Key Committing AEADs

Shay Gueron

University of Haifa and AWS

Abstract. This note describes some methods for adding a key commitment property to a generic (nonce-based) AEAD scheme. We analyze the the privacy bounds and key commitment guarantee of the resulting constructions, by expressing them in terms of the properties of the underlying AEAD scheme and the added key commitment primitive. We also offer concrete constructions for a key committing version of AES-GCM.

Keywords. AEAD, Robust encryption, key commitment.

1 Introduction

Recent publications (e.g., [3], [4]) and new shaping designs (OPAQUE [6]) raised the question and interest in the relation between the result ciphertext decryption and the key that was presentably used to encrypt "the" plaintext. Reality turns out to be be sometimes counter intuitive. We focus here on symmetric key authenticated encryption with associated data (AEAD).

A nonce-based AEAD scheme receives, at encryption, a nonce N, a public header A and a message M, and uses a (secret) key (K) to produce the ciphertext C and the authentication tag T. Decryption for the input (N, A, C, T), in the context of the key K, produces either a decrypted message M if it was successfully authenticated, or an authentication failure indication $\perp$. The design goal of an AEAD is to provide privacy and authenticity for two communicating parties who share a secret key K, loosely described as:

– Privacy: an adversary that sees C, T (and N, A) samples, computed over its chosen N, A, M inputs, has negligible advantage in distinguishing them from random strings with the matching lengths.
– Authenticity: if the tuple (N, A, C, T) is input to decryption, and C, T is not the output of a previous encryption of a tuple (N, A, M), then the probability that decryption does not return $\perp$ is negligible.

However, binding a tuple (N, A, C, T) to a specific key is *not* a design goal of an AEAD: such a tuple can have non-negligible (even large) probability to pass authentication under two distinct keys K_1, K_2. In fact, with some AEADs it is easy for an adversary that knows (or chooses) K_1, K_2 to compute such tuples. One example is AES-GCM where the authentication tag is the one-time-pad encryption of a non-cryptoraphic hash function.

Binding a tuple (N, A, C, T) to a key. Our goal is to tweak a given AEAD by using an additional value ($\mathsf{K_C}$) that we call here a *Key Committing string*, which serves as a visible (non-secret) key identifier. The resulting key committing scheme would encrypt a tuple N, A, M to C, T, $\mathsf{K_C}$, and decryption would take N, A, C, T, $\mathsf{K_C}$ as input. The design needs to provide privacy and authenticity guarantees similar to those of the original underlying AEAD scheme, and also to prevent an adversary from finding distinct keys K_1, K_2 and a tuple $(N, A, C, T, \mathsf{K_C})$ that passes authentication under both keys.

2 Preliminaries and notation

Let $\{0,1\}^*$, $\{0,1\}^s$, $\{0,1\}^{\leq s}$ ($s \geq 0$) be, respectively, the set of all binary strings (including the empty string of length 0), the set of all binary strings of length s (bits), and the set of binary strings of length at most s.

The length of a string S is denoted by $|S|$. By convention, strings of bits are written in a way that the bit in the leftmost position is called the most significant bit and the bit in the rightmost position is called the least significant bit. For example, the most (least) significant bit of 10100100 is 1 (0). For S with $|S| \geq \alpha$ we denote the α least significant bits of S by $S_{[\alpha]}$. For example if $S = 10100100$ then $|S| = 8$ and $S_{[3]} = 100$. The string that consists of $s \geq 0$ repeated zero bits is denoted by 0^s (the degenerate case $s = 0$, is the empty string). For example $0^8 = 00000000$. For brevity, we also use the hexadecimal notation for strings of bits. For example, 10100100 is $0xA4$ in hexadecimal notation.

The concatenation of the strings U and V is a string of $|U|+|V|$ bits, denoted by $U \parallel V$. By convention $U \parallel V$ has the bits of U in the $|U|$ leftmost positions and the bits of V in the $|V|$ rightmost positions. For example, if $U = 1111$ and $V = 0000$, then $U \parallel V = 11110000$.

The symbol $\perp$ indicates authentication failure. For a finite set W we denote the uniform sampling from W and assigning the value to w by $w \leftarrow_{\$} W$.

