministic algorithm that takes as input a key K, nonce N, and global associated data G, and outputs a ciphertext header T_g and initial state S.
- $C_i \xleftarrow{\$} \Pi.\mathsf{EncSeg}(S, p, A_i, M_i)$: A randomized algorithm that takes as input an encryption state S, a position identifier p, associated data segment A_i, and message segment M_i. It returns a ciphertext segment C_i.
- $S \leftarrow \Pi.\mathsf{StartDec}(K, N, G, T_g)$: A deterministic algorithm that takes as input a key K, nonce N, global associated data G, and a ciphertext header T_g. It outputs either a state S or an error symbol $\perp$.
- $M_i \leftarrow \Pi.\mathsf{DecSeg}(S, p, A_i, C_i)$: A deterministic algorithm that takes as input a decryption state S, position identifier p, associated data segment A_i, and a ciphertext segment C_i. It outputs message segment M_i or error symbol $\perp$.

We also define $\Pi.\mathsf{Hdrs}$ as the set of all possible header strings (e.g., $\{0,1\}^{256}$) and $\Pi.\mathsf{Ctxts}(p, \ell)$ to be a function that maps a position indicator p and a message length ℓ to a set of all possible ciphertexts for messages of that length and position.

Like coAE schemes, raAE schemes operate over segmented messages and segmented associated data of the form $\mathbf{M} = (M_1, ..., M_n)$ and $\mathbf{A} = (A_1, ..., A_n)$. We similarly have a dedicated algorithm, $\mathsf{StartEnc}$, to initiate an encryption stream. Our syntax here has two important differences from that of coAE schemes. First, $\mathsf{StartEnc}$ accepts an (optional) additional parameter, which is a "global" associated data G. Second, $\mathsf{StartEnc}$ may output a header T_g alongside the state. This header is included as part of the ciphertext (see below for further discussion of headers), and so resulting ciphertexts have the form $\mathbf{C} = T_g, (C_1, ..., C_n)$. We say that a raAE scheme is *headerless* if header T_g is omitted from outputs and inputs. Equivalently, one can think of it as a scheme for which $\Pi.\mathsf{Hdrs} = \{\varepsilon\}$, and thus T_g is always ε.

Discussion: position indicators. There are notable differences between our segment encryption/decryption algorithms and those of prior work. Most notably, coAE schemes rely on separate segment encryption/decryption routines for terminal and non-terminal segments. Instead, our raAE formalization surfaces explicitly segment position as an input to EncSeg and DecSeg. A segment position identifier p captures context about the location of an input segment. The specific format of position identifiers will not be important to our correctness or security definitions: it suffices that they provide a total ordering of segments and somehow flag when a segment is terminal. The latter is important for security against truncation attacks. For simplicity, we will assume that p is a tuple of the form (i, b), where $i \in \mathbb{N}$ is a segment number and $b \in \{0, 1\}$ a terminal bit indicator.

Our approach of including explicit position indicators is key to a simpler formalization compared to coAE. The previous formalization had to explicitly model how a syntax designed for sequential decryption could be rejiggered to allow random access. This required modeling explicitly specification of decryption orderings as an API input (e.g., to security definition oracles). In our approach, we instead externalize from the formal model how applications treat groups of segments, giving applications instead the ability to directly engage with encryption and decryption of individual segments. At the same time, position identifiers allow applications to take advantage of security guarantees related to position and the total number of segments. As we will see, this increases flexibility, e.g., allowing for sequential encryption of a complete set of segments, an arbitrary order of encryption or decryption of segments, or even partial encryption or decryption (not all segments end up processed).

Discussion: immutable state. Another notable design choice for our syntax is that we have state S be fixed: segment encryption and decryption do not update it. This choice reflects our random-access setting, where encryption and decryption processes of large plaintexts or ciphertexts, respectively, should be fully parallelizable with no dependencies between different segments. In-use coAE schemes are fully parallelizable (see previous section); applications demand it. At the same time, our formalization allows cryptographically binding position to segment, ensuring we do not lose anything in terms of security.

Discussion: ciphertext headers. Our formalization includes explicit ciphertext headers generated by $\mathsf{StartEnc}$ and validated by $\mathsf{StartDec}$. Looking ahead, we will use these for committing to the key, nonce, and global associated data. This reflects within the formalization an attractive design pattern, in which applications store a cryptographically verifiable ciphertext header in some untrusted location (e.g., cloud storage), and use it to validate the ciphertext-wide decryption context (key, nonce, and global associated

data). As we will see in our case study, our raAE formalization easily supports specification and analysis of such designs. Relatedly, we note that per-segment associated data was the only type supported by coAE formalizations. We additionally allow a global associated data, which proves useful in applications and is supported efficiently using headers.

Furthermore, we consider headerless schemes that omit headers. As such, our confidentiality and integrity security goals will come in two flavors, one for schemes with headers and one for headerless. Headers provide an extra integrity check on decryption contexts that headerless schemes do not support.

Correctness. As usual, we require correctness for raAE schemes. Correctness in our setting is more nuanced than that of traditional AEAD schemes, since it needs to account for encryptions and decryptions of all segments simultaneously, which may occur in any order. However, an important observation is that, since decryption is random access, it suffices to consider *correctness of decrypting individual ciphertext segments*, which together imply correctness of decrypting the entire segmented ciphertext, irrespective of decryption order. Essentially, if correctness of one segment decryption depends on the correctness of another segment decryption, then the scheme cannot be random access, as the former cannot be decrypted before the latter.

We now formally define correctness for raAE schemes, based on this insight. Intuitively, a scheme is correct if decryption of any ciphertext segments recovers the underlying message segment.

Definition 1 (raAE correctness). *An raAE scheme* Π *is correct if, for any* K, N, G, p, M, A, C *defined as above, it holds that*

$$\begin{aligned}&(T_g, S) \leftarrow \Pi.\mathsf{StartEnc}(K, N, G)\ ;\ \Pi.\mathsf{EncSeg}(S, p, A, M) = C\\ \implies\ &S' \leftarrow \Pi.\mathsf{StartDec}(K, N, G, T_g)\ ;\ \Pi.\mathsf{DecSeg}(S', p, A, C) = M.\end{aligned}$$

In words, running StartEnc on any key, nonce, and global associated data, followed by running EncSeg on the resulting state plus any position, segment associated data, and message segment results in a ciphertext segment that will be decrypted successfully by running StartDec and DecSeg appropriately. Notice that here EncSeg is assumed to always output a valid ciphertext; our schemes will successfully encrypt for any input position indicator. In Appendix A, we show a more general definition in terms of correctness of decrypting entire segmented ciphertexts, and show that our definition here is equivalent the more general definition, given that decryption is random access.

We note that prior treatments of online AE schemes (Section 3) do not formally define correctness; it is only referred to in passing in [18], stating that for correctness of oAE schemes, "we require that decryption reverses encryption." Yet, this is not trivial to define for coAE schemes. Our formalization makes defining correctness simpler.

Confidentiality and authenticity security definitions. We now turn to defining security for raAE schemes. As is standard for authenticated encryption, we want raAE schemes to maintain confidentiality and authenticity. We capture these two goals in an all-in-one, simulation-based security definition, which we call *random-access real-or-random* (ra-ROR). Our definition is based on the standard multi-user real-or-random (mu-ROR) security notion for traditional AEAD schemes, but adapted to the random-access setting, which intuitively means that ROR security should hold on a *per-segment* basis.

In more detail, ra-ROR consists of two games, ra-Real and ra-Rand, which we formally define in Figure 8. In both games, the adversary $\mathcal{A}$ has oracles to request encryptions and decryptions of message and ciphertext segments. Importantly, $\mathcal{A}$ can arbitrarily interleave segment encryption/decryption queries corresponding to different segmented messages or ciphertexts, even across different users. We keep track of key instances (i.e., users) via variable u, and, for key instance i, we keep track of active encryption and decryption sessions (i.e., segmented message/ciphertext queries) via variables e_i and d_i, respectively. When calling EncSeg or DecSeg, the adversary specifies as input a tuple (i, j), indicating the key instance and session within that instance that this query corresponds to. In the real world, ra-Real, $\mathcal{A}$ receives outputs computed using the input raAE scheme. In the ideal world, ra-Rand, encryptions return random samples from the space of possible headers and ciphertexts for the input message length, and decryptions always return an error message.

We define the advantage of an ra-ROR adversary $\mathcal{A}$ over raAE scheme Π as

$$\mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{A}) = \left|\Pr\left[\,\text{ra-Real}^{\mathcal{A}}_{\Pi} = 1\,\right] - \Pr\left[\,\text{ra-Rand}^{\mathcal{A}}_{\Pi} = 1\,\right]\right| .$$

We assume that adversaries are *nonce respecting*, meaning they do not query StartEnc twice using the same nonce for the same key. Adversaries can repeatedly query StartDec on the same nonce and

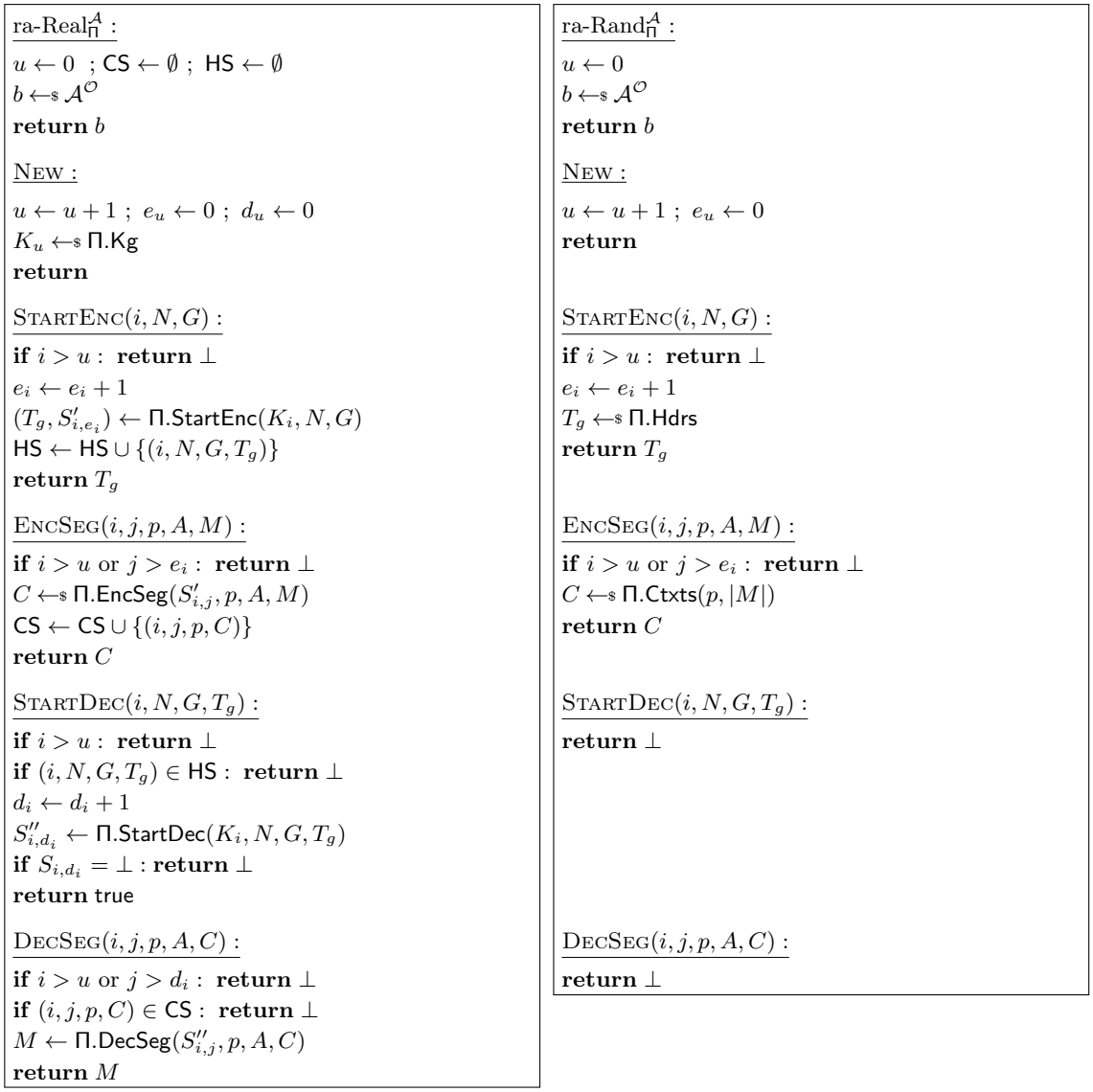

Fig. 8. Games for ra-ROR security of raAE schemes. Adversary $\mathcal{A}$ has access to oracles $\mathcal{O} = \{\textsc{New}, \textsc{StartEnc}, \textsc{EncSeg}, \textsc{StartDec}, \textsc{DecSeg}\}$.

key. We will also sometimes restrict attention to adversaries that are *position respecting*, meaning that they never query EncSeg twice on the same (i, j, p) triple.

To disallow trivial wins, the adversary cannot query the decryption oracle with a ciphertext segment under a key instance, session number, position identifier, and associated data that was obtained from an encryption query with these same parameters. Note that we do allow reusing ciphertexts across different key instances, sessions, and even across different positions within a segmented ciphertext. In other words, trivial wins only consist of decryption oracle queries that correspond exactly to previous encryption oracle queries. We prevent trivial wins explicitly in our game definitions.

In Section 5, we analyze the relationship between ra-ROR and nOAE2, the security definition for coAE schemes from prior work (Section 3).

