Flexible Authenticated Encryption

Sanketh Menda Julia Len Viet Tung Hoang Mihir Bellare Thomas Ristenpart

July 1, 2023

Abstract

We define and build a new type of AEAD scheme that we call flexible. Flexibility is intended as an answer to the growing list of desired security and performance features for future AEAD standards. Rather than a scheme per requirement, we offer a single scheme that flexibly incorporates multiple requirements, yet in a unified, systematic, and performance-optimal way. Mandatory for our definition are to provide classic unique-nonce AE security and, importantly and more novel, context commitment; then additionally to allow keys and nonces of arbitrary length. Beyond this, the scheme is configurable through an application-chosen input called a *configuration*. Via this input, one says what further or advanced security or performance attributes one wants; for example, misuse resistance, nonce-hiding, preservation of length, or parallelizability. The choice can be made dynamically and the scheme will provide the chosen set of attributes without changing the key. In providing a flexible scheme, we take a clean-slate approach. Our Flex scheme is built from a single permutation. Our implementations show that, for each configuration, the performance of Flex is competitive with that of current, dedicated schemes that directly and only provide the features named in that particular configuration.

1 Introduction

In this work we propose a new evolution of symmetric encryption, what we refer to as flexible AEAD. We think this will facilitate and enable upcoming standardization efforts. We give definitions for flexible AEAD as well as a concrete realization in the form of a scheme we simply call Flex. First we set the stage.

Emerging goals for AEAD. Recall that in a scheme for AEAD (Authenticated Encryption with Associated Data) [26], encryption takes key, nonce, associated data and message to deterministically return a ciphertext. The classical security requirement is unique-nonce AE (UNAE) security. This means privacy of the message, and authenticity of the message and associated data, assuming encryption never reuses a nonce. The NIST standard GCM is an example of an AEAD scheme.

Since the standardization of GCM, developers and researchers have identified a number of further, desirable security and operational attributes for AEAD. Security attributes include committing security [1, 3, 11, 14, 17, 18], misuse resistance [28] and AE2-security (also called nonce-hiding) [4]; operational attributes include parallelizability, robustness in the sense of [21] and support for arbitrary-length nonces and keys. These and other attributes are part of a comprehensive list in the IETF draft on properties of AEAD algorithms [9]. Let us now expand on (some of) these attributes and why they are desirable.

Committing security. A recent line of work has demonstrated the need for key commitment for AEAD schemes. Key commitment asks that it be hard to find a ciphertext that decrypts correctly under two (or more) different adversarially-chosen keys [17, 18]. Non-key-committing AEAD was first shown to be a problem for moderation in encrypted messaging [14, 18], and later in password-based encryption [24] and

envelope encryption [1], among others. These findings have pushed the cryptography community to begin proposing [1] and deploying new key-committing AEAD schemes [1, 2]. Indeed the recent IETF draft on properties of AEAD algorithms explicitly cites key commitment as a security goal [9].

Key commitment definitions don't model attacks in which adversaries exploit lack of commitment to the associated data or nonce. Bellare and Hoang [3] introduced the notion of *context commitment*, and recent work by Menda et al. [25] highlights that many AEAD schemes, including some that achieve key commitment, do not achieve context commitment. This motivates the need to expand our goal to context commitment.

Unfortunately, no currently standardized schemes achieve context commitment. This suggests we need to define and standardize new AEAD schemes.

Misuse resistance. Stipulating non-repeating nonces is easy in theory but harder to ensure in practice, where errors and misconfigurations have been reported to lead to repeating nonces. For many widely used and standardized UNAE AEAD schemes, this has led to damaging attacks [8]. Misuse-resistant AE (MRAE) [28] mitigates this by providing UNAE-security when nonces do not repeat, plus as good as possible security if they do. Standardization of an MRAE scheme is a desirable goal.

AE2 security. Embedded in the syntax and usage of AEAD is a weakness relating to the way nonces are handled. Namely, since the nonce is needed for decryption, it is sent in the clear unless the receiver already has it. But, as Bellare, Ng, and Tackmann [4] point out, some choices of nonces compromise message privacy (for both UNAE and MRAE) and others (like counters) compromise sender anonymity. AE2 (which comes in two forms, UNAE2 strengthening UNAE, and MRAE2 strengthening MRAE) hides the nonce, increasing privacy. This is important for anonymous AE [10]. AE2 emerges as another desirable attribute for a standard.

Robustness. The ciphertext expansion of a scheme is defined as the difference between ciphertext length and plaintext length. UNAE security requires some ciphertext expansion. (Usually 128 bits for GCM.) Some applications cannot permit this. An answer is robust AE [21], where one gets the best security possible with a given constraint on the ciphertext expansion.

Key and nonce lengths. The 96-bit nonce-length of GCM is viewed as a limitation because with random nonces it permits at most 2^{48} encryptions before a key change is needed. Schemes would ideally have either no maximum nonce length or a very large one. Similarly, different users or applications want different security levels and thus different key sizes. The need for post-quantum security is also pushing key sizes for symmetric cryptography up. Ideally, a scheme should handle keys of different and arbitrary lengths.