2.1 AEAD schemes

A nonce-based Authenticated Encryption with Associated Data (AEAD) is a triple of algorithms, $\Pi = (Gen, Enc, Dec)$ associated with the key $\mathcal{K}$, nonce $\mathcal{N}$, message $\mathcal{M}$, header $\mathcal{A}$, ciphertext $\mathcal{C}$, and tag $\mathcal{T}$ spaces, all being finite subsets of $\{0,1\}^*$. With no loss of generality, we assume hereafter that $\mathcal{N} = \{0,1\}^\nu$ for some nonce length $\nu > 0$, that $\mathcal{K} = \{0,1\}^\kappa$ for some key length $\kappa > 0$, and that *Gen* is $K \leftarrow_{\$} \{0,1\}^\kappa$.

Encryption is a deterministic algorithm that takes input $K, N, A, M \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$ and outputs ciphertext and tag $C, T \in \mathcal{C} \times \mathcal{T}$. We assume here that $|C| = |M|$[1]. We denote the encryption operation by $Enc(K, N, A, M)$ and the result by (C, T). Decryption is a deterministic algorithm that takes input $K, N, A, C, T \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T}$ and outputs either a message $M \in \mathcal{M}$ (if

[1] There exist AEAD schemes where $|C| > |M|$, but we ignore such schemes here.

tag authentication passes) or a failure symbol $\perp$ (if tag authentication fails). We denote the decryption operation by $Dec(K, N, A, C, T)$, where the output is either M or $\perp$. An AEAD scheme satisfies the following: if $K, N, A, M \in \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M}$ and $Enc(K, N, A, M) = (C, T)$, then $Dec(K, N, A, C, T) = M$ (in particular, $\neq \perp$).

A random nonce (randomized) AEAD scheme is defined analogously to a nonce-based AEAD scheme above, with the following difference: Enc takes only K, A, M as input, generates $N \leftarrow_\$ \mathcal{N}$ and outputs $C, T\ N$.

Payloads. The AEAD definition captures also the case where the header / message is a "payload" divided (logically) to multiple chunks, and the nonce (input or generated) may also be divided to multiple chunks that are processed independently. The ciphertext and tag are divided to multiple chunks accordingly. In such cases, it is assumed that Π is used in a context that defines, unambiguously, how inputs (N, A, M) (to Enc) and inputs (N, A, C, T) (to Dec) are parsed. Constraints on the uniqueness of the nonce(s) can also be imposed.

Example 1 (Payload encryption with AES-GCM). Let Π be a "payload encryption" scheme using the standard `AES-GCM`. The payload, (N, A, M), input to encryption, is parsed as: v header/plaintext chunks $A = a_1 \parallel a_2 \parallel \ldots \parallel a_v$, $M = m_1 \parallel m_2 \parallel \ldots \parallel m_v$, and v (sub-)nonces $N = n_1 \parallel n_2 \parallel \ldots \parallel n_v$ where $|n_j| = 96$, $j = 1, \ldots, v$, with the constraint that $n_1, n_2, \ldots, n_v$ are distinct. It is assumed that the context and input/output format for Π defines unambiguous slicing to chunks and nonces. The encryption process uses a key K, computes $(c_j, t_j) = \texttt{AES-GCM}(K, n_j, a_j, m_j)$, $j = 1, \ldots, v$, and outputs $C = c_1 \parallel c_2 \parallel \ldots \parallel c_v$, and $T = t_1 \parallel t_2 \parallel \ldots \parallel t_v$, in an agreed format. We point out that the message format of AWS Encryption SDK [1] is (in some configurations) also an example for encrypting a payload.

2.2 Advantage against a scheme

Let $\mathcal{K} = \{0,1\}^\kappa$ (key space) and $\mathcal{IN}$, $\mathcal{OUT}$ (input/output space) be finite subsets of $\{0,1\}^*$. Let $\mathsf{Scheme} : \mathcal{K} \times \mathcal{IN} \rightarrow \mathcal{OUT}$ be a function/procedure and denote its operation over input $K \in \mathcal{K}$ and $U \in \mathcal{IN}$ by $\mathsf{Scheme}(K, U) = Res$. An oracle $\mathcal{O}_{\mathsf{Scheme}}$ for Scheme is an entity that chooses, uniformly at random, a challenge bit b and a key $K \in \mathcal{K}$, and answers queries from $\mathcal{IN}$ ("legitimate" queries hereafter). For a query $U \in \mathcal{IN}$, $\mathcal{O}_{\mathsf{Scheme}}$ computes $\mathsf{Scheme}(K, U) = Res$, generates $R_\$ \leftarrow_\$ \{0,1\}^{|Res|}$, and returns Res if $b = 1$ and $R_\$$ if $b = 0$.