Headerless confidentiality and authenticity. No headerless scheme can achieve ra-ROR security, because it requires no adversary be able to forge a valid input to StartDec. But for headerless schemes, StartDec always outputs a value (never $\perp$) in ra-Real. We therefore give a variant of ra-ROR security called raw-ROR. For any scheme Π, we define game raw-Real$_{\Pi}$ to be the same as ra-Real$_{\Pi}$ except that StartDec returns $\perp$ even if $S_{i,d_i} \neq \perp$. Advantage of an adversary $\mathcal{A}$ is defined as

$$\mathrm{Adv}^{\text{raw-ror}}_{\Pi}(\mathcal{A}) = \left| \Pr\left[\, \text{raw-Real}^{\mathcal{A}}_{\Pi} = 1 \,\right] - \Pr\left[\, \text{ra-Rand}^{\mathcal{A}}_{\Pi} = 1 \,\right] \right| .$$

$\underline{\text{ra-CMT-x}^{\mathcal{A}}_{\Pi}}$:
$\big(T_g, C, (G_1, p_1, K_1, N_1, A_1), (G_2, p_2, K_2, N_2, A_2)\big) \leftarrow \mathcal{A}$
if x $\neq$ p **and** $p_1 \neq p_2$:
  return false
$S_1 \leftarrow \Pi.\mathsf{StartDec}(K_1, N_1, G_1, T_g)$
$M_1 \leftarrow \Pi.\mathsf{DecSeg}(S_1, p_1, A_1, C)$
$S_2 \leftarrow \Pi.\mathsf{StartDec}(K_2, N_2, G_2, T_g)$
$M_2 \leftarrow \Pi.\mathsf{DecSeg}(S_2, p_2, A_2, C)$
if $M_1 = \perp$ or $M_2 = \perp$: **return** 0
return $(G_1, p_1, K_1, N_1, A_1) \neq (G_2, p_2, K_2, N_2, A_2)$

Fig. 9. Game template for the ra-CMT-p (with x = p) and ra-CMT (with x = ε) security games for raAE schemes. The latter requires that the two position identifiers match.

Notice that the ra-Rand game is the same for both security goals.

Commitment security. We now turn to commitment security. Unlike confidentiality and authenticity, ours is the first treatment of this security property in the online AE setting. Similar to correctness, commitment for online schemes is more complex than for traditional AEAD schemes, since we deal with two "types" of ciphertexts: segmented ciphertexts, and individual ciphertext segments therein. Commitment could thus be framed in terms of both ciphertext types. Just like correctness, however, we can focus on *commitment to individual ciphertext segments*, since this implies commitment to the entire segmented ciphertext, given that the scheme is random access.

Another subtlety of the online setting is defining the *decryption context*. That is: of all inputs to StartDec and DecSeg (excluding ciphertexts), which ones can be adversarially-controlled? Matching commitment definitions for traditional AE schemes, we assume that keys, associated data, and nonces can be manipulated by the adversary. Whether the final parameter, the position identifier p, can be manipulated by the adversary, is application-specific. We thus capture both settings via two definitions—we refer to *position* commitment as the setting when p is part of the decryption context, and can thus be controlled by the adversary; and to *positionless* commitment as the setting when p must match across both contexts.

We formally define commitment security, capturing the ideas above, via two games. They are close variants of each other, and so in Figure 9 we give a template that captures both, which we denote by ra-CMT-x. Adversary $\mathcal{A}$ produces a header and a ciphertext segment, alongside two decryption contexts that include a key, global AD, position, nonce, and AD segment. In the positionless commitment variant, ra-CMT, $\mathcal{A}$ produces a single position identifier as enforced by the first if statement; and, in the position variant, ra-CMT-p, $\mathcal{A}$ produces two identifiers. We define the positionless commitment variant by ra-CMT = ra-CMT-x with x = ε, and the position commitment variant by ra-CMT-p = ra-CMT-x with x = p. Finally, we define the advantage of a ra-CMT-p adversary $\mathcal{A}$ over a random access AEAD scheme Π as

$$\mathrm{Adv}^{\text{ra-cmt-p}}_{\Pi}(\mathcal{A}) = \Pr\left[\,\text{ra-CMT-p}^{\mathcal{A}}_{\Pi} = 1\,\right]$$

and analogously for ra-CMT.

5 Relations with Existing Notions

The preceding section introduced our new formalism for raAE. It simplifies syntax and security, relative to prior definitions. We now turn to the relationships between prior notions and ours.

ra-ROR **implies** nOAE2**.** We start with confidentiality and authenticity, comparing nOAE2 and ra-ROR. Our goal is to show that ra-ROR implies nOAE2, and thus our new security definition is stronger than prior ones. Notice, however, that nOAE2 and ra-ROR are defined over *different primitives* (coAE and raAE schemes, respectively), and so we cannot directly compare both definitions over a single scheme. Instead, we can interpret a raAE scheme as a coAE scheme, simply by restricting usage of EncSeg to be in order, and defining the next and last encryption (resp., decryption) algorithms as EncSeg (resp., DecSeg) with terminal bit in p set to 0 and 1, respectively. We formally define this simple transform, R2C, in Figure 10, which wraps any raAE scheme Π into its canonical coAE scheme.

$\underline{\mathsf{R2C}[\Pi].\mathsf{Kg}:}$
$K \leftarrow\!\!\$\ \Pi.\mathsf{Kg}$
return K

$\underline{\mathsf{R2C}[\Pi].\mathsf{Enc.init}(K, N):}$
$(T_g, S^*) \leftarrow \Pi.\mathsf{StartEnc}(K, N, \varepsilon)$
return $(1, S^*)$

$\underline{\mathsf{R2C}[\Pi].\mathsf{Enc.next}(S, A, M):}$
$(i, S^*) \leftarrow S\ ;\ p \leftarrow (i, 0)$
$C \leftarrow \Pi.\mathsf{EncSeg}(S^*, p, A, M)$
$S \leftarrow (i+1, S^*)$
return (C, S)

$\underline{\mathsf{R2C}[\Pi].\mathsf{Enc.last}(S, A, M):}$
$(i, S^*) \leftarrow S\ ;\ p \leftarrow (i, 1)$
$C \leftarrow \Pi.\mathsf{EncSeg}(S^*, p, A, M)$
return C

$\underline{\mathsf{R2C}[\Pi].\mathsf{Dec.init}(K, N):}$
$(T_g, \tilde{S}) \leftarrow \Pi.\mathsf{StartEnc}(K, N, \varepsilon)$
$S^* \leftarrow \Pi.\mathsf{StartDec}(K, N, \varepsilon, T_g)$
return $(1, S^*)$

$\underline{\mathsf{R2C}[\Pi].\mathsf{Dec.next}(S, A, C):}$
$(i, S^*) \leftarrow S\ ;\ p \leftarrow (i, 0)$
$M \leftarrow \Pi.\mathsf{DecSeg}(S^*, p, A, C)$
$S \leftarrow (i+1, S^*)$
return (M, S)

$\underline{\mathsf{R2C}[\Pi].\mathsf{Dec.last}(S, A, C):}$
$(i, S^*) \leftarrow S\ ;\ p \leftarrow (i, 1)$
$M \leftarrow \Pi.\mathsf{DecSeg}(S^*, p, A, C)$
return M

Fig. 10. Wrapper algorithm that transform an raAE scheme into a coAE scheme.

Equipped with this transform, we can now show that a scheme that is ra-ROR-secure is also (after casting it in the syntax of coAE) nOAE2-secure.

Theorem 1. *Let* Π *be a raAE scheme, and* $\mathcal{A}$ *be a* $\mathrm{nOAE2}_{\mathsf{R2C}[\Pi]}$ *adversary. We give an* ra-ROR_{Π} *adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\mathrm{noae2}}_{\mathsf{R2C}[\Pi]}(\mathcal{A}) \leq \mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{B}).$$

Adversary $\mathcal{B}$ *makes the same number of segment encryption queries as* $\mathcal{A}$*, at most as many segment decryption queries as the number of ciphertexts in* $\mathcal{A}$*'s decryption queries, the same number of decryption start queries, and one additional encryption start query. Adversary* $\mathcal{B}$ *runs in time similar to that of* $\mathcal{A}$*.*

We show the proof of this theorem in Appendix B.1. Intuitively, it follows from the fact that, if a scheme is secure when segment encryptions can be performed in any order, then it is secure in the particular case when encryptions are performed in order.

While ra-ROR implies nOAE2, the converse of Theorem 1 does not hold. Consider the contrived scheme Π shown earlier in Figure 7 and discussed in Section 3. Recall that Π is an extension to STREAM, which uses a PRF F to reveal $\mathsf{F}(K_1^*, i+1)$ during encryption of message segment i, and includes a backdoor that undermines security if $M = \mathsf{F}(K_1^*, i+1)$. As discussed earlier, since nOAE2 enforces in-order encryptions, the adversary cannot learn $\mathsf{F}(K_1^*, i+1)$ until encryption of segment $i+1$, and so leveraging the backdoor in Π requires inverting F. Conversely, consider an equivalent version of Π, adapted in the canonical way to the syntax of raAE, i.e., using $p = (i, b)$ to set the value of i in S and to choose between calling the next or last encryption/decryption algorithms. This scheme is clearly not secure under ra-ROR: the adversary can just call EncSeg with any message for segment position $i+1$ to obtain $(F(K_1^*, i+1), R, C)$, followed by a call to EncSeg for segment position i with with $M = F(K_1^*, i+1)$.

The scheme in Figure 7 thus implies that ra-ROR is *strictly* stronger than nOAE2, and so existing schemes that are nOAE2-secure do not directly satisfy ra-ROR security. In particular, it is not clear whether security holds when existing constructions are used (if allowed by the scheme) to encrypt segments in any order, a natural use case in many real-world scenarios.

ra-ROR **security of** STREAM**.** We proceed to address this gap, and show that STREAM is indeed secure under random-access encryption. While STREAM is defined as a coAE scheme, we can directly cast it in the syntax of raAE by, as described above, using $p = (i, b)$ to set S and call the correct next or last encryption/decryption algorithms. We refer to this scheme as STREAM^*, which, for completeness, we define in Figure 16 in Appendix B.2. Note that STREAM^* is headerless, and so we use the headerless variant of ra-ROR in the theorem that follows.

Theorem 2. *Let* AE *be an AEAD scheme,* $\Pi \leftarrow \mathsf{STREAM}^*[\mathsf{AE}]$*, and* $\mathcal{A}$ *be a* raw-ROR$_\Pi$ *adversary. We give a* mu-ROR$_{\mathsf{AE}}$ *adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\text{raw-ror}}_{\Pi}(\mathcal{A}) \leq \mathrm{Adv}^{\text{mu-ror}}_{\mathsf{AE}}(\mathcal{B}).$$

Adversary $\mathcal{B}$ *makes as many encryption and decryption queries as the number of segment encryption and decryptions, respectively, from* $\mathcal{A}$*. Adversary* $\mathcal{B}$ *runs in time similar to that of* $\mathcal{A}$*.*

We show the proof of this theorem in Appendix B.3. This follows from a straightforward reduction, where adversary $\mathcal{B}$ runs raw-Real$^{\mathcal{A}}_{\Pi}$ using the encryption and decryption oracles of its mu-ROR game to compute segment encryptions and decryptions, respectively.

A similar argument shows that $\mathsf{SE1}$ also satisfies (the headerless variant of) ra-ROR security, but with an additional $\mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}$ advantage term in the bound above to account for the PRF F used to derive keys at the start of encryption and decryption. This introduces an initial game hop in the proof, where we replace computations of F with random strings of the appropriate length, which then allows us to reduce to the security of AE.

Commitment security of existing schemes. Lastly, we consider commitment security. Since commitment is not addressed in prior works, there are no equivalent definitions against which to compare ra-CMT and ra-CMT-p. Furthermore, it is unclear whether existing constructions are committing or not. We address this by deriving a bound for the commitment (in)security of STREAM, which follows from a reduction to the commitment security of the underlying AEAD AE. We focus on STREAM for clarity of presentation and generality, but note that an analogous argument gives the same bound for $\mathsf{SE1}$ (and, thus, for Tink). As in Theorem 2, we show our result in terms of scheme STREAM^*, which is simply STREAM mapped to the syntax of raAE (Figure 16 in Appendix B.2).

Theorem 3. *Let* AE *be an AEAD scheme,* $\Pi \leftarrow \mathsf{STREAM}^*[\mathsf{AE}]$*, and* $\mathcal{A}$ *be a* ra-CMT$_\Pi$ *adversary. We give a* CMT$_{\mathsf{AE}}$ *adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\text{ra-cmt}}_{\Pi}(\mathcal{A}) \leq \mathrm{Adv}^{\text{cmt}}_{\mathsf{AE}}(\mathcal{B}).$$

Adversary $\mathcal{B}$ *runs in time similar to that of* $\mathcal{A}$*.*

We show the proof of Theorem 3 in Appendix B.4. Adversary $\mathcal{B}$ obtains a ciphertext and pair of decryption contexts from $\mathcal{A}$ for Π, which $\mathcal{B}$ can use in its own game against AE. Notice that the bound above is for the positionless commitment variant, and so it applies directly to the position variant too.

Let us now analyze the tightness of the bound in Theorem 3. For the position commitment variant, any non-committing AEAD leads to a non-committing instantiation of STREAM, since any CMT$_{\mathsf{AE}}$ adversary $\mathcal{A}$ can be used to construct a ra-CMT-p$_\Pi$ adversary $\mathcal{B}$ with the same advantage. On output $\big(C, (K_1, N_1, A_1), (K_2, N_2, A_2)\big)$ from $\mathcal{A}$, adversary $\mathcal{B}$ first parses the nonces as $(N_1^* \parallel i_1 \parallel b_1) \leftarrow N_1$ and $(N_2^* \parallel i_2 \parallel b_2) \leftarrow N_2$, where b_1, b_2 are single bits and i_1, i_2 are any substrings preceding them. Then, $\mathcal{B}$ outputs $\big(\varepsilon, C, (\varepsilon, (i_1, b_1), K_1, N_1^*, A_1), (\varepsilon, (i_2, b_2), K_2, N_2^*, A_2)\big)$ in its game. Notice that the probability that $\mathcal{B}$ wins game ra-CMT-p$_\Pi$ is equal to the probability that $\mathcal{A}$ wins game CMT$_{\mathsf{AE}}$: the decryption contexts from $\mathcal{B}$ result in calls to $\mathsf{AE.Dec}(K_1, (N_1^*, i_1, b_1), A_1, C)$ and $\mathsf{AE.Dec}(K_2, (N_2^*, i_2, b_2), A_2, C)$ inside $\Pi.\mathsf{DecSeg}$, matching the behavior in CMT$_{\mathsf{AE}}$.

The positionless commitment variant is more subtle, since it requires that position indicators are the same in both decryption contexts. Therefore, we cannot generically construct a ra-CMT$_\Pi$ adversary by parsing N_1 and N_2 from a ra-CMT$_\Pi$ adversary as above. Instead, we require that N_1 and N_2 are such that $(i_1 \parallel b_1) = (i_2 \parallel b_2)$, so that position indicators (i_1, b_1) and (i_2, b_2) are equal in both decryption contexts from the ra-CMT$_\Pi$ adversary. Therefore, for positionless commitment, a non-committing AEAD leads to a non-committing instantiation of STREAM if and only if there exists a CMT$_{\mathsf{AE}}$ adversary for which *(at least) the last two bits of* N_1 *and* N_2 *do not matter for its attack*, and can thus be equal. If such an adversary exists, we can construct a ra-CMT$_\Pi$ adversary in the same manner as for the position variant above. In practice, most non-committing AEADs have attacks that satisfy this condition on nonces (including AES-GCM, ChaCha20/Poly1305, etc.) [14], and so STREAM would not be ra-CMT-secure when instantiated with these.