The challenge for standardization. One approach for standardization would be to standardize a different scheme for each of some choice of goals. However, the dimensions indicated above give rise to rather a lot of goals. Indeed, there are two choices for AE security (UNAE or MRAE), then a choice for committing security (yes or no) and another for AE2 (yes or no), already $2^3 = 8$ goals, with further possible choices for robustness, parallelizability, streaming support, and key and nonce lengths.

An alternative is to standardize a scheme for the strongest goal (for example, MRAE, committing and AE2) but this will mean that applications needing less stringent attributes will pay unnecessarily in cost. Indeed, one can't unfortunately achieve all the goals simultaneously while providing best-possible speed for each individual goal. Misuse resistance requires a full pass over the plaintext before the first bits of ciphertext can be produced, and known robust AE construction approaches require building a length-preserving enciphering scheme and then combining with the encode-then-encipher paradigm [5]. In either case you cannot have a scheme that is streaming, let alone parallelizable.

With flexibility, we suggest a different route.

Flexible AEAD. At a high level, our idea is to reformulate AEAD so that it takes as run-time input a configuration. The configuration, denoted cfg throughout, specifies an operating mode. But the same secret key can be safely used across different configurations, even on a message-by-message basis. So one can encrypt one message under a secret key using a fully parallelizable configuration, and another message under an MRAE configuration, with the same secret key. Protocol developers can use this flexibility as needed for their applications. In most cases they will presumably use a single configuration; in this case flexible AEAD provides some defense-in-depth with respect to misconfigurations for the same secret key.

We provide new definitions to capture the syntax, semantics, and security of flexible AEAD. Flexible AEAD will start with some mandatory (always provided) security and operational attributes. Then through choices of the above-discussed configuration, it will further provide other attributes as options. Yet, this will be implemented in a unified way with what is essentially a single scheme.

As mandatory, we ask for classic UNAE and context commitment. The reason for making the latter mandatory is to avoid errors arising from developers not knowing that they need to turn it on. (Indeed, the errors and attacks we have seen are arising from applications assuming implicitly that commitment is present.) We also always ask for the ability to handle arbitrary-length keys and nonces. As optional, if requested in cfg, we support providing MRAE, AE2 and streaming capability. Other configurations can be added as needed.

The Flex scheme. Towards realizing flexible AEAD, we propose a clean-slate approach that reimagines AEAD scheme construction to allow it to provide modern security and performance for a broad range of application domains. Our Flex scheme conceptually has a straightforward structure: use a key derivation function (KDF) to derive sub-keys that can be safely used with different underlying schemes. Thus constructively, Flex, brings into the AEAD what used to be external, namely, a KDF function.

This flexibility would perhaps seem to require a complex AEAD scheme, depending on many underlying mechanisms. But we build a "wide waist" strategy: one scheme has a variety of different configurations all using the same underlying primitives. In particular, we build all of Flex's various configurations from a single underlying cryptographic permutation. For instance, we use a realization of tweaked Even-Mansour [13] built from a permutation. While Flex works for any sufficiently wide permutation, we have mostly focused on instantiations using Simpira [19].

We propose three primary Flex configurations. The OCH configuration provides an OCB-like mode of operation that is fully parallelizable, but not MR nor robust. It is the most performant. We also provide CIV (committing synthetic IV) an SIV [29] variant that provides MR security, and is as parallelizable as possible. Finally, we aim to provide robust AE based on prior constructions [20, 21] using our underlying tweakable block cipher.

Organization. The rest of the paper is organized as follows. In Section 2 we define our new cryptographic primitive flexible AEAD (F-AEAD). In Section 3 we provide an overview of our F-AEAD construction called Flex. This includes an overview of the three primary Flex configurations. Finally, in Section 4 we describe preliminary benchmarks on the OCH configuration compared with other AEAD constructions.

2 Flexible AEAD Definitions

We start with new definitions for AEAD. These build off a variety of prior work, as we explain further below, but weave in new aspects that we think will benefit secure, flexible deployability and usage.

Basic notation. We let $B = \{0,1\}^8$ be the set of 8-bit bytes. To be better aligned with practice, we will define primitives over byte strings rather than bit strings, but note that our constructions can easily be

changed to handle arbitrary bit strings.

Elements of F-AEAD. Recall that in classical AEAD [26, 27], encryption $C \leftarrow \mathsf{Enc}(K, N, AD, M)$ takes key, nonce, AD and message to return a ciphertext, while decryption $M \leftarrow \mathsf{Dec}(K, N, AD, C)$ takes key, nonce, AD and ciphertext to return the message. Security can be basic (unique-nonce) [26, 27], requiring that encryption under a key not repeat a nonce, or advanced (misuse resistant) [28], requiring only that it not repeat nonce, AD and message.