An adversary $\mathbf{C}$ against Scheme is a Polynomial Time Probabilistic (PPT) adaptive algorithm that has access to $\mathcal{O}_{\mathsf{Scheme}}$, and a budget for (say q) queries to its oracle $\mathcal{O}_{\mathsf{Scheme}}$ and some run time $t_{\mathbf{C}}$. $\mathbf{C}$ submits at most q legitimate and non-redundant queries (e.g., no repeated queries), receives $\mathcal{O}_{\mathsf{Scheme}}$'s responses, and outputs a bit b'. The distinguishing (Real-or-Random) advantage of $\mathbf{C}$ against Scheme is

$$Adv_{\mathsf{Scheme}}(\mathbf{C}) = \left| Prob(b' = 1 |\ b = 1) - Prob(b' = 1 |\ b = 0) \right| \quad (1)$$

A randomized scheme draws a uniformly random value V (of a specified length), involves V in the computation of $\mathsf{Scheme}(K, U) = Res$, and returns Res and V.

Multi-key setting. The notion of oracle and advantage can be generalized to the following multi-key setting where $\mathcal{O}_{\mathsf{Scheme}}$ samples q keys $K_j \leftarrow_{\$} \mathcal{K}$, $j = 1, \ldots, q$, and uses K_j for a query of the form (j, U) that $\mathbf{C}$ submits ($j \leq q$) as follows: computes $\mathsf{Scheme}(K_j, U) = Res$, generates $R_{\$} \leftarrow_{\$} \{0,1\}^{|Res|}$, and returns Res if $b = 1$ and $R_{\$}$ if $b = 0$. As in the single-key scenario, $\mathbf{C}$ outputs a bit b', and the advantage is as in (1). The case $q = 1$ degenerates to the standard single-key scenario. A special case is the "fresh-multi-key" setting where a new key from the list $K_1, \ldots, K_q$ is used for every call/query, i.e., queries do not specify explicitly the key to be used (the keys can be generated a priori or per call — "online"). Note that responses to (at most) q queries that $\mathbf{C}$ may submit involve at most q distinct keys because there could be colliding values among $K_1, \ldots, K_q$. The probability that the q keys are distinct is at least $1 - q^2/2^{1+\kappa}$.

Remark 1 (Key guessing). Independently of oracle queries, $\mathbf{C}$ may compute ("offline") Scheme with T_{Scheme} chosen keys (and inputs). If a guessed key equals to a key that $\mathcal{O}_{\mathsf{Scheme}}$ actually used in the queries, then $\mathbf{C}$ wins. With at most q different keys for $\mathcal{O}_{\mathsf{Scheme}}$, the key guessing probability is at most $(T_{\mathsf{Scheme}} \cdot q)/2^{\kappa}$.

3 Constructing a key committing AEAD

3.1 Construction

Let $\Pi = (Gen, Enc, Dec)$ be a nonce-based AEAD scheme defined with the spaces $\mathcal{K} = \{0,1\}^{\kappa}$, $\mathcal{A}$, $\mathcal{N}$, $\mathcal{M}$, $\mathcal{C}$, $\mathcal{T}$ for some $\kappa > 0$. We construct the schemes $\mathsf{DeriveKey}\Pi$ and $\mathsf{CommitKey}\Pi$ over Π.

The constructions are defined with the positive parameters κ_0, ν_1, c where, with no loss of generality, $\kappa_0 \geq \max\ (\kappa, c)$, and the two *distinct* public strings (labels) L_1, L_2 with equal lengths $|L_1| = |L_2| = \ell_L$. Denote $\ell = \ell_L + \nu_1$. Let

$$F(K, L) : \{0,1\}^{\kappa_0} \times \{0,1\}^{\leq(\ell_L + \nu_1)} \rightarrow \{0,1\}^{\max\ (\kappa, c)}$$

be a pseudorandom function keyed $K \in \{0,1\}^{\kappa_0}$. Both schemes use a key $K \in \{0,1\}^{\kappa_0}$, called "main key", and their key generation procedure is $K \leftarrow_{\$} \{0,1\}^{\kappa_0}$. The input to encryption is a legitimate payload $(N, A, M) \in \mathcal{N} \times \mathcal{A} \times \mathcal{M}$ (i.e., legitimate input to Π encryption), and possibly a nonce $N_1 \in \{0,1\}^{\nu_1}$.