Note that AEAD commitment is *not* a requirement of STREAM nor $\mathsf{SE1}$, and it is likely that common instantiations use non-committing AEADs. In particular, Google's Tink library supports [2,1] two AEADS: AES-GCM, which is not comitting [14]; and AES-CTR-HMAC, which is comitting [14].[2]

[2] Unless independent keys are used for CTR and HMAC, in which case AES-CTR-HMAC is not committing [14], which Tink does not do.

6 FLOE: A New raAE Scheme for Practical Constraints

Our results from the previous section establish that existing constructions, such as STREAM, are secure raAE schemes. However, such a scheme does not satisfy our original motivation, which is a FIPS-compatible mode of operation for AES-GCM that can be used securely to encrypt large amounts of data. We therefore provide a new scheme called *Fast Lightweight Online Encryption (FLOE)*. It provides a FIPS-compliant, high-performance, and secure raAE.

Practical goals. We start by outlining the practical challenges that motivated this project, which are based on real-world product concerns.

FIPS compliance and AEADs: The elegant designs of STREAM and SE1 are natural solutions to the problem of streaming encryption. Yet, they face an important usability issue: they *do not* comply with NIST's cryptographic standards. Specifically, Section C.H of FIPS 140-3 [27] mandates a handful of allowed ways of selecting IVs for use with any AEAD algorithm. The simplest is 96-bit random IVs, where the IV is generated by a FIPS-validated random number generator. But STREAM and SE1 use structured IVs with the underlying AEAD: they append segment numbers and terminal bits (both critical to security) to the nonce. Another FIPS-compliant option is to use deterministic nonces, but these have a prescribed structure that the nonces used in STREAM and SE1 do not match. FIPS compliance is important in many production environments, for example, because customers of an encryption-using product require it.

Even if an application need not be FIPS compliant, many production environments nevertheless use FIPS-validated cryptographic modules. Many of these do not allow callers to specify deterministic nonces, and thus it is impossible to implement STREAM nor SE1 using those systems.

Ultimately, the landscape in practice is that we need an raAE scheme that uses black-box calls to an underlying AEAD that uses random IVs.

FIPS compliance and KDFs: Similarly to the above, FIPS compliance mandates use of approved key derivation functions (KDFs). NIST specifies in SP 800-133 Revision 2 [5] the use of KDFs from SP 800-108 Revision 1 [11], which includes ones based on HMAC [23], KMAC [22], and CMAC [13]. This rules out some design approaches, such as DNDK [15,16].

Commitment: As shown in Section 5, the ra-CMT and ra-CMT-p security of STREAM and SE1 reduce to the commitment security of the underlying AEAD. But most widely used AEADs are not committing (e.g., AES-GCM, ChaCha20/Poly1305, etc.), and so these schemes do not provide committing raAE. It would instead be desirable to decouple commitment of the raAE scheme from commitment of the AEAD, thereby enabling the use of generic AEADs while maintaining security.

Long key lifespans: We need to support encryption of very large payloads, which requires splitting up plaintexts across many AEAD encryptions. This can, in turn, lead to small key lifespans, meaning the amount of plaintext that can be encrypted under the same key. Small lifespans complicate deployments, which then must rotate keys frequently. As required by FIPS 140-3, when using random nonces, one needs to ensure a low probability of nonce collisions under the same key; such collisions lead to complete security loss. In particular, NIST requires that this probability be no greater than 2^{-32} [25]. For AES-GCM with 96-bit random nonces, this implies key rotations after just 2^{32} encryption calls, which is well within the scale of modern workloads [21].

Neither STREAM nor SE1 provide satisfying key lifespans. STREAM uses the same input key for all AEAD encryptions, and so even if one replaces its use of structured nonces with random nonces, one would require key rotations after encrypting just 2^{32} segments (regardless of the number of messages). SE1 has better key lifespans, due to deriving a fresh key for every plaintext. This, however, still implies that we cannot encrypt a single message with more than 2^{32} segments.

Even ignoring nonce collisions, the 128-bit block size of AES also constraints the lifespan of keys: after q encryption calls with total length at most σ blocks, an adversary has distinguishing advantage $(\sigma + q + 1)^2/2^{129}$ [20], which exceeds 2^{-32} after just 2^{49} blocks. Once again, this limits the amount of data that can be encrypted across messages (for STREAM) and within a message (for SE1).

Other goals: In addition to the main requirements above, there are a number of other usability goals that would make a scheme more friendly for developers. In particular, it would be useful for a scheme to be able to output *different error messages during decryption*, granularly indicating where and why decryption failed (instead of a single, generic symbol $\perp$), which can aid developers when debugging systems. However, returning multiple error messages may lead to security failures, as it could introduce side channels that can be leveraged by an adversary [9]. Therefore, we want a scheme to remain secure while supporting an expanded set of error symbols.

Parameter	Symbol	Description
Input key length	κ	Length of key K input to StartEnc/StartDec
Input nonce length	ℓ_N	Length of nonce N input to StartEnc/StartDec
Intermediate key length	κ_{in}	Length of message key K_{in} derived from K
Header length	ℓ_{T_g}	Length of ciphertext header T_g
Segment length	s	Length of non-terminal message segments
Max. segments	$\max_s$	Max. number of segments a message can have
AEAD block size	n	Block size of AEAD AE used in EncSeg/DecSeg
AEAD key length	κ_{se}	Length of key K_{se} for AE derived from K_{in}
AEAD nonce length	v_e	Length of nonce for AE sampled at random
AEAD tag length	t	Length of tag for ciphertexts from AE
Rotation mask	r	Number of encryptions (2^r) before rotating K_{se}

Fig. 11. Parameters for FLOE[AE, F], which we capture in variable ρ. Segment length s is the only parameter not fixed by the FLOE specification and left to the control of developers.

FLOE.Kg :
$K \xleftarrow{\$} \{0,1\}^\kappa$
return K

FLOE.StartEnc(K, N, G) :
$T_g \leftarrow \mathsf{F}(K, \rho \,\|\, N \,\|\, \mathsf{c}_1 \,\|\, G, \ell_{T_g})$
$K_{in} \leftarrow \mathsf{F}(K, \rho \,\|\, N \,\|\, \mathsf{c}_2 \,\|\, G, \kappa_{in})$
$S \leftarrow (K_{in}, N, G)$
return (T_g, S)

FLOE.EncSeg(S, p, M) :
$(K_{in}, N, G) \leftarrow S$; $(i, b) \leftarrow p$
$k \leftarrow \mathsf{mask}(i, r)$
$K_{se} \leftarrow \mathsf{F}(K_{in}, \rho \,\|\, N \,\|\, \mathsf{c}_3 \,\|\, \mathsf{encode}(k, 8) \,\|\, G, \kappa_{se})$
$A \leftarrow \mathsf{encode}(i, 8) \,\|\, \mathsf{encode}(b, 1)$
$C \xleftarrow{\$} \mathsf{AE.Enc}(K_{se}, A, M)$
if $b = 0$: $H \leftarrow \mathsf{c}_4$
else : $H \leftarrow \mathsf{encode}(4 + |C|_8, 4)$
return $H \,\|\, C$

FLOE.Ctxts(p, ℓ) :
$(i, b) \leftarrow p$
if $b = 0$: **return** $\{\mathsf{c}_4\} \times \{0,1\}^{\mathsf{clen}(s)}$
else : **return** $\{\mathsf{encode}(4 + \ell, 4)\} \times \{0,1\}^{\mathsf{clen}(s)}$

FLOE.StartDec(K, N, G, T_g) :
if $T_g \neq \mathsf{F}(K, \rho \,\|\, N \,\|\, \mathsf{c}_1 \,\|\, G, \ell_{T_g})$: **return** $\perp$
$K_{in} \leftarrow \mathsf{F}(K, \rho \,\|\, N \,\|\, \mathsf{c}_2 \,\|\, G, \kappa_{in})$
$S \leftarrow (K_{in}, N, G)$
return S

FLOE.DecSeg$(S, p, H \,\|\, C)$:
if $S = \perp$: **return** $\perp$
$(K_{in}, N, G) \leftarrow S$; $(i, b) \leftarrow p$
if $b = 0$ and $H \neq \mathsf{c}_4$: **return** $\perp$
if $b = 1$ and $H \neq \mathsf{encode}(|C| + 4, 4)$: **return** $\perp$
$k \leftarrow \mathsf{mask}(i, r)$
$K_{se} \leftarrow \mathsf{F}(K_{in}, \rho \,\|\, N \,\|\, \mathsf{c}_3 \,\|\, \mathsf{encode}(k, 8) \,\|\, G, \kappa_{se})$
$A \leftarrow \mathsf{encode}(i, 8) \,\|\, \mathsf{encode}(b, 1)$
$M \leftarrow \mathsf{AE.Dec}(K_{se}, A, C)$
return M

Fig. 12. Our raAE scheme FLOE using AEAD AE, KDF F, and associated parameter set ρ (Figure 11). All c_i are fixed constants, appropriately encoded: c_1 = ‘‘`HEADER_TAG:`’’; c_2 = ‘‘`MESSAGE_KEY:`’’; c_3 = ‘‘`DEK:`’’; and c_4 = `0xFFFFFFFF`. Function $\mathsf{encode}(x, \ell)$ encodes input x as an usigned big-endian value of length ℓ bytes; and function $\mathsf{mask}(i, r) = i - (i \bmod 2^r)$ masks the low-order r bits of i to 0.

For clarity of presentation, we defer discussion of multiple error symbols to Appendix F, and restrict attention to a single error symbol $\perp$ in the rest of the body of the paper. In Appendix F, we specify the full list of FLOE's error symbols, refine ra-ROR and ra-CMT-x to account for schemes that may return multiple error symbols during decryption, and show that FLOE meets these definitions.

The FLOE scheme. Motivated by these requirements, we present FLOE. It is configurable, with a number of developer-specified parameters; [3] see Figure 11. We denote the parameters used via variable ρ. Pseudocode appears in Figure 12, and a diagram depicting the main encryption steps in Figure 13.

FLOE relies on two underlying cryptographic components: a variable-output-length PRF F, and a randomized AEAD scheme AE. We discuss their instantiations later. In more detail, we assume a PRF $\mathsf{F} \colon \mathcal{K} \times \{0,1\}^* \times \mathbb{N} \to \mathcal{Y}$ where $\mathcal{K} \supseteq \{0,1\}^\kappa \cup \{0,1\}^{\kappa_{in}}$ supports the relevant key lengths and $\mathcal{Y} \supseteq$

[3] In practice, all of these values are fixed by the FLOE specification (see Section 8) with the exception of segment length.

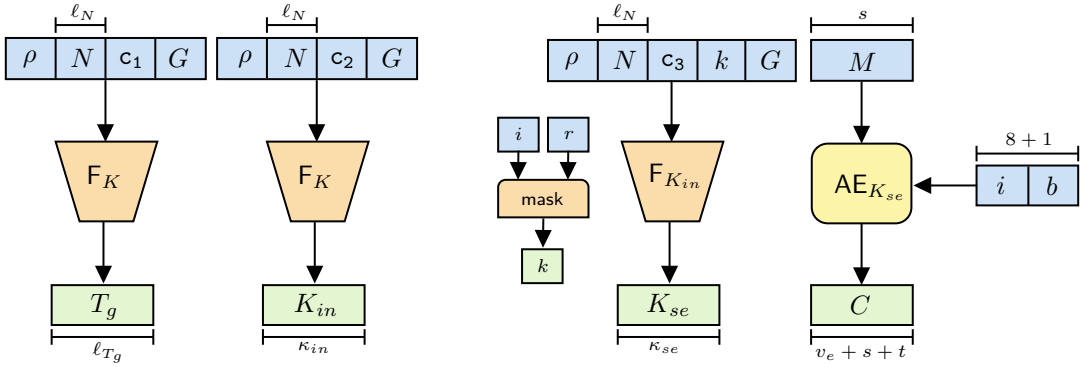

Fig. 13. Diagram of the main steps in the encryption process of FLOE[AE, F]. The left side shows StartEnc, and the right side shows EncSeg. See pseudocode in Figure 12 for the full steps of FLOE.

$\{0,1\}^{\ell_{T_g}} \cup \{0,1\}^{\kappa_{in}} \cup \{0,1\}^{\kappa_{se}}$ supports intermediate key length and the AEAD's key length. We require that $\mathsf{F}(K, X, \ell)$ outputs a bit string of length ℓ bits, and we will only use ℓ such that $2^\ell \in \mathcal{Y}$.

FLOE encryption works as follows. Running $\mathsf{StartEnc}(K, N, G)$ on key K, nonce N, and global AD G generates a PRF-output T_g that commits to K, N, G. It also commits to the parameters ρ, which will be useful in practice for preventing use of the incorrect parameters during decryption. Finally, StartEnc also prepares an intermediate key (also called message key). For simplicity of implementation, we use two calls to the same PRF F, with almost identical inputs except, importantly, domain separation provided by fixed constants. Note that all inputs to F except for the global associated data are prefix-free. The resulting state is the intermediate key K_{in}, and we include also N and G since we use them again during segment encryptions.

Encryption of an individual message segment derives an AEAD key, using the same F in the same manner as before, but with K_{in} and a distinct constant. Here, however, derivation depends on the segment number and a parameter r: we zero out the low-order r bits of the segment number before we include it in the AEAD key derivation. We denote this masking process via function $\mathsf{mask}(i, r) = i - (i \bmod 2^r)$. This results in what we call *key epochs*: we use the same AEAD key with 2^r consecutive segments. This allows a performance-security trade-off, as discussed further below. Finally, we encrypt the message using *randomized* AEAD AE (thus complying with FIPS), and under an associated data that includes an encoding of the position (that is, segment number and terminal bit), which prevents from, e.g., dropping or reorganizing ciphertext segments. (In theory, we can therefore allow up to $\max_s = 2^{64}$ segments, though the forthcoming specification will by default establish a more reasonable limit of 2^{40}). We output the resulting ciphertext (which includes the nonce and tag) preceded by a 4-byte plaintext header that is either a fixed constant string (0xFFFFFFFF for non-terminal segments) or the length of the full segment output (four plus the length of the ciphertext in bytes) for a terminal segment.