We provide a slightly different and more general formalization called flexible AEAD (F-AEAD). A novel element of our framework is the introduction of an extra initialization algorithm Init which chooses the appropriate AEAD algorithm based on the parameters specified in a configuration input cfg. For instance, amongst other things, cfg includes a flag cfg.mr that says whether or not misuse resistance is requested. This allows a user to dynamically make this choice in different settings without changing the key.

F-AEAD syntax. Formally, a flexible AEAD (F-AEAD) scheme Π specifies several algorithms and associated quantities, as follows:

- $K_* \xleftarrow{\$} \Pi.\mathsf{Kg}(\kappa)$: The randomized key generation algorithm takes an integer $\kappa \in \Pi.\mathsf{KL}$, which is the key length (in bytes), and outputs a byte string $K_* \in B^\kappa$ called a (secret) key. Here $\Pi.\mathsf{KL} \subseteq \mathbb{N}$ is the set of allowed key lengths.
- $K \leftarrow \Pi.\mathsf{Init}(\mathsf{cfg}, K_*, AD_s)$: The deterministic initialization algorithm initializes a new configuration based on parameters specified in the algorithm. It takes as input a configuration cfg, key K_*, and setup associated data $AD_s \in B^*$. The scheme specifies a set $\Pi.\mathsf{Cfgs}$ of possible (allowed) configurations, and it is required that cfg belongs to this set. The algorithm outputs a (secret) subkey $K \in \Pi.\mathsf{Cfgs} \times B^*$, where the first component is the configuration and the second component is the encryption key, or it outputs the distinguished error symbol $\perp$. If $K \neq \perp$, its encryption key component has length $\Pi.\mathsf{SL}(\mathsf{cfg}, |K_*|)$, where $\Pi.\mathsf{SL}$ is called the subkey-length function. The configuration associated with K is specified by $K.\mathsf{cfg}$.
- $C \leftarrow \Pi.\mathsf{Enc}(K, N, AD_m, M)$: The deterministic encryption algorithm takes as input subkey K, nonce $N \in B^*$, message associated data $AD_m \in B^*$, and message $M \in B^*$, and returns a ciphertext $C \in B^* \cup \{\perp\}$. If $C \neq \perp$, it has length $\Pi.\mathsf{CL}(K.\mathsf{cfg}, |N|, |M|)$, where $\Pi.\mathsf{CL}$ is called the ciphertext-length function.
- $M \leftarrow \Pi.\mathsf{Dec}(K, \tilde{N}, AD_m, C)$: The decryption algorithm takes as input subkey K, nonce $\tilde{N}$, associated data AD_m, and ciphertext C, and returns a message $M \in B^* \cup \{\perp\}$ that is either a byte string or the distinguished error symbol $\perp$. If nonce-hiding is desirable, then $\tilde{N}$ may be set to the empty string ε.

We proceed to some remarks, explanations, and choices.

Configurations. Configurations are customizable, but a configuration cfg always includes flags cfg.mr, cfg.nh $\in \{\mathsf{true}, \mathsf{false}\}$ indicating whether or not misuse resistant security or nonce-hiding is requested and cfg.perm $\in B^*$ indicating the type of permutation to use. These flags may be set independently of each other.

Flag	Configuration	Type
mr	misuse resistance	boolean
nh	nonce-hiding	boolean
perm	permutation	string

Many key lengths. Typically, an encryption scheme mandates a single length for its key. Our definition instead allows keys of different lengths. (In our construction, any length up to some maximum.) Security

will depend on the shortest key length used. The usual setting of a single key length is captured by having $\Pi.\mathsf{KL}$ be a singleton set. (It contains just one length.)

Validity of inputs. Whether or not $\Pi.\mathsf{Enc}(K, N, AD_m, M)$ returns $\perp$ is required to depend only on $K.\mathsf{cfg}$, $|K|$, $|AD_m|$, $|M|$ and can be easily computed given these quantities. We require that $\Pi.\mathsf{Enc}(K, N, AD_m, M) = \perp$ if $K.\mathsf{cfg} \notin \Pi.\mathsf{Cfgs}$. Furthermore, if $K.\mathsf{cfg.mr} = \mathsf{true}$, meaning that misuse resistance is requested, then we require that $\Pi.\mathsf{Dec}(K, \tilde{N}, AD_m, C) = \perp$ if $\tilde{N} \neq \varepsilon$. This is because specifying a nonce during decryption when nonce misuse is requested should indicate an error.

Correctness. For correctness, we require that decryption of legitimate ciphertexts always succeeds. In detail, for any K, N, AD_m, M, if $C \leftarrow \Pi.\mathsf{Enc}(K, N, AD_m, M) \neq \perp$ and $K.\mathsf{cfg.mr} = \mathsf{false}$, then $\Pi.\mathsf{Dec}(K, N, AD_m, C) = M$. Otherwise, if $K.\mathsf{cfg.mr} = \mathsf{true}$, then it should be that $\Pi.\mathsf{Dec}(K, \varepsilon, AD_m, C) = M$.