$\mathsf{DeriveKey}\Pi$ is an AEAD that derives an encryption key $\mathsf{K_E}$ from K and possibly a nonce $N_1 \in \{0,1\}^{\nu_1}$, uses $\mathsf{K_E}$ to encrypt the payload with Enc and outputs the resulting C, T. $\mathsf{CommitKey}\Pi$ extends $\mathsf{DeriveKey}\Pi$ by deriving an additional value $\mathsf{K_C} \in \{0,1\}^{c}$ from K and possibly a nonce $N_1 \in \{0,1\}^{\nu_1}$. It outputs C, T, $\mathsf{K_C}$, where $\mathsf{K_C}$, is hereafter called a *Key Committing* string, serves as a non-confidential *key identifier*.

We use different ways to derive $\mathsf{K_E}$ and $\mathsf{K_C}$ from the main key (K), with or without the nonce N_1, as follows:

Fixed: $\mathsf{K_E} = F_{[\kappa]}(K, L_1)$; or nonce-based $\mathsf{K_E} = F_{[\kappa]}(K, L_1 \parallel N_1)$;
Fixed: $\mathsf{K_C} = F_{[c]}(K, L_2)$; or nonce-based $\mathsf{K_C} = F_{[c]}(K, L_2 \parallel N_1)$;

(we use $F_{[c]}(K, L_2)$ to denote $\big(F(K, L_2)\big)_{[c]}$) and name the four corresponding $\mathsf{CommitKey}\Pi$ (two for $\mathsf{DeriveKey}\Pi$) flavors by:

Type I) fixed $\mathsf{K_E}$ and fixed $\mathsf{K_C}$;
Type II) nonce-based $\mathsf{K_E}$ and fixed $\mathsf{K_C}$;
Type III) fixed $\mathsf{K_E}$ and nonce-based $\mathsf{K_C}$;
Type IV) nonce-based $\mathsf{K_E}$ and nonce-based $\mathsf{K_C}$.

$\mathsf{DeriveKey}\Pi$ decryption is obvious. $\mathsf{CommitKey}\Pi$ decryption uses the input $\mathsf{K_C}$ to verify the main key K. The flows are illustrated in Figure 1 (top). The different flavors of $\mathsf{CommitKey}\Pi$ are associated with different incremental computational (computing F) and bandwidth overheads on top of $\mathsf{CommitKey}\Pi$ and on top of Π. Figure 1 (bottom) describes these different overheads.

Remark 2 (Randomized versions). The randomized version of $\mathsf{CommitKey}\Pi$ ($\mathsf{DeriveKey}\Pi$) samples $N_1 \leftarrow_{\$} \{0,1\}^{\nu_1}$ during Enc and includes N_1 as part of the encryption output. Nonce collision probability across q messages is at most $q^2/2^{1+\nu_1}$.

Remark 3 (The $\mathsf{CommitKey}\Pi$ constructions). $\mathsf{CommitKey}\Pi$ can be viewed as either an enhancement of Π (adding a derivation of $\mathsf{K_E}$ and $\mathsf{K_C}$ to Π) or an enhancement of $\mathsf{CommitKey}\Pi$ (adding (only) the derivation of $\mathsf{K_C}$ to $\mathsf{DeriveKey}\Pi$).

Remark 4 (Domain separation). The requirements $L_1 \neq L_2$ and $|L_1| = |L_2|$ implies that the equal length values $L_1 \parallel N_1$ and $L_2 \parallel N_1$ are distinct for all distinct values of N_1. This guarantees domain separation for the invocations of $F(K, \cdot)$ in the derivation of $\mathsf{K_E}$ and $\mathsf{K_C}$. Different ways to secure this domain separation can be used analogously.