Decryption proceeds in the obvious way, rederiving keys and performing decryption using the underlying AEAD with the appropriate key. Segment decryption additionally checks that the plaintext header is correct. In implementations, all routines should additionally check that inputs are allowed (e.g., of appropriate lengths); we omit such checks from our pseudocode for simplicity.

Discussion. FLOE requires non-terminal segments to be fixed lengths. This is pragmatic since applications will use FLOE to encrypt very large files by splitting them into $m - 1$ full-length segments and possibly one partial-length segment, thus making length encodings of non-terminal segments superfluous. We could support variable length non-terminal segments; our security analyses allow adversaries to make such queries and so cover this case even though it deviates from intended usage.

One may wonder why have intermediate message keys. The reason is defense-in-depth. By deriving K_{in} using F, even if K_{in} is somehow compromised, the adversary learns nothing about the root key K and, in particular, other plaintexts will still be secure. For example, this means implementations can use a separate, isolated module for StartEnc and StartDec, providing extra protections for K.

From a security perspective, because K_{in} is already bound to N and G, we could drop the latter two from the input to the final AEAD key derivation within EncSeg and DecSeg. But in practice N and G are short, and it simplifies implementation to have similar F input message format everywhere. It also somewhat simplifies security analysis and slightly improves some security bounds (i.e., because we do not have to worry about intermediate key collisions when unique nonces are used).

The parameter r allows a security-performance trade-off. Note that for each key epoch, encryption (or decryption) can cache K_{in} and skip recomputing F and also avoid any re-keying costs for the underlying AEAD. For our FIPS-compliant instantiation, we will end up using a relatively slow F based on HMAC, and so larger r will be rather beneficial for performance. On the other hand, as we will show in the security analysis in Section 7, increasing r lowers concrete security when using an underlying AEAD with relatively short random nonces (such as AES-GCM's 96-bit nonces) due to increasing probability of nonce collision under a same AEAD key.

7 Security Analysis of FLOE

We analyze the security of FLOE in this section. We start with confidentiality and authenticity, and then turn to commitment.

Confidentiality and authenticity. Our target security notion is ra-ROR, as defined in Section 4. We first present a general security bound, reducing to the security of the underlying AEAD AE and KDF F. We later instantiate this result with a particular AE and F, to obtain concrete bounds for specific useful-in-practice FLOE configurations.

Theorem 4 (FLOE confidentiality and authenticity). *Let AE be an AEAD scheme, F be a PRF, and $\Pi = \mathsf{FLOE}[\mathsf{AE}, \mathsf{F}]$ with associated parameter set ρ. Let $\mathcal{A}$ be a position-respecting* ra-ROR$_\Pi$ *adversary that makes q_{se} start encryption queries, q_{e} segment encryption queries, q_{sd} start decryption queries, and q_{d} segment decryption queries. We give a* mu-PRF$_{\mathsf{F}}$ *adversary $\mathcal{B}$ and a* mu-ROR\$$_{\mathsf{AE}}$ *adversary $\mathcal{C}$ such that*

$$\mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}) + \mathrm{Adv}^{\text{mu-ror\$}}_{\mathsf{AE}}(\mathcal{C}) + \frac{q_{\mathsf{sd}}}{2^{\ell_{T_g}}}.$$

Adversary $\mathcal{B}$ makes at most $q_{\mathcal{B}} = \max\{2(q_{\mathsf{se}} + q_{\mathsf{sd}}), q_{\mathsf{e}} + q_{\mathsf{d}}\}$ queries to at most $q_{\mathcal{B}}$ users. Adversary $\mathcal{C}$ makes q_{e} encryption queries and q_{d} decryption queries to at most $q_{\mathsf{e}} + q_{\mathsf{d}}$ users. Both adversaries run in time similar to that of $\mathcal{A}$.

A benefit of FLOE's design and our new, random-access-first definitions is that the proof follows from a straightforward analysis: the simpler proofs are to verify the more assurance we have in scheme security. We defer details to Appendix C, and just sketch the game-hopping argument here. By the mu-PRF security of F, and the fact that $\mathcal{A}$ is nonce-respecting, we can replace computations of header tags and intermediate message keys during StartEnc and StartDec calls with sampling random strings. Then, by the mu-PRF security of F again, and the fact that message keys are now random, we can similarly replace computations of AEAD keys within EncSeg and DecSeg with sampling random strings. Next, by the mu-ROR\$ security of AE, and the fact that AEAD keys are now random, we can replace all AEAD encryptions with sampling random ciphertexts, and decryptions always returning $\perp$. Lastly, in this world the adversary can only query a valid header tag to DecSeg with probability at most $1/2^{\ell_{T_g}}$—a union bound gives the final term in the advantage inequality.

Bounds for concrete instantations. Theorem 4 is a general template, which can be instantiated with a specific AE and F to obtain concrete bounds. We now consider FLOE-GCM, which is FLOE but with AE being AES-GCM using 96-bit random nonces, and F being HMAC in the expand mode of HKDF with SHA-384 as the underlying hash function. These choices align FLOE-GCM with FIPS requirements, and allow black-box use of cryptographic modules that only provide interfaces for AES-GCM using internally generated random nonces.

Towards full security bounds, we will rely on (a slight variant of) the multi-user AES-GCM bound from Gueron and Ristenpart [16], shown below, which follows by applying hybrid arguments to the single-user confidentiality and authentiticy bounds of [20]. The bound in [16] considers a granular treatment of adversaries, accounting for how queries are distributed across key instances. In particular, they introduce the notion of a $\vec{q}$-*bound* adversary, which makes at most i encryption queries to $\vec{q}_i$ key instances. For example, an adversary that is $\vec{q}$-bound for $\vec{q} = (10, 5, 0, ..., 0)$ makes at most one query to 10 different key instances, and two queries to at most five different key instances. Note that the total number of key instances is $\sum_i \vec{q}_i$ (15 in our example), and the total number of queries is $\sum_i i \cdot \vec{q}_i$ (20 in our example). They also define the *weight* of $\vec{q}$, denoted by $w(\vec{q})$, as the largest $\vec{q}_i$. This treatment will be useful for us, allowing us to derive more fine-grained bounds. Below, let $\delta_n(a) = (1 - (a-1)/2^n)^{-a/2}$.

Theorem 5 (AES-GCM multi-user security [16]). *Let* $\mathsf{AE} = \mathsf{AES\text{-}GCM}$ *used with 96-bit nonces and* τ*-bit tags built from an* n*-bit block cipher* E*. Let* $\mathcal{A}$ *be a* $\vec{q}$*-bound* mu-ROR$\$_{\mathsf{AE}}$ *adversary that makes at most* d *decryption queries, where the maximum plaintext length across all encryption queries is* L_1 *blocks and the maximum input length across all encryption and decryption queries is* L_2 *blocks. Then, there exists an explicit* mu-PRP$_{\mathsf{E}}$ *adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\text{mu-ror\$}}_{\mathsf{AE}}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\text{mu-prp}}_{\mathsf{E}}(\mathcal{B}) + \sum_{i=1} \left(\frac{\vec{q_i} \cdot (iL_1 + i + 1)^2}{2^{n+1}} + \frac{\vec{q_i} \cdot i \cdot (i-1)}{2^{97}} \right) + \frac{d(L_2+1)}{2^\tau} \cdot \delta_n(w(\vec{q}) \cdot (L_1+1) + d + 1).$$

Adversary $\mathcal{B}$ *runs in time that of* $\mathcal{A}$*, and makes at most* $\sum_i \vec{q_i}$ *key generation queries,* $\sum_i \vec{q_i} \cdot i \cdot (L_1+2)$ *enciphering queries, and* d *decryption queries.*

This presentation of Theorem 5 makes two minor modifications to the bounds shown in [16]. First, [16] assumes nonce-respecting adversaries, whereas we deal with randomized nonces. However, by a standard reduction (see, e.g., [19]), we can simply add an additional term to the bound to account for the probability of nonce collisions under a same key; this can be derived using a birthday bound to compute the probability of collisions under a single key, and taking a union bound over all key instances, which corresponds to the $\vec{q_i} \cdot i \cdot (i-1)/2^{97}$ term in Theorem 5. Second, [16] shows separate confidentiality and authenticity bounds, but, by a standard reduction, these can be summed to get a single all-in-one bound.

We can now apply Theorem 5 to Theorem 4, and get a bound for FLOE-GCM. Notice, however, that the parameters in Theorem 5 depend on the specific *query profile* from the ra-ROR adversary $\mathcal{A}$: the key instance of AE used by an EncSeg(i, j, p, A, M) query from $\mathcal{A}$ depends on the key epoch that p corresponds to. For instance, if $\mathcal{A}$ makes q_{e} queries, all positions p therein could correspond to unique key epochs, leading to q_{e} key instances in mu-ROR\$ and one query per key; conversely, the positions p could fill all key epochs, leading to $q_{\mathsf{e}}/2^r$ key instances and 2^r queries per key; and so on. These yield different instantiations of Theorem 5, and so we need a more fine-grained handle on queries from $\mathcal{A}$:

Definition 2. *An* ra-ROR *adversary* $\mathcal{A}$ *has query profile of* $\vec{f}$ *if it makes at most* i EncSeg *oracle queries for* $\vec{f_i}$ *key epochs.*

In words, $\vec{f_i}$ encodes the number of key epochs for which EncSeg queries from $\mathcal{A}$ use up to i positions from that epoch (out of 2^r total). Since, recall, we assume that $\mathcal{A}$ is position-respecting, it follows that $|\vec{f}| = 2^r$. Going back to our examples above, all unique epochs per query corresponds to $\vec{f} = (q_{\mathsf{e}}, 0, ..., 0)$, and full key epochs corresponds to $\vec{f} = (0, ..., 0, q_{\mathsf{e}}/2^r)$. This formalization of query profiles now lets us benefit from the granular treatment of AEAD queries in Theorem 5,[4] arriving at the bound for FLOE-GCM shown below, where we use the ideal cipher model for the security of the block cipher underlying AES.

Theorem 6 (FLOE-GCM confidentiality and authenticity). *Let* $\mathsf{AE} = \mathsf{AES\text{-}GCM}$ *used with 96-bit nonces and* τ*-bit tags built from an ideal cipher* $\mathsf{E} : \{0,1\}^{\kappa_{se}} \times \{0,1\}^n \to \{0,1\}^n$*,* F *be a PRF, and* $\Pi = \mathsf{FLOE}[\mathsf{AE}, \mathsf{F}]$ *with associated parameter set* ρ*. Let* $\mathcal{A}$ *be a position-respecting* ra-ROR$_\Pi$ *adversary that makes* p *primitive queries,* q_{se} *start encryption queries,* q_{e} *segment encryption queries with query profile* $\vec{f}$*,* q_{sd} *start decryption queries, and* q_{d} *segment decryption queries. We give a* mu-PRF$_{\mathsf{F}}$ *adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}) + \frac{q_{\mathsf{e}}p + q_{\mathsf{e}}(q_{\mathsf{e}}-1)}{2^{\kappa_{se}-1}} + \sum_{i \in [1,2^r]} \left(\vec{f_i} \cdot \frac{(is+in+n)^2}{n^2 2^{n+1}} + \vec{f_i} \cdot \frac{i(i-1)}{2^{97}} \right) + \frac{q_{\mathsf{d}}(s+10n)}{n2^\tau} \cdot \delta_n(w(\vec{f}) \cdot (s/n+1) + q_{\mathsf{d}} + 1) + \frac{q_{\mathsf{sd}}}{2^{\ell_{T_g}}}.$$

Adversary $\mathcal{B}$ *makes at most* $q_{\mathcal{B}} = \max\{2(q_{\mathsf{se}} + q_{\mathsf{sd}})\}$ *and runs in time similar to that of* $\mathcal{A}$*.*

Above, r and s correspond to the rotation mask and segment size in ρ (see Figure 11), respectively. We show the simple proof of this theorem in Appendix D, which follows from instantiating Theorem 4 with Theorem 5 for the $\mathrm{Adv}^{\text{mu-ror\$}}_{\mathsf{AE}}$ term, and translating $\vec{f}$ for $\mathcal{A}$ to a $\vec{q}$-bound for $\mathcal{C}$.

Theorem 6 gives the adversary flexibility to distribute queries however they wish (subject to the position-respecting assumption). We can focus on particular query profiles of interest, to understand

[4] This is similar to how Gueron and Ristenpart [15] introduce the notion of "nonce head respecting" adversaries in their security definition, in order to interface with the $\vec{q}$-bound notion in their bound for AES-GCM.

security in these settings. For instance, we can consider query profile $\vec{f} = (0, 0, ..., q_{\mathsf{e}}/2^r)$ mentioned earlier, which corresponds to the "canonical" usage of an online scheme (i.e., where all segments within a key epoch are encrypted). This yields the following for the summation term in Theorem 6:

$$\frac{q_{\mathsf{e}}(2^r s + 2^r n + n)^2}{n^2 2^{n+r+1}} + \frac{q_{\mathsf{e}}(2^r - 1)}{2^{97}}.$$

Furthermore, since each segment encryption query is of length (at most) s, the total number of encrypted bytes across all queries is $Q \coloneqq q_{\mathsf{e}} s$, and so we can restate this equation as

$$\frac{Q(2^r s + n2^r + n)^2}{sn^2 2^{n+r+1}} + \frac{Q(2^r - 1)}{s2^{97}}.$$

We discuss our confidentiality and authenticity security bounds in more detail in Section 8, where we analyze how adversarial advantage scales as a function of FLOE's parameters, and thus the security achieved in practice.

Commitment security. We now shift gears to commitment security, where our target security notion is ra-CMT, as defined in Section 4.

Positionless commitment: Unlike STREAM and SE1, the positionless commitment security of FLOE does not depend on the commitment security of AE. As mentioned in Section 6, FLOE's ciphertext header commits to the key, nonce, and global associated data, and thus the collision resistance of F guarantees that two different decryption contexts will not both correctly pass the header check during decryption start. We formalize this in the theorem below.