Tidiness. Extending [28], we say that Π is *tidy* if: For any K, N, AD_m, C, if $M \leftarrow \Pi.\mathsf{Dec}(K, N, AD_m, C) \neq \perp$ then $\Pi.\mathsf{Enc}(K, N, AD_m, M) = C$. This is not always required, but our schemes provide it.

Ciphertext length. Typically the ciphertext is stretched by some constant τ as compared with the plaintext message. But we may have schemes for which the configuration dictates different security levels, and in turn config-dependent stretch.

3 The Flex AEAD

We propose an F-AEAD called Flex. Flex offers solutions to common deployment settings and security across different configurations, while retaining performance that is competitive with or beats the best known (non-flexible) AEAD schemes. To do so, Flex's design is modular, and can be viewed as baking into the AEAD's design what is traditionally left to users of AEAD, namely choice of different AEAD types (misuse-resistance versus not) and key derivation. We will show how this affords Flex to easily adapt to application requirements, while maintaining programmatic configuration for developers with associated simple-to-use APIs.

Overview. We build Flex modularly. At a high level there are four main components all built from the same underlying permutation π:

- A key derivation function (KDF) that achieves CR-PRF security. It is used to derive a subkey from the secret subkey, associated data, and configuration. By default this is a simple sponge-style hash construction, but we also give modes that allow parallelization and precomputation of static elements of the input, such as the associated data .
- A non-misuse resistant encryption mode called OCH that is an OCB-like mode that provides context committing security and nonce-hiding while being fully parallelizable.
- A misuse resistant encryption mode called CIV that is an SIV-like mode that provides context committing security and nonce-hiding while being maximally parallelizable.
- A robust encryption mode.

All components are new to this work, but of course use some design elements seen in prior work as we will highlight where relevant. We can then mix-and-match the different KDF modes with the different encryption modes to provide various designs, which enables a wide variety of AEAD schemes. At this level our individual constructions follow the Hash-then-Encrypt mechanism from Bellare and Hoang [3] that transforms a key-committing AEAD into a context-committing AEAD. But we carefully arrange our constructions, down to the level of using a fast-to-rekey specially constructed Even-Mansour [16] cipher, to avoid inefficiencies.

Preliminaries. Let $\pi : \{0,1\}^n \to \{0,1\}^n$ be a permutation that we will model as an ideal permutation. Fix a number $\tau < n$. We will target security of $\tau/2$ bits or greater. As a running example we will use $\tau = 256$ and $n = 512$, giving 128-bit security.

A permutation-based function. We use a few basic primitives built from π. The first is a variable-output-length function $F : \{0,1\}^* \times \mathbb{Z} \times \{0,1\}^* \to \{0,1\}^*$. We define that $F(K, \ell, X)$ outputs a bit string of length ℓ using key K. We will require that F be a collision-resistant (CR) pseudorandom function (PRF). One possibility for F is to use a sponge-type construction [6] with the desired output length prepended to the message. In this work, we provide our own construction that minimizes the underlying calls to π. For simplicity, we do not describe the details of this construction here.

A permutation-based TBC. Our second underlying primitive is a tweakable blockcipher (TBC), also built from π. Let $\mathcal{N} = \{0,1\}^{\leq n-8}$ be the set of all bit strings whose length is at most $n-8$. Then we use a tweakable blockcipher $\tilde{E} : \mathsf{Keys} \times \mathsf{Tweaks} \times \{0,1\}^n \to \{0,1\}^n$ whose key space is $\mathsf{Keys} = \{0,1\}^\tau \times \{0,1\}^\tau \times \{0,1\}^r$, tweak space is $\mathsf{Tweaks} = \{0, 1, (1,1), (1,2)\} \cup \left(\mathbb{Z} \times [1,2,3] \times \mathcal{N}\right)$ and whose block length is n bits. The set of possible nonces includes any bit string N with $0 \leq |N| \leq n-1$.

At a high level, $\tilde{E}$ is constructed using a special version of tweaked Even-Mansour [13] built from a permutation. For simplicity, we will not cover the details of the construction here.

3.1 Flex Overview

We provide a top-down description of Flex. It supports up to 256 different configurations, allowing cfg to be represented by one byte. For clarity in our presentation, in addition to the configuration flags we specify in Section 2, we let cfg.TL indicate the number of bits of tag we desire, cfg.NL indicate the length of the nonce, and cfg.SL indicate value τ such that the subkey is length 3τ. We define Flex as follows.