Using the different $\mathsf{CommitKey}\Pi$ flavors. Comparing to the direct use of Π, $\mathsf{DeriveKey}\Pi$ is a method for extending the lifetime of a key, using one nonce-based key derivation (see, e.g., [5] and [2]). $\mathsf{CommitKey}\Pi$ Type I (over Π) and type II (over $\mathsf{CommitKey}\Pi$) carry the lowest incremental overheads due to using a fixed key identifier ($\mathsf{K_C}$) for the main key K. These are useful under the assumption that associating groups of visible encrypted payloads (N, C, T, $\mathsf{K_C}$, N_1) with the same main key does not violate the privacy requirements of the communication (and hence, a fixed key identifier is acceptable). For example, this is the case when a main key is used for only one session between the communicating parties. Deriving a nonce-dependent $\mathsf{K_C}$ value, as in Types III and IV, prevents this association, and comes at some incremental cost (see Figure 1). It is useful with multiple main keys used across multiple payloads, when bundling encrypted payloads under the same main key is undesired.

$\mathsf{CommitKey}\Pi$ **Encryption**
Input: (K, N_1, N, A, M)
$|K| = \kappa_0$, $|N_1| = \nu_1$, $N \in \mathcal{N}$, $A \in \mathcal{A}$, $M \in \mathcal{M}$

1. $\mathsf{K_E} = F_{[\kappa]}(K, L_1)$ (fixed) or $\mathsf{K_E} = F_{[\kappa]}(K, L_1 \parallel N_1)$ (nonce-based)
2. $\mathsf{K_C} = F_{[c]}(K, L_2)$ (fixed) or $\mathsf{K_C} = F_{[c]}(K, L_2 \parallel N_1)$ (nonce-based)
3. $C, T = Enc(\mathsf{K_E}, N, A, M)$
4. Output: $C, T, \mathsf{K_C}$

$\mathsf{CommitKey}\Pi$ **Decryption** Input: $(K, N_1, N, A, C, T, \mathsf{K_C})$
$|K| = \kappa_0$, $|N_1| = \nu_1$, $N \in \mathcal{N}$, $A \in \mathcal{A}$, $C \in \mathcal{C}$, $T \in \mathcal{T}$, $|\mathsf{K_C}| = c$

1. $r_1 = 0$, $r_2 = 0$
2. $\mathsf{K_E}' = F_{[\kappa]}(K, L_1)$ (fixed) or $\mathsf{K_E}' = F_{[\kappa]}(K, L_1 \parallel N_1)$ (nonce-based)
3. $\mathsf{K_C}' = F_{[c]}(K, L_2)$ (fixed) or $\mathsf{K_C}' = F_{[c]}(K, L_2 \parallel N_1)$ (nonce-based)
4. If $\mathsf{K_C}' = \mathsf{K_C}$ then $r_1 = 1$
5. If $Dec(\mathsf{K_E}', N, A, C, T) = M$ (i.e., $\neq \perp$) then $r_2 = 1$
6. If $r_1 \cdot r_2 = 0$ then output $\perp$; else output M

$\mathsf{CommitKey}\Pi$ Type	$\mathsf{K_E}/\mathsf{K_C}$ derivation	Calls to F to encrypt (decrypt) q messages	Communication overhead over $\mathsf{DeriveKey}\Pi$	Communication overhead over Π
I	Fixed/Fixed	$1+1$	c	c
II	Nonce/Fixed	$q+1$	c	$c+\nu_1$
III	Fixed/Nonce	$1+q$	$c+\nu_1$	$c+\nu_2$
IV	Nonce/Nonce	$2q$	$c+\nu_1$	$c+\nu_1$

Fig. 1: **Top:** $\mathsf{CommitKey}\Pi$ (and $\mathsf{DeriveKey}\Pi$) encryption and decyption. $\mathsf{DeriveKey}\Pi$ encryption is obtained (from $\mathsf{CommitKey}\Pi$ encryption) by skipping Step 2, and omitting $\mathsf{K_C}$ from the output. $\mathsf{DeriveKey}\Pi$ decryption is obtained (from $\mathsf{CommitKey}\Pi$ decryption) by ignoring r_2 (setting $r_2 = 1$), $\mathsf{K_C}$, and skipping Steps 3 and 4. Four flavors of $\mathsf{CommitKey}\Pi$ are defined: Type I (fixed $\mathsf{K_E}$, fixed $\mathsf{K_C}$), Type II (nonce-based $\mathsf{K_E}$ fixed $\mathsf{K_C}$), Type III (fixed $\mathsf{K_E}$, nonce-based $\mathsf{K_C}$), Type IV (nonce-based $\mathsf{K_E}$ and nonce-based $\mathsf{K_C}$). For a randomized version, N_1 (and possibly N) is generated uniformly at random from $\{01\}^{\nu_1}$ (instead of being part of the input) during $\mathsf{CommitKey}\Pi$ ($\mathsf{DeriveKey}\Pi$) encryption. In such that case, the generated value is added to the encryption output.
Bottom: The overheads involved with the different flavors of $\mathsf{CommitKey}\Pi$, when encrypting (decrypting) q payloads with the main key K.