Theorem 7 (FLOE positionless commitment). *Let* AE *be an AEAD scheme,* F *be a PRF, and* $\Pi = \mathsf{FLOE}[\mathsf{AE}, \mathsf{F}]$ *with associated parameter set* ρ*. Let* $\mathcal{A}$ *be a* ra-CMT$_\Pi$ *adversary. We give a* CR$_{\mathsf{F}}$ *adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\text{ra-cmt}}_{\Pi}(\mathcal{A}) \leq \mathrm{Adv}^{\text{cr}}_{\mathsf{F}}(\mathcal{B})\,.$$

Adversary $\mathcal{B}$ *runs in time similar to that of* $\mathcal{A}$*.*

We show the proof of this theorem in Appendix E. During every segment decryption, AEAD nonces are included in the ciphertext, and so are committed to explicitly. Then, AD segments are computed from p, which are required to be the same in both decryption contexts in ra-CMT. Lastly, AEAD keys are derived from the input key, nonce, and global associated data, and so their commitment reduces to the collision resistance of F, as mentioned above.

Position commitment: Like STREAM, the position commitment of FLOE depends on the commitment security of AE. This is due to the fact that segment identifiers are not committed to, and so $\mathcal{A}$ could submit two decryption contexts with different p_1 and p_2 that would lead to different AD segments A_1 and A_2. Therefore, if AE is not committing, these would both result in valid decryptions.

Theorem 8 (FLOE position commitment). *Let* AE *be an AEAD scheme,* F *be a PRF, and* $\Pi = \mathsf{FLOE}[\mathsf{AE}, \mathsf{F}]$ *with associated parameter set* ρ*. Let* $\mathcal{A}$ *a* ra-CMT-p$_\Pi$ *adversary. We give a* CMT$_{\mathsf{AE}}$ *adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\text{ra-cmt-p}}_{\Pi}(\mathcal{A}) \leq \mathrm{Adv}^{\text{cmt}}_{\mathsf{AE}}(\mathcal{B}).$$

Adversary $\mathcal{B}$ *runs in time similar to that of* $\mathcal{A}$*.*

This theorem follows by an analogous reduction to that of Theorem 3, so we omit it for brevity. We note that, since AES-GCM is not committing, consequently FLOE-GCM is not position committing. However, minor modifications to FLOE could add support for position commitment. For example, one could simply add p to ciphertext segments, at the cost of an additional 9 bytes of ciphertext length.

8 Performance of FLOE

As discussed in preceding sections, rotation mask and segment size are the two key levers of FLOE, guiding its usage of the underlying AEAD. These parameters are critical for both security and performance, and thus their selection is an important consideration for practical deployment. We explore this security-performance tradeoff in detail in this section, focusing on FLOE-GCM. We first revisit our security bounds

from Section 7, quantifying how adversarial advantage scales as a function of these parameters, and thus the security achieved in practice. We then turn to FLOE-GCM's performance, analyzing how parameter selection translates to computational overhead, followed by an experimental evaluation of FLOE-GCM's runtime.

Security. Our security bound for FLOE-GCM (Theorem 6) relies on various parameters and adversarial considerations. We start by contextualizing our bound for practical parameter choices, analyzing what this entails for security in practice.

Interpreting our bound: Let us first fix standard parameter values for AES-GCM: a block size of $n = 128$ bits for AES-GCM, tag size of $\tau = 128$ bits, and key size of $\kappa_{se} = 256$ bits. Similarly, let us set the header length to be $\ell_{T_g} = 256$ bits. Next, recall that Theorem 6 depends on the adversary's specific query profile, i.e., how their segment encryption queries are distributed across key epochs. For our analysis, we will focus on the "canonical" query profile mentioned in Section 7, where the adversary encrypts all 2^r positions in every key epoch. In this setting and with these parameters, our bound from Theorem 6 becomes:

$$\begin{aligned}\mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{A}) \leq & 2 \cdot \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}) + \frac{q_{\mathsf{e}}p + q_{\mathsf{e}}(q_{\mathsf{e}}-1)}{2^{255}} + \frac{q_{\mathsf{sd}}}{2^{256}} + \\ & \frac{Q(2^r s + 2^{r+7} + 2^7)^2}{s2^{r+143}} + \frac{Q(2^r - 1)}{s2^{97}} + \frac{q_{\mathsf{d}}(s+2^7)}{2^{135}} \cdot \delta_n(Q/2^{r+7} + Q/(s2^r) + q_{\mathsf{d}} + 1).\end{aligned}$$

To simplify this further, following [16], let us assume that $Q/2^{r+7} + Q/(s2^r) + q_{\mathsf{d}} + 1 \leq 2^{64}$. Then, we can rely on a result from Bernstein [8] that shows that, if $n = 128$ and $x \leq 2^{64}$, it follows that $\delta_n(x) \leq 1.7$. We thus arrive at:

$$\begin{aligned}\mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{A}) \leq & 2 \cdot \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}) + \frac{q_{\mathsf{e}}p + q_{\mathsf{e}}(q_{\mathsf{e}}-1)}{2^{255}} + \frac{q_{\mathsf{sd}}}{2^{256}} + \\ & \frac{Q(2^r s + 2^{r+7} + 2^7)^2}{s2^{r+143}} + \frac{Q(2^r - 1)}{s2^{97}} + \frac{q_{\mathsf{d}}(s+2^7)}{2^{134}}. \end{aligned} \tag{1}$$

With this simplified bound, we can now analyze its dominant terms, and how they depend on r and s. First, if $r > 1$, the dominant term in Equation 1 is $Q(2^r - 1)/(s2^{97})$, which corresponds to the probability of nonce collisions under any individual AEAD key. Notice that this term *decreases if r decreases and/or s increases*. This makes sense: rotating keys more frequently implies fewer nonces are used per key, and larger segment sizes implies fewer key instances are required for a fixed adversarial workload. Therefore, in this case, lower r and larger s improve security.

If $r = 1$, however, this term disappears: if keys are only used once, the probability of same-key nonce collisions is zero. If so, assuming that $q_{\mathsf{e}}/q_{\mathsf{d}} \geq 512s$, the dominant term is instead

$$\frac{Q(2^0 s + 2^{0+7} + 2^7)^2}{s2^{0+143}} = \frac{Q(s+2^8)^2}{s2^{143}},$$

which corresponds to the multi-user security of AES-GCM. Notice how, converse to the $r > 1$ case, *decreasing s improves security*, as this term is proportional to s. This follows from the fact that decreasing s decreases the number blocks that the adversary can encrypt under each AEAD key. If $q_{\mathsf{e}}/q_{\mathsf{d}} < 512s$ instead, the dominant term becomes $q_{\mathsf{d}}(s+2^7)/2^{134}$, which is also proportional to s.

In summary, decreasing r improves security, as this lowers the probability of nonce collisions. Decreasing s also improves security for the same reason, unless we derive a fresh key for every segment encryption, in which case a lower s leads to better security.

Security scaling: To make this analysis concrete, we now quantify adversarial advantage for different values of r and s. Figure 14 shows how (the dominant terms of) Equation 1 scales as we vary r and s, under a fixed adversarial workload of $Q = 2^{50}$ encrypted plaintext bytes (i.e., one petabyte). Notice how, as explained earlier, decreasing r improves security, and increasing s improves security if and only if $r > 1$. The red dashed horizontal line indicates where adversarial advantage crosses 2^{-32}. We can see that, even for a fairly small segment size of 1 KB, a large rotation mask of $r = 25$ yields acceptable levels of security.

We can similarly quantify adversarial advantage as a function of the adversary's workload Q instead, fixing values for r and s. In particular, notice that Equation 1 is a linear function of Q. So, if we set s and r to, for example, $s = 1$ MB and $r = 20$, the dominant terms of Equation 1 become

$$\frac{Q(2^{40} + 2^{27} + 2^7)^2}{2^{183}} + \frac{Q(2^{20} - 1)}{2^{117}} \leq \frac{Q}{2^{102}} + \frac{Q}{2^{97}} = \frac{33Q}{2^{102}}.$$

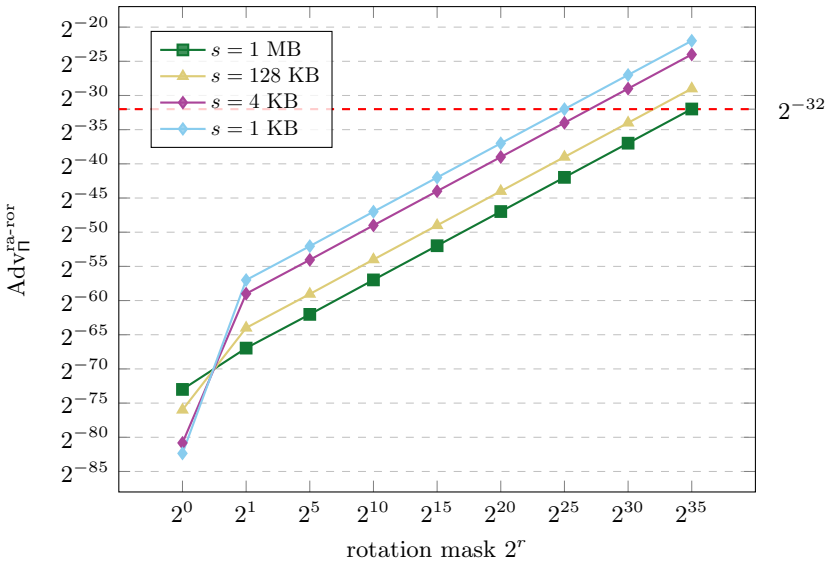

Fig. 14. Adversarial advantage as a function of r and s, for an adversarial workload of $Q = 2^{50}$ and the canonical query profile (Section 7). Lower is better. The red horizontal line indicates where adversarial advantage crosses 2^{-32}. Notice that decreasing r improves security, and increasing s improves security if and only if $r > 1$.

Therefore, an adversary can encrypt approximately 2^{65} bytes (32,786 petabytes or 32 exabytes) before their advantage is greater than 2^{-32}.

Performance. Our analysis thus far shows that FLOE's security benefits from a smaller rotation mask and larger segment size. However, these parameter choices heavily affect FLOE's performance, which we now explore.

Analytical evaluation: We start by analyzing how varying r and s lead to additional computational steps during message segment encryptions, thus arriving at closed-form formula for overhead as a function of some fixed runtime for F and AES-GCM's underlying operations.

We first consider key derivations. Let T_{F} be the cost of deriving a key of length κ_{se} using F. Decreasing either r or s results in additional AEAD keys: the former results in deriving keys more frequently, and the latter results in more segment encryptions. In particular, for a segmented message of size ℓ, the number of required keys are $\left\lceil \frac{\ell}{s2^r} \right\rceil$, which leads to an overhead of $\mathsf{T}_{\mathsf{F}} \cdot \left\lceil \frac{\ell}{s2^r} \right\rceil$.

Turning to AEAD encryptions, only s affects performance, since encryption overhead is independent of the key being used. Decreasing s affects encryption overhead in two ways: more encryption calls are required, but the input to each is smaller. We refer readers to [24, Figure 1] for a helpful diagram of the steps involved in AES-GCM's encryption process. First, notice that, since AES-GCM uses counter mode for encryption, the time required to process each plaintext block is the same regardless of how these blocks are distributed across encryption calls: each requires one call to AES, two XOR operations, and one GHASH multiplication. Then, each new encryption call requires one additional AES computation, two additional GHASH multiplication operations, and two additional XOR operations. Furthermore, since FLOE uses random nonces, each encryption call requires sampling a new nonce of length v_e.

To make this concrete, let $\mathsf{T}_{\mathsf{AES}}$ be the cost of one AES computation, $\mathsf{T}_{\mathsf{GHASH}}$ be the cost of one GHASH multiplication, and $\mathsf{T}^{v_e}_{\mathsf{rand}}$ be the cost of sampling a random string of length v_e; we omit the cost of XOR operations for clarity as their overhead is negligible. Then, for a segmented message of size ℓ, the total encryption cost is

$$\mathsf{T}_{\mathsf{AES}} \cdot \left(\left\lceil \frac{\ell}{n} \right\rceil + \left\lceil \frac{\ell}{s} \right\rceil \right) + \mathsf{T}_{\mathsf{GHASH}} \cdot \left(\left\lceil \frac{\ell}{n} \right\rceil + 2 \left\lceil \frac{\ell}{s} \right\rceil \right) + \mathsf{T}^{v_e}_{\mathsf{rand}} \cdot \left\lceil \frac{\ell}{s} \right\rceil =$$
$$(\mathsf{T}_{\mathsf{AES}} + \mathsf{T}_{\mathsf{GHASH}}) \cdot \left\lceil \frac{\ell}{n} \right\rceil + (\mathsf{T}_{\mathsf{AES}} + 2\mathsf{T}_{\mathsf{GHASH}} + \mathsf{T}^{v_e}_{\mathsf{rand}}) \cdot \left\lceil \frac{\ell}{s} \right\rceil .$$

Putting it all together, we arrive at a total message encryption cost of:

$$\mathsf{T}_{\mathsf{FLOE\text{-}GCM}}(\ell, r, s) = \mathsf{T}_{\mathsf{F}} \cdot \left\lceil \frac{\ell}{s2^r} \right\rceil + (\mathsf{T}_{\mathsf{AES}} + \mathsf{T}_{\mathsf{GHASH}}) \cdot \left\lceil \frac{\ell}{n} \right\rceil + (\mathsf{T}_{\mathsf{AES}} + 2\mathsf{T}_{\mathsf{GHASH}} + \mathsf{T}^{v_e}_{\mathsf{rand}}) \cdot \left\lceil \frac{\ell}{s} \right\rceil . \quad (2)$$

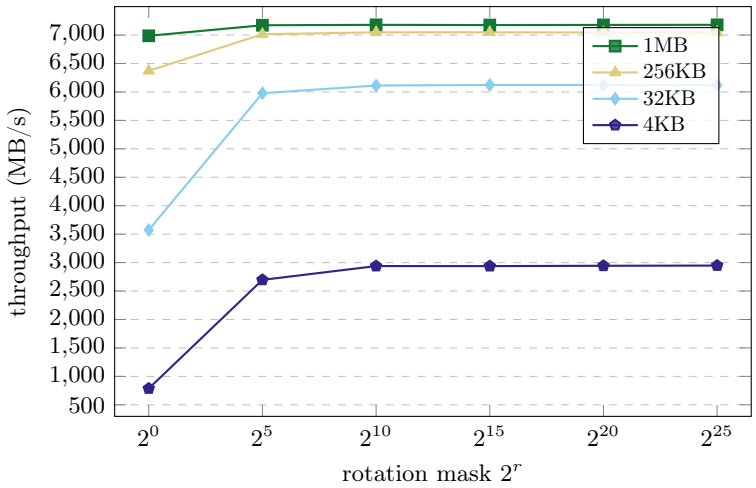

Fig. 15. Throughput in millions of bytes per second when processing an input message of length 1 GB, under various rotation masks and segment sizes. Higher is better.