- Flex.Kg(κ) outputs a uniform bit string K_* of length κ.
- Flex.Init(cfg, K_*, AD_s) verifies that cfg belongs to the set of possible (allowed) configurations Π.Cfgs and that $|K_*|$ is in the set of allowed key lengths Π.KL. If not, it outputs an error. Otherwise, it applies the key derivation function KD(K_*, cfg, AD_s) to generate three keys (K_1, K_2, K_3) each of length τ = cfg.SL bits and forms subkey $K \leftarrow (\mathsf{cfg}, K_1 \parallel K_2 \parallel K_3)$. We thus define Π.SL(cfg, $|K_*|$) as returning 3·cfg.SL. Optionally it may perform some precomputation of values that will help speed-up encryption (e.g., the table L used for fast computation of our TBC). We refer to such values that get used by multiple encryption calls as state, and notate it by st.
- Flex.Enc(K, N, AD_m, M) works in different ways depending on K.cfg. For instance, if K.cfg.mr = true, then misuse resistant mode CIV is chosen for encryption; otherwise, non-misuse resistant mode OCH is chosen. The output is a ciphertext consisting of a triple (C_{hdr}, C, T) consisting of a (possibly empty) ciphertext header C_{hdr}, a ciphertext body C, and a tag T and an updated state st. Some modes may have C_{hdr} empty.
- Flex.Dec($K, \tilde{N}, AD_m, C$), like encryption, proceeds depending on the parameters specified in K.cfg. If K.cfg.nh = true and $\tilde{N} \neq \varepsilon$, then decryption returns an error. Otherwise, it proceeds according to the mode specified by the configuration to return the message M.

3.2 The Flex KDF

We start by describing the KDF used by Flex, which simply combines an encoding of the inputs (cfg, K_*, AD_s) and generates subkeys K_1, K_2, K_3 and (optionally) some pre-computed state.

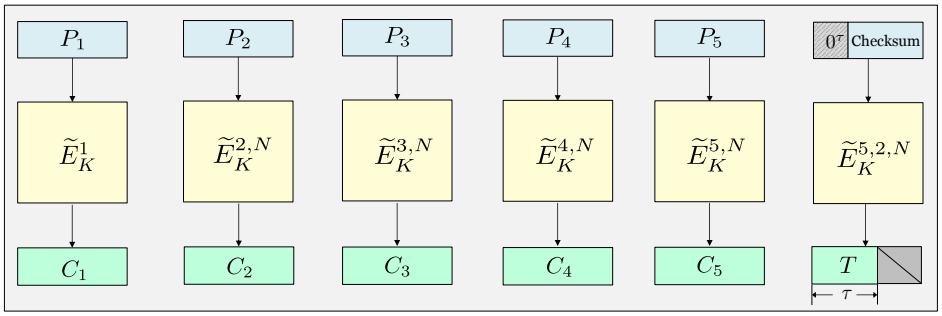

Figure 1. Illustration of OCH. Given a nonce N and a message M, we parse $N \parallel M$ into $P_1 \cdots P_m$, where each $|P_i| = n$. When $N \parallel M$ is block-aligned, $|N| + |M| \geq n$, and $n \geq 2\tau$, Checksum is computed as the first $n - \tau$ bits of $P_1 \oplus P_2 \oplus P_3 \oplus P_4 \oplus P_5$.

We compute an encoding K' of the key K_* to serve as the PRF key. In particular, if Flex.KL is a singleton set (only one allowed key length), then we let K' be K_*. Otherwise, we let K' be a 64-bit encoding of the key length followed by the key: $\langle |K_*| \rangle_{64} \| K_*$. The bit string encoding S is computed as follows. The first byte of S is cfg. Similar to how K_* is encoded, if cfg allows only a single setup AD length, then we append AD_s and otherwise append a 64-bit encoding of the length and then the AD. All of this ensures an unambiguous encoding of cfg and AD_s. We then apply $F(K', 3\tau, S)$ to yield an encryption subkey triple (K_1, K_2, K_3). In our scheme, K_1 will serve as a commitment to the input.

3.3 The Flex Non-Misuse Resistant Mode: OCH

We now describe the OCH AEAD scheme used by Flex when the configuration specifies that misuse resistance is not needed. It is by default nonce-hiding and context committing. Recall that $\tau =$ cfg.SL, which is used to derive the length of the subkey, and n is the block length of our TBC $\tilde{E}$. For this construction, we assume $\tau < n$. Detailed pseudocode is provided in Figure 2, while an example illustration is provided in Figure 1.

Encryption takes as input a subkey $K = (\mathsf{cfg}, K_1 \parallel K_2 \parallel K_3)$ where each K_i for $i \in \{1, 2, 3\}$ is of size τ. In addition, it takes as input the per-message associated data AD_m, and a plaintext M. Encryption first verifies that the length of the nonce matches that specified by cfg.NL and returns the error symbol $\perp$ otherwise. If $|AD_m| > 0$, the algorithm then computes a new subkey K_1 via $K_1 \leftarrow F(K_1, \tau, AD_m)$. Encryption proceeds depending on the length of the nonce and message, whether they are block-aligned, and how n compares to τ. For simplicity, we will describe the case when $N \parallel M$ is block-aligned and $n \geq 2\tau$, but details of each case are presented in Figure 2.