3.2 Simple instantiation examples

We provide an example of simple instantiation for F, for the case where $\kappa_0 = \kappa = 256$. Assume that $\nu_1 \leq 256$. Set $c = 256$. Define

$$F(K, L) = \texttt{SHA256}(K \parallel L)$$

For concreteness, define some (fixed) label $L0$ of length 48 bits (6 bytes). A possible example is $L0 = 0x436f6d6d6974$ (=`Commit` in hexadecimal notation). Set:
For Type I: $L_1 = L0 \parallel 0x01 \parallel 0x01$, $L_2 = L0 \parallel 0x01 \parallel 0x02$
For Type II: $L_1 = L0 \parallel 0x02 \parallel 0x01$, $L_2 = L0 \parallel 0x02 \parallel 0x02$
For Type III: $L_1 = L0 \parallel 0x03 \parallel 0x01$, $L_2 = L0 \parallel 0x03 \parallel 0x02$
For Type IV: $L_1 = L0 \parallel 0x04 \parallel 0x01$, $L_2 = L0 \parallel 0x04 \parallel 0x02$

Note that the $\mathsf{CommitKey}\Pi$ flavors are encoded in the labels L_1, L_2. With this choice, $|K \parallel L_1 \parallel N_1| = |K \parallel L_2 \parallel N_1| \leq 576$ so deriving $\mathsf{K_E}$ and $\mathsf{K_C}$ require (for each computation) at most two calls to the `SHA256` compression function. For Type I, computing $\mathsf{K_E}$ and $\mathsf{K_C}$ invokes the `SHA256` compression function only once. For Type II, computing $\mathsf{K_C}$ involves calling the `SHA256` compression function only once (and twice for computing $\mathsf{K_E}$).

A Type I key committing AES-GCM Let Π be the standard AES-GCM scheme with parameters $\kappa = 256$, $\nu = 96$ (and block size $n = 128$). Select parameter values $\kappa_0 = 256$ $(= \kappa)$, $c = 256$, $\ell = \ell_L = 48$ (there is no N_1 nonce, so effectively $\nu_1 = 0$) and consider a Type I scheme. Define $F(K, L) = \texttt{SHA256}(K \parallel L)$ and call it with $L_1 = L0 \parallel 0x01 \parallel 0x01$ and $L_2 = L0 \parallel 0x01 \parallel 0x02$ where $L0 = 0x41455347434d$ (=`AESGCM` in hexadecimal notation). The resulting $\mathsf{CommitKey}\Pi$ (Type I) is a "Robust Key AES-GCM" denoted here, for brevity, by RK-AES-GCM. At setup, RK-AES-GCM encryption requires AES key expansion for the encryption key $\mathsf{K_E}$, and also one computation of $AES(\mathsf{K_E}, 0^{128})$ for the GHASH key. This setup overhead is the same as the setup for AES-GCM.

4 Analysis of CommitKeyΠ

Our analysis of $\mathsf{CommitKey}\Pi$ consists of two parts: a) upper bounding the privacy and authenticity guarantees of $\mathsf{CommitKey}\Pi$, compared to the underlying scheme Π (or to $\mathsf{DeriveKey}\Pi$); b) analysis for the key commitment property that $\mathsf{CommitKey}\Pi$ offers.

Upper bounds for the ciphertext indistinguishability of CommitKeyΠ. Theorem 1 is stated for $\mathsf{CommitKey}\Pi$ Type IV, where we assume, for convenience, that $\kappa = c$. The statements for Types I, II, III are analogous.