Notice that the middle term is independent of r and s, and thus the first and third terms correspond to the added overhead from varying r and s. In practice, however, it will generally be the case that $n \ll s$ and $n \ll 2^r$, and so the middle term dominates as ℓ increases.

Experimental evaluation: Equation 2 provides a general template for gauging the overhead of FLOE-GCM on arbitrary platforms. To get a better sense of how it translates to overhead in practice, we now report on an empirical evaluation of FLOE-GCM.

We implemented FLOE-GCM in C++ on top of OpenSSL [28] for AES-GCM and HKDF-Expand, and evaluated our implementation on an Apple Macbook Pro with an M1 Pro chip for various values of r and s. (We discuss our implementations further in the section.) For each parameter set, we first performed a warm up of 500ms, followed by five iterations of encrypting a message of length 1GB. For each iteration, we computed the time required to encrypt the entire message (encryption start, followed by all segment encryptions), from which we derived the throughput in terms of millions of bytes per second (MB/s). We then report the median value across all five iterations. We use throughput as opposed to the more standard metric of cycles-per-byte since Apple does not provide access to cycle counters for an M1 Pro chip. Nevertheless, throughput is sufficient to showcase trends in performance.

We show the results of our experiments in Figure 15. We consider segment sizes of 4KB, 32KB, 256KB, and 1MB, and rotation masks of 0, 5, 10, 15, 20, and 25. As expected from Equation 2, a small rotation mask of $r = 1$ leads to a noticeable decrease in throughput across all segment sizes, since a large number of key derivations are required (e.g., 262,144 keys when $s = 4$ KB). However, the number of key derivations quickly decreases as we increase r (e.g., 8,192 keys when $r = 5$ and $s = 4$ KB), and this overhead is drowned by the cost of processing the plaintext. Similarly, decreasing the segment size also leads to noticeable overhead for small sizes, which primarily comes from deriving more random nonces. However, like key rotations, this quickly plateaus: as segments become larger and fewer nonces are required, the cost of encrypting all plaintext blocks dominates.

Implementation. Snowflake provides open-source reference implementations of FLOE-GCM in C++, Go, Java, and Rust at `https://github.com/Snowflake-Labs/floe-specification`. In addition, a developer-friendly specification documenting the usage of FLOE-GCM can be found at `blob/main/spec/README.md`.

Acknowledgements

This work was funded in part by NSF grants CNS-2120651 and CNS-2427390, as well Google's Cyber NYC program. We also thank Julien Boeuf, Kevin Dyer, Mike Halcrow, and Benedikt Schmidt of Snowflake for their useful discussions and tweaks to FLOE.

References

1. AES-CTR HMAC Streaming AEAD. https://developers.google.com/tink/streaming-aead/aes_ctr_hmac_streaming
2. AES-GCM-HKDF Streaming AEAD. https://developers.google.com/tink/streaming-aead/aes_gcm_hkdf_streaming
3. Tink Cryptographic Library. https://developers.google.com/tink
4. Backendal, M., Clermont, S., Fischlin, M., Günther, F.: Key Derivation Functions Without a Grain of Salt . In: EUROCRYPT (2025)
5. Barker, E., Roginsky, A., Davis, R.: NIST SP 800-133 Rev. 2: Recommendation for Cryptographic Key Generation. https://csrc.nist.gov/pubs/sp/800/133/r2/final (2020)
6. Bäumer, F., Brinkmann, M., Schwenk, J.: Terrapin Attack: Breaking SSH Channel Integrity By Sequence Number Manipulation. In: USENIX Security (2024)
7. Bellare, M., Hoang, V.T.: Efficient Schemes for Committing Authenticated Encryption. In: EUROCRYPT (2022)
8. Bernstein, D.J.: Stronger Security Bounds for Wegman-Carter-Shoup Authenticators. In: EUROCRYPT (2005)
9. Boldyreva, A., Degabriele, J.P., Paterson, K.G., Stam, M.: On Symmetric Encryption with Distinguishable Decryption Failures. In: Fast Software Encryption (2013)
10. Chan, J., Rogaway, P.: On Committing Authenticated Encryption. In: ESORICS. pp. 275–294 (2022)
11. Chen, L.: NIST SP 800-108 Rev. 1: Recommendation for Key Derivation Using Pseudorandom Functions. https://csrc.nist.gov/pubs/sp/800/108/r1/upd1/final (2022)
12. Coron, J.S., Patarin, J., Seurin, Y.: The Random Oracle Model and the Ideal Cipher Model are Equivalent. In: CRYPTO (2008)
13. Dworkin, M.: NIST SP 800-38B: Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication. https://csrc.nist.gov/pubs/sp/800/38/b/upd1/final (2005)
14. Grubbs, P., Lu, J., Ristenpart, T.: Message Franking via Committing Authenticated Encryption. In: CRYPTO (2017)
15. Gueron, S.: Double nonce derive key aes-gcm (dndk-gcm). Tech. rep., Internet-Draft draft-gueron-cfrg-dndkgcm-01, Internet Engineering Task Force (2024)
16. Gueron, S., Ristenpart, T.: DNDK: Combining Nonce and Key Derivation for Fast and Scalable AEAD. Cryptology ePrint Archive (2025)
17. Hoang, V.T., Reyhanitabar, R., Rogaway, P., Vizár, D.: Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance. In: CRYPTO (2015)
18. Hoang, V.T., Shen, Y.: Security of Streaming Encryption in Google's Tink Library. In: CCS (2020)
19. Hoang, V.T., Tessaro, S., Thiruvengadam, A.: The Multi-user Security of GCM, Revisited: Tight Bounds for Nonce Randomization. In: ACM CCS (2018)
20. Iwata, T., Ohashi, K., Minematsu, K.: Breaking and Repairing GCM Security Proofs. In: CRYPTO (2012)
21. Kampanakis, P., Campagna, M., Crocket, E., Petcher, A., Gueron, S.: Practical Challenges with AES-GCM and the need for a new cipher. In: Third NIST Workshop on Block Cipher Modes of Operation (2024)
22. Kelsey, J., jen Chang, S., Perlner, R.: NIST SP 800-108: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash. https://csrc.nist.gov/pubs/sp/800/185/final (2016)
23. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Feb 1997). https://doi.org/10.17487/RFC2104, https://www.rfc-editor.org/info/rfc2104
24. McGrew, D., Viega, J.: The Galois/Counter Mode of Operation (GCM). submission to NIST Modes of Operation Process (2004)
25. Morris Dworkin: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. hhttps://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf (2007)
26. Mouha, N., Luykx, A.: Multi-Key Security: The Even-Mansour Construction Revisited. In: CRYPTO (2015)
27. National Institute of Standards and Technology, Canadian Centre for Cyber Security: Implementation Guidance for FIPS 140-3 and the Cryptographic Module Validation Program. https://csrc.nist.gov/csrc/media/Projects/cryptographic-module-validation-program/documents/fips%20140-3/FIPS%20140-3%20IG.pdf (2019)
28. OpenSSL Authors: Openssl (2025), https://github.com/openssl/openssl

A Additional Definition of Correctness for raAE Schemes

In Section 4, we defined correctness for raAE schemes based on correctness of decrypting individual segments (Definition 1). This is based on the observation that, in the random access setting, correctness of decrypting individual ciphertext segments is equivalent correctness of decrypting the entire segmented ciphertext. We now make this concrete, by introducing a correctness definition in terms of segmented ciphertexts, and showing that it is equivalent to Definition 1.

Definition 3. *An raAE scheme* Π *is correct if, for any key* K*, nonce* N*, global AD* G*, segmented message* $\boldsymbol{M}$*, segmented AD* $\boldsymbol{A}$*, segmented ciphertext* $\boldsymbol{C}$*, positive integer* n*, subset-permutation* I *of* $[1, n]$*, and subset-permutation* J *of* I*, it holds that*

$$(T_g, S) \leftarrow \Pi.\mathsf{StartEnc}(K, N, G)\ ;\ T_g \parallel \big(\mathsf{EncSeg}(S, (i, (i = n)), A_i, M_i)\big)_{i \in I} = \boldsymbol{C}$$
$$\implies S \leftarrow \mathsf{StartDec}(K, N, A, T_g)\ ;\ \big(\mathsf{DecSeg}(S, (j, (j = n)), A_i, C_i)\big)_{j \in J} = (M_j)_{j \in I}.$$

Note how this definition is parametrized by subset-permutations I and J, which specify the order of segment encryptions and decryptions, respectively. We now show that Definition 3 is equivalent Definition 1. Intuitively, if a scheme supports random-access decryption, correctness of decrypting a ciphertext segment cannot depend on correctness of decrypting another ciphertext segment.

Theorem 9. *A raAE* Π *is correct according to Definition 3 if and only if* Π *is correct according to Definition 1.*

Proof. Say that we have K, N, G, $\mathbf{M} = (M_1, ..., M_n)$, $\mathbf{C} = (C_1, ..., C_n)$, I and J that fails the conditions of Definition 3. If so, there must be some $j \in J$ such that $\mathsf{DecSeg}(S, j, (j = n), A_i, C_i) \neq M_j$. Importantly, S is fixed across all segment decryptions, and so $(K, N, G, (j, (j = n)), M_j, A_j, C_j)$ would also not satisfy the conditions in Definition 1. For the converse, say that there is some $(K, N, G, (i, b), M_i, A_i, C_i)$ that fails the conditions of Definition 1. These same parameters would not satisfy Definition 3, simply by setting $\mathbf{M} = M_i$, $\mathbf{A} = A_i$, $\mathbf{C} = C_i$, and $I = J = [i]$.

B Additional Details on Relations Between raAE and coAE

In this section, we present the full proofs and additional details for the results of Section 5.

B.1 Proof that ra-ROR implies nOAE2

We show the proof of Theorem 1 in this section.

Proof. Adversary $\mathcal{B}$ runs $\text{nOAE2}^{\mathcal{A}}_{\mathsf{R2C}[\Pi]}$, using the oracles from its ra-ROR_Π game to simulate the environment and reply to queries from $\mathcal{A}$, as follows.

Initialization of keys. First, before any inputs from $\mathcal{A}$, $\mathcal{B}$ calls the NEW oracle in its game once for every key initialized in the setup phase of $\text{nOAE2}^{\mathcal{A}}_{\mathsf{R2C}[\Pi]}$.

Start of encryption streams. On every $\text{INITENC}(i, N)$ oracle call from $\mathcal{A}$, $\mathcal{B}$ calls $\mathsf{StartEnc}(i, N, \varepsilon)$, and returns an integer corresponding to the number of encryption streams that have been initialized for key instance i, which $\mathcal{B}$ keeps track of locally. Internally in ra-ROR_Π, in the real world, this results in a call to $\Pi.\mathsf{StartEnc}(K_i, N, \varepsilon)$, which matches the behavior of $\mathsf{R2C}[\Pi].\mathsf{Enc.init}$.

Encryption of non-terminal segments. On every $\text{NEXTENC}(i, s, A, M)$ oracle call from $\mathcal{A}$, $\mathcal{B}$ calls $\text{ENCSEG}(i, s, (j, 0), A, M)$ and forwards the resulting ciphertext C to $\mathcal{A}$, where j is the number of calls of the form $\text{NEXTENC}(i, s, \cdot, \cdot)$ made by $\mathcal{A}$ thus far, which $\mathcal{B}$ keeps track of locally. In the real world of ra-ROR_Π, this results in a call to $\Pi.\mathsf{EncSeg}(S_{i,s}, (j, 0), A, M)$, which matches the behavior of $\mathsf{R2C}[\Pi].\mathsf{Enc.next}$.

Encryption of terminal segments. This is analogous to the encryption of non-terminal segments mentioned above, expect that calls from $\mathcal{B}$ are now of the form $\text{ENCSEG}(i, s, (j, 1), A, M)$.

Decryptions. On every $\text{DEC}(i, N, \mathbf{A}, \mathbf{C}, \mathbf{I}, a)$ oracle call from $\mathcal{A}$, $\mathcal{B}$ will do as follows. First, $\mathcal{B}$ checks whether $|\mathbf{A}| \neq |\mathbf{C}|$, returning false if so. Otherwise, $\mathcal{B}$ computes T_g corresponding to this query, as follows. If there was a previous query of the form $\mathsf{StartEnc}(i, N, \varepsilon)$, $\mathcal{B}$ simply uses the result of that query. Otherwise, $\mathcal{B}$ queries $\mathsf{StartEnc}(i, N, \varepsilon)$. Then, using T_g, $\mathcal{B}$ calls $\text{STARTDEC}(i, N, \varepsilon, T_g)$. In the real world of ra-ROR_Π, this results in a call to $\Pi.\mathsf{StartDec}(K_i, N, \varepsilon, T_g)$, which matches the behavior of $\mathsf{R2C}[\Pi].\mathsf{Dec.init}$.