Encryption. When $\ell = |N| + |M| \geq n$ and ℓ is a multiple of n, we use an OCB-style mode to encrypt $P = N \parallel M$. We assume here that $|N|$ is fixed by cfg.NL, or otherwise simple concatenation would be an ambiguous encoding. To encrypt, we first split P into m blocks $P_1, \ldots, P_m$, each of length n bits. The algorithm then computes $C_1 \leftarrow \tilde{E}_K^1(P_1)$ and for $2 \leq i \leq m$ it computes $C_i \leftarrow \tilde{E}_K^{i,N}(P_i)$. To generate the tag, we compute Checksum $\leftarrow P_1 \oplus \cdots \oplus P_m$ and then we let $T \leftarrow \tilde{E}_K^{m,2,N}(0^\tau \parallel \text{Checksum}[1..n-\tau])[1..\tau]$. This corresponds to having enough room in n to allow us to commit to the key K_1 — our design of $\tilde{E}$ guarantees that prepending 0^τ for any tweak is committing to K_1 if $n \geq 2\tau$ — while simultaneously using $\tilde{E}$ as a PRF to complete generation of an authentication tag. Note that here we don't actually require all of Checksum to be processed, so we truncate it to a sufficiently large amount.

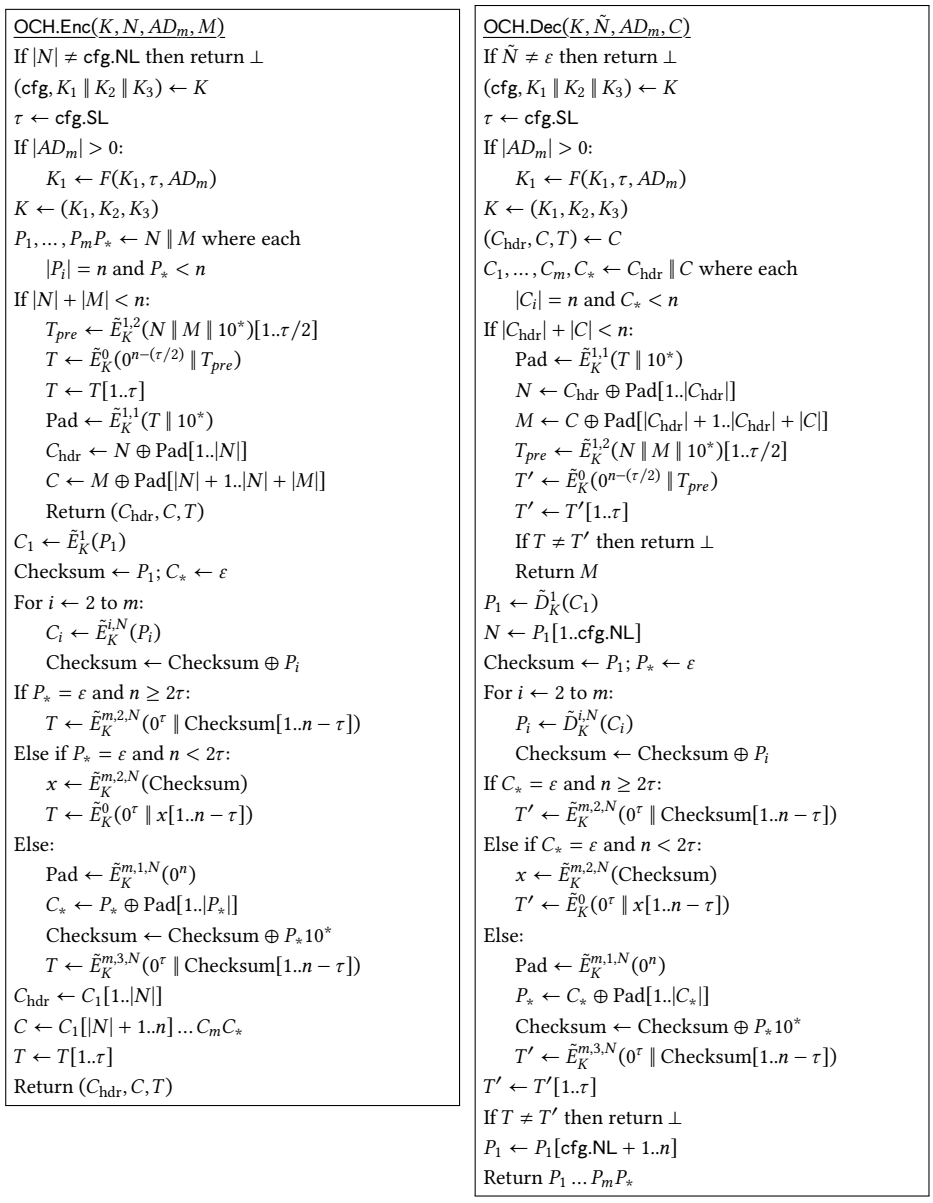

Figure 2. The OCH encryption and decryption subroutines.

Decryption. The decryption algorithm of OCH proceeds as expected. Since OCH is nonce-hiding, if the nonce $\tilde{N}$ is specified rather than left as the ε, the algorithm returns error symbol $\perp$. The algorithm decrypts each ciphertext block, then recomputes the authentication tag, and verifies that it matches the tag that is part of the ciphertext. If they do not match, then decryption fails and returns error symbol $\perp$. Otherwise, decryption returns the plaintext message.