Then, $\mathcal{B}$ checks that every index in $\mathbf{I}$ is between 1 and $|\mathbf{C}|$, returning false if not. If this check passes, $\mathcal{B}$ then iterates over every $r \in [1, |\mathbf{I}|]$, doing as follows. If $a = 0$ or r is such that $\mathbf{I}[r] \neq |\mathbf{C}|$, $\mathcal{B}$ calls $\text{DECSEG}(i, d, (\mathbf{I}[r], 0), \mathbf{A}[r], \mathbf{C}[r])$, where d is the number of calls of the form $\text{DEC}(i, \cdot)$ made by $\mathcal{A}$ thus far, which $\mathcal{B}$ keeps track of locally. If any of these calls return $\perp$, $\mathcal{B}$ returns false to $\mathcal{A}$. In the real world of ra-ROR_Π, each of these calls results in a call to $\Pi.\mathsf{DecSeg}(S_{i,d}, (\mathbf{I}[r], 0), \mathbf{A}[r], \mathbf{C}[r])$, which matches the behavior of $\mathsf{R2C}[\Pi].\mathsf{Dec.next}$. Conversely, if $a = 1$ and r is such that $\mathbf{I}[r] = |\mathbf{C}|$, $\mathcal{B}$ will instead call $\text{DECSEG}(i, d, (\mathbf{I}[r], 1), \mathbf{A}[r], \mathbf{C}[r])$, returning false if this oracle call fails. If no $\perp$ is returned at the end,

$\underline{\mathsf{STREAM}^*.\mathsf{Kg}:}$
$K \xleftarrow{\$} \{0,1\}^\kappa$
return K

$\underline{\mathsf{STREAM}^*.\mathsf{StartEnc}(K,N,G):}$
$(1,K,N) \leftarrow \mathsf{STREAM.Enc.init}(K,N)$
$S \leftarrow (K,N)$
$T_g \leftarrow \varepsilon$
return (T_g, S)

$\underline{\mathsf{STREAM}^*.\mathsf{EncSeg}(S,p,M):}$
$(K,N) \leftarrow S\ ;\ (i,b) \leftarrow p$
$\tilde{S} \leftarrow (i,K,N)$
if $b = 0$:
  $(C, \tilde{S}) \leftarrow \mathsf{STREAM.Enc.next}(\tilde{S}, A, M)$
else :
  $C \leftarrow \mathsf{STREAM.Enc.last}(\tilde{S}, A, M)$
return C

$\underline{\mathsf{STREAM}^*.\mathsf{StartDec}(K,N,G,T_g):}$
$(1,K,N) \leftarrow \mathsf{STREAM.Dec.init}(K,N)$
$S \leftarrow (K,N)$
return S

$\underline{\mathsf{STREAM}^*.\mathsf{DecSeg}(S,p,C):}$
$(K,N) \leftarrow S\ ;\ (i,b) \leftarrow p$
$\tilde{S} \leftarrow (i,K,N)$
if $b = 0$:
  $(M, \tilde{S}) \leftarrow \mathsf{STREAM.Dec.next}(\tilde{S}, A, C)$
else :
  $M \leftarrow \mathsf{STREAM.Dec.last}(\tilde{S}, A, C)$
return M

Fig. 16. Scheme STREAM^*, which casts STREAM in the syntax of raAE.

$\mathcal{B}$ returns true to $\mathcal{A}$. By an analysis analogous to the one for non-terminal segments, this matches the behavior of $\mathsf{R2C}[\Pi].\mathsf{Dec.last}$.

Lastly, we analyze whether simulating the DEC oracle requires calls that result in trivial wins in ra-ROR but not nOAE2. Trivial wins in ra-ROR result from calls to $\textsc{DecSeg}(i,s,(j,b),A,C)$ where C was the output of some query of the form $\textsc{EncSeg}(i,s,(j,b),A,M)$. So, if a call to DEC from $\mathcal{A}$ requires an underlying DECSEG call from $\mathcal{B}$ that leads a trivial win in ra-ROR, $\mathcal{B}$ simply skips this call and continues iterating through ciphertext segments, interpreting this call as if its output was different from $\perp$. If this is required for all queries, then this is also a trivial win in nOAE2, so $\mathcal{A}$ would not be allowed to make a query of this form to begin with.

B.2 STREAM in the Syntax of raAE

While STREAM is defined as a coAE scheme, we can directly cast it as an raAE scheme. We call this scheme STREAM^*, and show its specification in Figure 16.

B.3 Proof of STREAM's ra-ROR Security

We show the proof of Theorem 2 in this section.

Proof. We build adversary $\mathcal{B}$ as follows. Adversary $\mathcal{B}$ runs $\text{ra-Real}_\Pi^{\mathcal{A}}$, using the oracles from its $\text{mu-ROR}_{\mathsf{AE}}$ game to reply to queries from $\mathcal{A}$. We leave implicit that, on every query from $\mathcal{A}$, $\mathcal{B}$ first verifies whether the inputs are valid, returning $\perp$ if not.

First, whenever $\mathcal{A}$ queries the NEW oracle, $\mathcal{B}$ queries the NEW oracle in its game accordingly. Since encryption keys for STREAM^* correspond to keys for AE, this ensures that key instances in both games remain synchronized.

Next, whenever $\mathcal{A}$ queries $\textsc{StartEnc}(i,N,G)$, $\mathcal{B}$ stores $((i,e_i),N)$ in a key-value store T and increments e_i, where e_i is a counter that keeps track of the number of encryption streams started for each key index thus far. Then, $\mathcal{B}$ returns ε to $\mathcal{A}$, since recall that STREAM^* is a headerless scheme.

Turning to segment encryptions, on every $\textsc{EncSeg}(i,j,p,A,M)$ call from $\mathcal{A}$, $\mathcal{B}$ makes oracle call $\textsc{Enc}(i,N^*,A,M)$ in its $\text{mu-ROR}_{\mathsf{AE}}$ game, where nonce N^* is computed by retrieving $N \leftarrow T[(i,j)]$ and appending $(i,b) \leftarrow p$ to it to obtain $N^* \leftarrow N \,\|\, i \,\|\, b$.

Decryptions are handled analogously: decryption starts keep track of nonces across key instances and decryption streams in a key-value store, which are later retrieved during segment decryptions to generate the correct input nonce to AE.

Lastly, since $\mathcal{A}$ is nonce-respecting, $\mathcal{B}$ is also nonce-respecting.

B.4 Proof of STREAM's Commitment Security

We show the proof of Theorem 3 in this section.

Proof. This follows from a standard reduction. Adversary $\mathcal{B}$ runs ra-CMT$^{\mathcal{A}}_{\Pi}$, and receives a ciphertext segment C alongside decryption contexts $(\varepsilon, (i,b), K_1, N_1, A_1)$ and $(\varepsilon, (i,b), K_2, N_2, A_2)$ from $\mathcal{A}$. (Notice that $\mathcal{A}$ does not output T_g, G_1, nor G_2, since STREAM* is headerless and does not use global ADs.) Then, in its game, $\mathcal{B}$ submits challenge ciphertext C and decryption contexts $(K_1, N_1 \parallel i \parallel b, A_1)$ and $(K_2, N_2 \parallel i \parallel b, A_2)$.

In CMT$_{\mathsf{AE}}$, this results in $M'_1 \leftarrow \mathsf{AE.Dec}(K_1, N_1 \| i \| b, A_1, C)$ and $M'_2 \leftarrow \mathsf{AE.Dec}(K_1, N_2 \| i \| b, A_2, C)$. In ra-CMT$_{\Pi}$, the outputs from $\mathcal{A}$ result in $S \leftarrow \Pi.\mathsf{StartDec}(K_1, N_1, \varepsilon, \varepsilon)$ followed by $\Pi.\mathsf{DecSeg}(S, (i,b), A_1, C)$, which in turns results in $M''_1 \leftarrow \mathsf{AE.Dec}(K_1, N_1 \parallel i \parallel b, A_1, C)$; and respectively for the second decryption context to obtain M''_2. Therefore, by the correctness of AE, it follows that $M'_1 = M''_1$ and $M'_2 = M''_2$, and so we arrive at the desired bound.

C Proof of FLOE's Confidentiality and Authenticity

We show the full proof of Theorem 4 in this section.

Proof. Our proof uses a game-hopping argument. Let G0 be equal to ra-Real$^{\mathcal{A}}_{\Pi}$. We begin by addressing encryption and decryption starts. We define G1 as G0, but replacing computations of F in $\Pi.\mathsf{StartEnc}$ and $\Pi.\mathsf{StartDec}$ during intermediate key derivations and header computations by calls (with an appropriate key index) to a random function with the same domain and range as F. Notice that, since F is domain-separated for intermediate key derivations and header computations, each of which have a fixed output length, no input to F is queried under more than one output length. By a standard reduction, there exists an adversary $\mathcal{B}_0$ such that

$$|\Pr[\,\mathrm{G1} = 1\,] - \Pr[\,\mathrm{G0} = 1\,]| \leq \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}_0).$$

Adversary $\mathcal{B}_0$ runs G1 but, in calls to StartEnc and StartDec, replaces all computations of $\mathsf{F}(K_i, \rho \| N \| \mathsf{c_1} \| G, \ell_{T_g})$ and $\mathsf{F}(K_i, \rho \parallel N \parallel \mathsf{c_2} \parallel G, \kappa_{in})$ with calls to its evaluation oracle for this key index in its own KDF$_{\mathsf{F}}$ game. Furthermore, any calls to NEW in G1 also result in calls to NEW in KDF$_{\mathsf{F}}$.

Recall that $\mathcal{A}$ is assumed to be nonce respecting. Therefore, K_{in} and T_g in G1 are computed via random functions on unique points. So, we can define G2 as G1, but directly sampling fresh random strings of the appropriate lengths for K_{in} and T_g. It follows that

$$\mathrm{G2} \approx \mathrm{G1}$$

where "$\approx$" denotes that both random variables are identically distributed.

Since all K_{in} are now random, we next define G3 as G2 but replacing calls to F in $\Pi.\mathsf{EncSeg}$ and $\Pi.\mathsf{DecSeg}$ during AEAD key derivations with calls (with an appropriate key index) to a random function with the same domain and range as F. Like before, there exists an adversary $\mathcal{B}_1$ such that

$$|\Pr[\,\mathrm{G3} = 1\,] - \Pr[\,\mathrm{G2} = 1\,]| \leq \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}_1).$$

Adversary $\mathcal{B}_1$ runs G3, and operates analogously to $\mathcal{B}_0$, but with computations of K_{se} instead of K_{in}. Note that, since all K_{in} are random in G2, they are distributed exactly like the keys in $\mathcal{B}_1$'s (real) game.

Then, like before, we can use the fact that $\mathcal{A}$ is nonce respecting to define a new game G4 which replaces computations of K_{se} via random functions by sampling fresh random strings of the appropriate length. Since inputs to the random functions in G3 are unique, it follows that

$$\mathrm{G4} \approx \mathrm{G3}.$$

Next, we define G5 as G4 but replacing AEAD ciphertexts in $\Pi.\text{EncSeg}$ with random strings of the appropriate length, and $\Pi.\text{DecSeg}$ always returning $\perp$. By a standard reduction, there exists an adversary $\mathcal{C}$ such that

$$|\Pr[\,\mathrm{G5} = 1\,] - \Pr[\,\mathrm{G4} = 1\,]| \leq \mathrm{Adv}^{\text{mu-ror\$}}_{\mathsf{AE}}(\mathcal{C}).$$

Adversary $\mathcal{C}$ runs G5 but, in calls to EncSeg, computes all ciphertext segments by first computing the appropriate segment ciphertext header H and segment associated data A based on the value of p, followed by calls to the encryption oracle in its own mu-ROR\$$_{\mathsf{AE}}$ game for the appropriate key index.

Then, in calls to DecSeg, $\mathcal{C}$ computes the segment associated data A based on p, followed by calls to the decryption oracle in its game. Lastly, derivations of K_{se} in G5 additionally trigger calls to New in mu-ROR$\$_{\mathsf{AE}}$. Note that AEAD keys in G5 are uniformly random strings, and so they are distributed exactly like the keys in $\mathcal{D}$'s (real) game.

Lastly, notice that the only difference between ra-Rand$^{\mathcal{A}}_{\Pi}$ and G5 is that the former always returns $\perp$ in StartDec, while the latter verifies ciphertext tag T_g instead, by comparing it against a random string of the same length (since, recall, G5 replaces F computations with sampling random strings). Therefore, since trivial wins prevent the adversary from using tags output by previous StartEnc queries, these games are only distinguishable when the adversary queries StartDec with a new T_g that collides with a fresh random string of length ℓ_{T_g}, which happens with probability $2^{-\ell_{T_g}}$. By a union bound across all StartDec queries, we get that G5 is distinguishable from ra-Rand$^{\mathcal{A}}_{\Pi}$ with probability $q_{\mathsf{sd}} \cdot 2^{-\ell_{T_g}}$

Putting all game hops together, we get that

$$\mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}) + \mathrm{Adv}^{\text{mu-ror\$}}_{\mathsf{AE}}(\mathcal{C}) + q_{\mathsf{sd}} \cdot 2^{-\ell_{T_g}},$$

as desired. Adversary $\mathcal{B}$ simply samples a random bit i and runs adversary $\mathcal{B}_i$; by a standard reduction $\mathcal{B}$ satisfies that

$$\mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}_0) + \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}_1) \leq 2 \cdot \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}).$$

D Proof of FLOE-GCM's Confidentiality and Authenticity

We show the full proof of Theorem 6 in this section.

Proof. We can use the bound from Theorem 5 to instantiate the $\mathrm{Adv}^{\text{mu-ror\$}}_{\mathsf{AE}}(\mathcal{B})$ term in Theorem 4. Recall that the advantage bound of Theorem 5 is as follows:

$$\mathrm{Adv}^{\text{mu-ror\$}}_{\mathsf{AE}}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\text{mu-prp}}_{\mathsf{E}}(\mathcal{C}) + \sum_{i=1} \left(\frac{\vec{q_i} \cdot (iL_1 + i + 1)^2}{2^{n+1}} + \frac{\vec{q_i} \cdot i \cdot (i-1)}{2^{97}} \right) + \frac{d(L_2+1)}{2^{\tau}} \cdot \delta_n(w(\vec{q}) \cdot (L_1+1) + d + 1).$$

By design, query profile $\vec{f}$ of ra-ROR adversary $\mathcal{A}$ directly maps to the $\vec{q}$-bound notion in Theorem 5, and so $\vec{q_i} = \vec{f_i}$ where $i \in [1, 2^r]$. Then, since we deal with a fixed message segment size s, it follows that $L_1 \leq s/n$ and $L_2 \leq 9 + s/n$, where the 9 correponds to the length of the segment ADs. Lastly, we can use the bound from [26] for the PRP security of E in the ideal cipher model:

$$\mathrm{Adv}^{\text{mu-prp}}_{\mathsf{E}}(\mathcal{C}) \leq \frac{q_{\mathsf{e}}p + q_{\mathsf{e}}(q_{\mathsf{e}}-1)}{2^{\kappa_{se}}}.$$

Plugging these pieces into Theorem 4, we get that

$$\mathrm{Adv}^{\text{ra-ror}}_{\Pi}(\mathcal{A}) \leq 2 \cdot \mathrm{Adv}^{\text{mu-prf}}_{\mathsf{F}}(\mathcal{B}) + \frac{q_{\mathsf{e}}p + q_{\mathsf{e}}(q_{\mathsf{e}}-1)}{2^{\kappa_{se}-1}} + \sum_{i \in [1,2^r]} \frac{\vec{f_i} \cdot (is/n + i + 1)^2}{2^{n+1}} + \sum_{i \in [1,2^r]} \frac{\vec{f_i} \cdot i \cdot (i-1)}{2^{97}} + \frac{q_{\mathsf{d}}(s/n+10)}{2^{\tau}} \cdot \delta_n(w(\vec{q}) \cdot (s/n+1) + q_{\mathsf{d}} + 1) + \frac{q_{\mathsf{sd}}}{2^{\ell_{T_g}}}$$

as desired.