Removing nonce hiding. OCH provides nonce hiding as a default property and so also encrypts the nonce. If it is desired to use a more basic setting that assumes unique-nonce security and no nonce hiding, then OCH can be updated so that only the message is encrypted and the nonce is then given as an input during decryption.

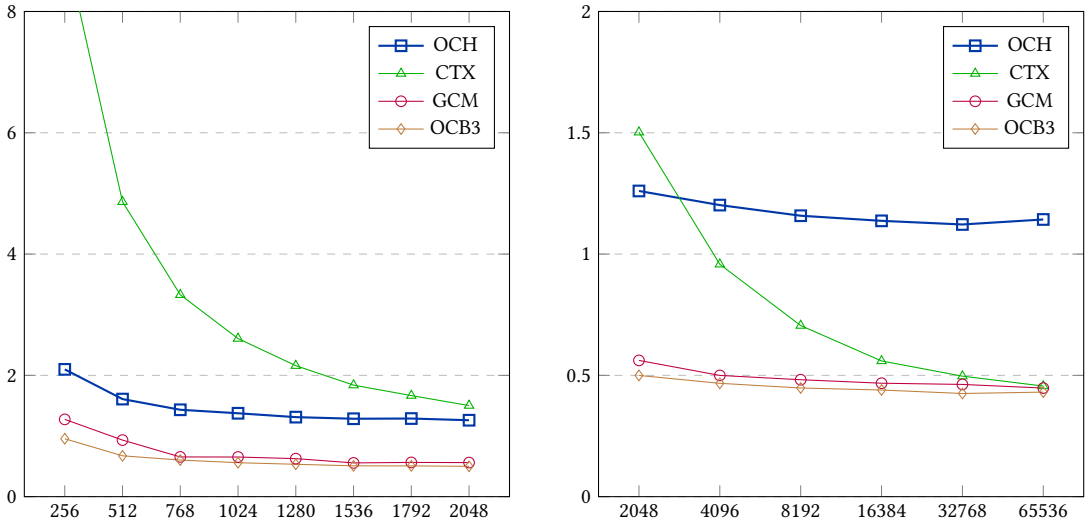

Figure 3. Encryption prformance on an Intel i7-9750H, an x86_64 processor with AES-NI. The x-axis shows message length in bytes and the y-axis shows the throughput in CPU cycles-per-byte (lower is better). **(Left)** Performance on medium-size messages, in linear scale. **(Right)** Performance on large-size messages, in log scale.

3.4 Other Flex Modes

We briefly describe the other modes provided by Flex.

The MR configuration CIV. CIV is the misuse resistant mode associated with Flex. It is used when the configuration specifies that MR is requested via `cfg.mr = true`. We briefly give an overview of how it works here. More details on its construction will be forthcoming.

At a high level, the construction first computes a PRF based on PMAC [7] over the message using the subkey. It computes a CR hash over the output of the PRF and the subkey to generate the authentication tag, which serves as the synthetic IV. Finally, it uses a CTR-mode with the subkey to encrypt the message and form the ciphertext.

The robust configuration. As we mention in the introduction, another desirable functionality for Flex is to provide a robust AEAD mode. Robust AEAD schemes allow specifying the maximal ciphertext stretch, where ciphertext integrity should be achieved up to the maximal possible.

We aim to support robust AEAD by using a prior TBC, such as encode-then-encipher with an EME* [20] mode or AEZ [21]. Like our MR configuration, more details on this construction will be forthcoming.

4 Performance

Figure 3 compares the performance of OCH to GCM [15], OCB3 [22,23], and the CTX construction [12] instantiated with GCM and HMAC-SHA256. GCM and OCB3 represent the performance of widely deployed AEAD schemes, while CTX with GCM and HMAC-SHA256 represents the performance of current 128-bit context committing AEAD schemes.

The implementation of GCM is from Ring [30] which uses OpenSSL's assembly GCM implementation[1]. The implementation of OCB3 is the optimized C implementation from the OCB3 webpage[2]. The CTX

[1] `https://github.com/briansmith/ring/blob/95948b39/crypto/fipsmodule/modes/asm/aesni-gcm-x86_64.pl`
[2] `https://www.cs.ucdavis.edu/~rogaway/ocb/news/`

implementation was created by combining Ring's GCM with Ring's HMAC-SHA256.

During benchmarking, we fixed the size of the associated data to be 16 bytes, and used default size nonces (12 bytes for GCM, OCB3, and CTX; and 32 bytes for OCH). A message buffer of the specified length was initialized with random bytes and this buffer was iteratively encrypted. This encryption was repeated at least 100,000 times for each configuration of message size and scheme, and preceded by a warm-up of at least 100,000 executions (to warm caches). Cycle counts were measured periodically using the `RDTSC` instruction. After ensuring that the variance between these periodic measurements was low, they were averaged to get the final measurements. Finally, the full experiment was repeated two times to check that the variance across runs was also low.