E Proof of FLOE's Commitment

We show the full proof of Theorem 7 in this section.

Proof. Let $\mathcal{A}$ be the ra-CMT$_{\Pi}$ adversary. We show how to construct a CR$_{\mathsf{F}}$ adversary $\mathcal{B}$ against F. $\mathcal{A}$ returns $\big(T_g, C, (G_1, p, K_1, N_1), (G_2, p, K_2, N_2)\big)$. Notice that, since this is for positionless commitment, the value p in both contexts is the same. Furthermore, since FLOE does not take as input associated data segments for DecSeg, the adversary $\mathcal{A}$ excludes them from the decryption contexts it returns.

Error	Symbol	Algorithm	Description	Condition
Header length	e_{hdr_ℓ}	StartDec	Ciphertext header does not have expected length	$\lvert T_g\rvert \neq \ell_{T_g}$
Header value	e_{hdr_v}	StartDec	Ciphertext header is not valid for input key, nonce, global AD, and parameters	$T_g \neq \mathsf{F}(K, \rho \parallel N \parallel \mathsf{c}_1 \parallel G, \ell_{T_g})$
Max. segments	$e_{\max}$	DecSeg	Ciphertext segment number is beyond maximum allowed	$b = 0$ and $i \geq \max_s - 1$ or $b = 1$ and $i \geq \max_s$
Non-terminal segment length	e_{nt_ℓ}	DecSeg	Non-terminal ciphertext segment does not have expected length	$b = 0$ and $\lvert H\rvert + \lvert C\rvert \neq 4 + v_e + s + t$
Terminal segment length (short)	$e_{\mathrm{t}_{\ell s}}$	DecSeg	Terminal ciphertext segment is shorter than allowed	$b = 1$ and $\lvert H\rvert + \lvert C\rvert < 4 + v_e + t$
Terminal segment length (long)	$e_{\mathrm{t}_{\ell l}}$	DecSeg	Terminal ciphertext segment is longer than allowed	$b = 1$ and $\lvert H\rvert + \lvert C\rvert > 4 + v_e + s + t$
Non-terminal segment header	e_{nt_h}	DecSeg	Header of non-terminal ciphertext segment does not match expected value	$b = 0$ and $H \neq \mathsf{c}_4$
Terminal segment header	e_{t_h}	DecSeg	Header of terminal ciphertext segment does not match expected value	$b = 1$ and $H \neq \mathsf{encode}(\lvert C\rvert, 4)$
AEAD decryption failure	e_{dec}	DecSeg	AEAD decryption of ciphertext segment failed	$\mathsf{AE.Dec}(K_{se}, A, C) = \perp$

Fig. 17. Description of error symbols that may be returned by $\mathsf{FLOE}^{\mathsf{err}}$, the variant of FLOE with an expanded set of error symbols.

For $\mathcal{A}$ to win, it must be the case that both calls to DecSeg in the ra-CMT game return valid decryptions, which also implies that both calls to StartDec must not return $\perp$. This then means that we have

$$T_g = \mathsf{F}(K_1, \rho \parallel N_1 \parallel \mathsf{c}_1 \parallel G_1, \ell_{T_g}) = \mathsf{F}(K_2, \rho \parallel N_2 \parallel \mathsf{c}_1 \parallel G_2, \ell_{T_g})\,.$$

We thus construct adversary $\mathcal{B}$ as follows. $\mathcal{B}$ runs $\text{ra-CMT}^{\mathcal{A}}_{\Pi}$, and receives the outputs as specified above. It then returns $(K_1, K_2, (\rho \parallel N_1 \parallel \mathsf{c}_1 \parallel G_1, \ell_{T_g}), (\rho \parallel N_2 \parallel \mathsf{c}_1 \parallel G_2, \ell_{T_g}))$. Notice that when $\mathcal{A}$ wins game ra-CMT_{Π}, then $\mathcal{B}$ wins game CR_{F}, thus completing the proof. □

F FLOE with Multiple Error Messages

Recall from Section 6 that a usability goal for FLOE is to output useful error messages on failed decryption, without introducing security issues. We discuss security under multiple error messages in this section. First, we extend the syntax of raAE schemes to allow for returning more than one error symbol during StartDec and DecSeg. We then introduce $\mathsf{FLOE}^{\mathsf{err}}$, a variant of FLOE that supports multiple error symbols. Lastly, we extend ra-ROR and ra-CMT-x to capture (in)security in this new setting, and argue that $\mathsf{FLOE}^{\mathsf{err}}$ meets these definitions.

Multi-error raAE schemes. Recall from Section 4 that our syntax for raAE schemes allows the StartDec and DecSeg algorithms to return a single error symbol $\perp$. So, to capture schemes with multiple error symbols, we extend this syntax in the natural way, and define *multi-error raAE schemes* as identical to raAE schemes, but with an associated *error space* $\mathcal{E}$, and StartDec and DecSeg may now return any symbol $e \in \mathcal{E}$.

FLOE's error symbols. We refer to the multi-error variant of FLOE as $\mathsf{FLOE}^{\mathsf{err}}$. We outline $\mathsf{FLOE}^{\mathsf{err}}$'s error space $\mathcal{E}$ in Figure 17, including the algorithm in which each error is output (StartDec or DecSeg) and the condition that triggers it. $\mathsf{FLOE}^{\mathsf{err}}$ is identical to FLOE, but verifies all conditions shown in the last column of Figure 17, except for AEAD decryption failures, in order immediately at the start of FLOE.StartDec or FLOE.DecSeg, returning the appropriate error symbol if any error check is triggered. AEAD decryption failures are verified as the last step of FLOE.DecSeg before returning message M.

Notice that only e_{hdr_v} and e_{dec} depend on secret information (keys K and K_{se}, respectively). Therefore, all other error symbols can be computed directly from the inputs to the StartDec and DecSeg oracles, and thus are publicly computable. This will be relevant later on when arguing that $\mathsf{FLOE}^{\mathsf{err}}$ remains secure under this expanded set of error symbols.

ra-ROR **with multiple error symbols.** As currently formulated, ra-ROR does not capture the (in)security of multi-error raAE schemes. That is, any such scheme will trivially be insecure: the StartDec and DecSeg oracles in ra-Rand always return a single error symbol $\perp$, and so the adversary can simply submit inputs to either oracle that result in a different error symbol; the real world would return the

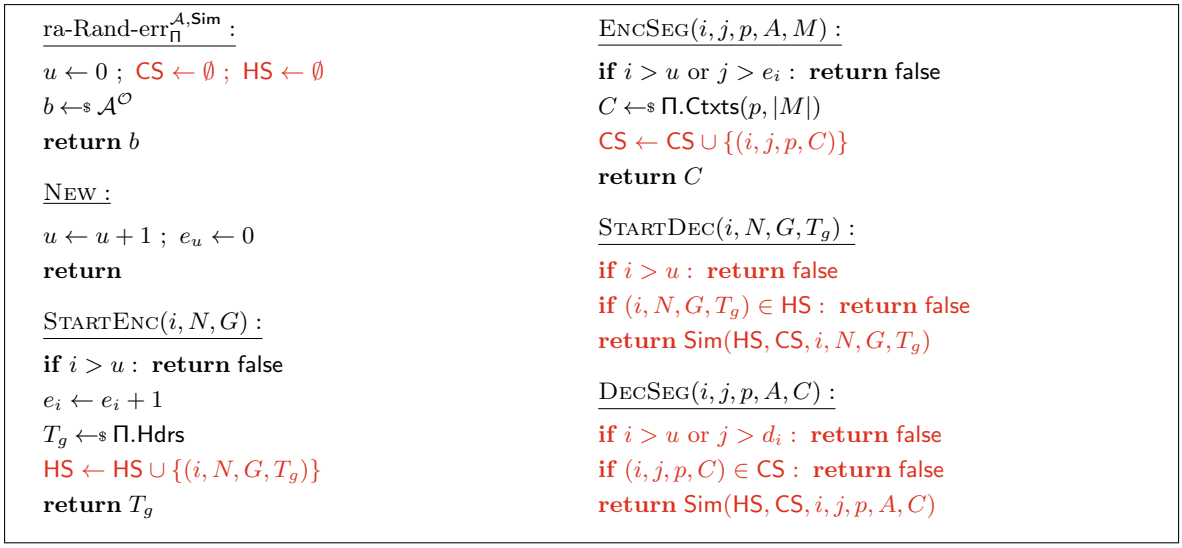

Fig. 18. Games for the ideal world of ra-ROR-err security of raAE schemes. Adversary $\mathcal{A}$ has access to oracles $\mathcal{O} = \{\text{New}, \text{StartEnc}, \text{EncSeg}, \text{StartDec}, \text{DecSeg}\}$. Changes with respect to the ideal world of ra-ROR are higlighted in red. The real world of ra-ROR-err is identical to that of ra-ROR.

correct error symbol, while the ideal world would always return $\perp$. As a concrete example, for $\mathsf{FLOE}^{\mathsf{err}}$, the adversary can call StartDec with a header T_g that is shorter than ℓ_{T_g}, which would return e_{hdr_ℓ} in the real world and $\perp$ in the ideal world.

We thus need to refine ra-ROR to account for multi-error raAE schemes. To do so, we simply generalize StartDec and DecSeg so that, instead of always returning $\perp$, they return the output of a *simulator* Sim that has access to the inputs and outputs from all oracle calls thus far (including the present oracle call). Thus, for a scheme to be secure, we require that there exists a Sim that is able to simulate the correct output error symbols given only the same knowledge as the adversary.

We refer to this variant of ra-ROR as ra-ROR-err, and show its ideal world in Figure 18, with the changes with respect to the ideal world of ra-ROR highlighted in red. The real world of ra-ROR-err is identical to that of ra-ROR. Notice how ra-Rand-err returns false exclusively if the inputs to StartDec or DecSeg trigger a trivial win (matching the real world), and instead defer production of the correct error symbol to Sim. Finally, we define the advantage of an ra-ROR-err adversary $\mathcal{A}$ over multi-error raAE scheme Π, with respect to simulator Sim, as

$$\mathrm{Adv}^{\text{ra-ror-err}}_{\Pi,\mathsf{Sim}}(\mathcal{A}) = \left|\Pr\left[\,\text{ra-Real-err}^{\mathcal{A}}_{\Pi} = 1\,\right] - \Pr\left[\,\text{ra-Rand-err}^{\mathcal{A}}_{\Pi,\mathsf{Sim}} = 1\,\right]\right| .$$

ra-ROR-err **security of** $\mathsf{FLOE}^{\mathsf{err}}$**.** Equipped with ra-ROR-err, we now argue that $\mathsf{FLOE}^{\mathsf{err}}$ meets this security definition, and thus FLOE remains secure when supporting multiple error messages.

The key idea is that, as mentioned earlier, all errors except for e_{hdr_v} and e_{dec} are publicly computable, and so Sim can just check the conditions in Figure 17 with the given oracle inputs to trigger the correct error symbol. If all checks pass, then Sim returns e_{hdr_v} in StartDec and e_{dec} in DecSeg. Finding an input to StartDec that does not return error symbol e_{hdr_v} in ra-Real-err requires forging an output of PRF F. Similarly, finding an input to DecSeg that does not return error symbol e_{dec} in ra-Real-err requires forging an AEAD ciphertext. Looking at FLOE's security bound in Theorem 4, header forgeries are captured by the game hops that replace F with random strings, and the probability of such strings colliding with $\mathcal{A}$'s input tags; and AEAD ciphertext forgeries are captured by the game hop that replaces AEAD ciphertexts with $\perp$. We capture these ideas in the following simple theorem:

Theorem 10 ($\mathsf{FLOE}^{\mathsf{err}}$ **confidentiality and authenticity).** *Let* AE *be an AEAD scheme,* F *be a PRF,* $\Pi = \mathsf{FLOE}^{\mathsf{err}}[\mathsf{AE}, \mathsf{F}]$ *and* $\Pi^* = \mathsf{FLOE}[\mathsf{AE}, \mathsf{F}]$ *both with associated parameter set* ρ*. Let* $\mathcal{A}$ *be a position-respecting* ra-ROR-err$_\Pi$ *adversary. There exists a* ra-ROR$_{\Pi^*}$ *adversary* $\mathcal{B}$ *and efficient simulator* Sim *such that*

$$\mathrm{Adv}^{\text{ra-ror-err}}_{\Pi,\mathit{Sim}}(\mathcal{A}) = \mathrm{Adv}^{\text{ra-ror}}_{\Pi^*}(\mathcal{B}).$$

ra-CMT-x **with multiple error symbols.** Just like ra-ROR, we also need to adapt ra-CMT-x to capture multi-error raAE schemes. The change here is simple: instead of requiring that decryption under both contexts returns a message different from $\perp$, we now require that both messages are outside the error space $\mathcal{E}$. Concretely, in the pseudocode in Figure 9, instead of verifying whether $M_1 = \perp$ or $M_2 = \perp$, we now check whether $M_1 \in \mathcal{E}$ or $M_2 \in \mathcal{E}$. We refer to this variant of ra-CMT-x as ra-CMT-err-x, and define the advantage of an adversary $\mathcal{A}$ over multi-error raAE scheme Π as

$$\mathrm{Adv}^{\text{ra-cmt-err-x}}_{\Pi}(\mathcal{A}) = \Pr\left[\text{ra-CMT-err-x}^{\mathcal{A}}_{\Pi} = 1\right]$$

Note that ra-CMT-err-x only verifies whether M_1 or M_2 is equal to *some* symbol in $\mathcal{E}$, returning 0 if so, irrespective of their specific value: even if M_1 and M_2 correspond to different symbols in $\mathcal{E}$, this would not constitute a win by $\mathcal{A}$. In other words, *commitment to error symbols* is outside the scope of ra-CMT-err-x. Exploring this notion is an interesting direction for future work, for both multi-error raAE schemes and other multi-error primitives more broadly. This would amount to defining a more fine-grained win condition than simply checking for (lack of) presence in $\mathcal{E}$.

ra-CMT-x **security of** $\mathsf{FLOE}^{\mathsf{err}}$**.** The positionless and position commitment bounds of $\mathsf{FLOE}^{\mathsf{err}}$ are equal to those of FLOE's, reducing to the collision resistance of F (Theorem 7) and the commitment security of AE (Theorem 8), respectively.