On medium-size messages (256 to 2048 bytes), OCH outperforms CTX, because of the high cost of invoking HMAC-SHA256. But, for large-size messages (4096 bytes and larger), since HMAC-SHA256 is only invoked once and its inputs (GCM tag, associated data, nonce, and key) don't vary with message size, this cost is amortized. Thus CTX outperforms OCH and eventually matches the performance of plain GCM.

Benchmarks on other platforms and against other AEAD schemes are forthcoming.

References

[1] A. Albertini, T. Duong, S. Gueron, S. Kölbl, A. Luykx, and S. Schmieg. How to abuse and fix authenticated encryption without key commitment. USENIX Security 2022, 2022.

[2] AWS. Supported algorithm suites in the AWS Encryption SDK, 2021. `https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/supported-algorithms.html`. Accessed 2021-09-23.

[3] M. Bellare and V. T. Hoang. Efficient schemes for committing authenticated encryption. In O. Dunkelman and S. Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 845–875. Springer, Heidelberg, May / June 2022.

[4] M. Bellare, R. Ng, and B. Tackmann. Nonces are noticed: AEAD revisited. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 235–265. Springer, Heidelberg, Aug. 2019.

[5] M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. In T. Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2000.

[6] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, and G. V. Assche. Cryptographic sponge functions. *Keccak team*, 2011.

[7] J. Black and P. Rogaway. A block-cipher mode of operation for parallelizable message authentication. In L. R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 384–397. Springer, 2002.

[8] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In N. Silvanovich and P. Traynor, editors, *10th USENIX*

Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016. USENIX Association, 2016.

[9] A. Bozhko. Properties of AEAD algorithms. Internet-Draft draft-irtf-cfrg-aead-properties-01, Internet Engineering Task Force, Mar. 2023. Work in Progress.

[10] J. Chan and P. Rogaway. Anonymous AE. In S. D. Galbraith and S. Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 183–208. Springer, Heidelberg, Dec. 2019.

[11] J. Chan and P. Rogaway. On committing authenticated-encryption. In V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, editors, *ESORICS 2022 , Part II*, volume 13555 of *LNCS*, pages 275–294. Springer, Heidelberg, Sept. 2022.

[12] J. Chan and P. Rogaway. On committing authenticated-encryption. In *European Symposium on Research in Computer Security*, pages 275–294. Springer, 2022.

[13] B. Cogliati, R. Lampe, and Y. Seurin. Tweaking Even-Mansour ciphers. In R. Gennaro and M. Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 189–208. Springer, 2015.

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

[15] M. Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, 2017.

[16] S. Even and Y. Mansour. A construction of a cipher from a single pseudorandom permutation. In H. Imai, R. L. Rivest, and T. Matsumoto, editors, *ASIACRYPT'91*, volume 739 of *LNCS*, pages 210–224. Springer, Heidelberg, Nov. 1993.

[17] P. Farshim, C. Orlandi, and R. Rosie. Security of symmetric primitives under incorrect usage of keys. *IACR Trans. Symmetric Cryptol.*, 2017(1):449–473, 2017.

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

[19] S. Gueron and N. Mouha. Simpira v2: A family of efficient permutations using the AES round function. In J. H. Cheon and T. Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 95–125, 2016.

[20] S. Halevi. EME*: Extending EME to handle arbitrary-length messages with associated data. In A. Canteaut and K. Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004, 5th International Conference on Cryptology in India, Chennai, India, December 20-22, 2004, Proceedings*, volume 3348 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2004.

[21] V. T. Hoang, T. Krovetz, and P. Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2015.

[22] T. Krovetz and P. Rogaway. The OCB Authenticated-Encryption Algorithm. Request for Comments - Informational, 2014.

[23] T. Krovetz and P. Rogaway. The design and evolution of OCB. *J. Cryptol.*, 34(4):36, 2021.

[24] J. Len, P. Grubbs, and T. Ristenpart. Partitioning oracle attacks. In M. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 195–212. USENIX Association, 2021.

[25] S. Menda, J. Len, P. Grubbs, and T. Ristenpart. Context discovery and commitment attacks - how to break CCM, EAX, SIV, and more. In C. Hazay and M. Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part IV*, volume 14007 of *Lecture Notes in Computer Science*, pages 379–407. Springer, 2023.

[26] P. Rogaway. Authenticated-encryption with associated-data. In V. Atluri, editor, *ACM CCS 2002*, pages 98–107. ACM Press, Nov. 2002.

[27] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In M. K. Reiter and P. Samarati, editors, *ACM CCS 2001*, pages 196–205. ACM Press, Nov. 2001.

[28] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 373–390. Springer, Heidelberg, May / June 2006.

[29] P. Rogaway and T. Shrimpton. The SIV Mode of Operation for Deterministic Authenticated-Encryption (Key Wrap) and Misuse-Resistant Nonce-Based Authenticated-Encryption, 2007. Draft 0.32.

[30] B. Smith et al. ring, 2023. `https://github.com/briansmith/ring/tree/95948b3977013aed16`. Accessed 2023-07-01.