We start with a few definitions. The oracle for Π encryption is denoted by $\mathcal{O}_\Pi$. A privacy adversary $\mathbf{A}$ against (the privacy of) Π submits encryption queries of the form (N, A, M) to $\mathcal{O}_\Pi$, and its advantage is denoted $Adv_\Pi^{priv}(\mathbf{A})$.

In the fresh-multi-key scenario, the oracle is denoted by $\mathcal{O}_{mk-\Pi}$ and the advantage is denoted by $Adv^{priv}_{mk-\Pi}(\mathbf{A})$. The oracle for F is denoted by $\mathcal{O}_F$. An adversary $\mathbf{B}$ against the PRF security of F submits queries of length $\ell_L + \nu_1$ to $\mathcal{O}_F$ and its advantage is denoted by $Adv^{PRF}_F(\mathbf{B})$. The oracle for $\mathsf{CommitKey}\Pi$ encryption is denoted by $\mathcal{O}_{\mathsf{CommitKey}\Pi}$. An adversary $\mathbf{A}$ against (the privacy of) $\mathsf{CommitKey}\Pi$ submits queries of the form (N_1, N, A, M) and receives either C, T, $\mathsf{K_C}$ or $R_\$ \leftarrow_\$ \{0,1\}^{|M|+\tau+c}$, depending on $\mathcal{O}_{\mathsf{CommitKey}\Pi}$'s challenge bit. Its advantage is denoted $Adv^{priv}_{\mathsf{CommitKey}\Pi}(\mathbf{A})$. The scheme $\mathsf{CommitKey}\Pi'$ is a "fresh key" analogue to $\mathsf{CommitKey}\Pi$. Its oracle is denoted by $\mathcal{O}_{\mathsf{CommitKey}\Pi'}$, and it selects uniformly random $\mathsf{K_E}$ and $\mathsf{K_C}$ values for every encryption query. A privacy adversary $\mathbf{A}$ submits encryption queries of the form (N_1, N, A, M) to $\mathcal{O}_{\mathsf{CommitKey}\Pi'}$, and its advantage is denoted by $Adv^{priv}_{\mathsf{CommitKey}\Pi'}(\mathbf{A})$. Note that from the indistinguishability viewpoint, $\mathsf{CommitKey}\Pi'$ is essentially equivalent to Π in the fresh-multi-key setting (the uniformly random $\mathsf{K_C}$ value appended to C and T has no impact on the distinguishing advantage).

Theorem 1 ($\mathsf{CommitKey}\Pi$ Privacy). *Let $\mathbf{A}$ be a privacy adversary against $\mathsf{CommitKey}\Pi$ Type IV. Let q, ℓ_A, ℓ_M, $\ell_{payload}$ be non-negative parameters, and assume, for convenience, that $\kappa = c$. Assume that $\mathbf{A}$ submits at most q encryption queries of the form (N_1, N, A, M), without repeating N_1 values, such that $|A| \leq \ell_A$, $|M| \leq \ell_M$, and the total encrypted payload, across all queries, is at most $\ell_{payload}$. Then, there exist: a) an adversary $\mathbf{B}$ against the PRF security of F that makes at most $2q$ queries of length $\ell_L+\nu_1$ to its oracle; b) a privacy adversary $\mathbf{A}'$ against Π in the fresh-multi-key setting, that makes at most q queries of the form (N, A, M) with $|A| \leq \ell_A$ and $|M| \leq \ell_M$, and overall encrypted payload of at most $\ell_{payload}$, such that*

$$Adv^{priv}_{\mathsf{CommitKey}\Pi}(\mathbf{A}) \leq Adv^{PRF}_F(\mathbf{B}) + Adv^{priv}_{mk-\Pi}(\mathbf{A}') \tag{2}$$

If $\mathbf{A}$ runs in $t_\mathbf{A}$ steps, then $\mathbf{B}$ runs in $O(t_\mathbf{A})+\Delta_1$ steps and $\mathbf{A}'$ runs in $O(t_\mathbf{A})+\Delta_2$ steps where: a) Δ_1 is the number of steps required to simulate $\mathbf{A}$' (at most q) queries with a given/known key; b) Δ_2 is the number of steps required generate (at most) q random values of length c.

Proof. We first build an adversary $\mathbf{B}$ against the PRF security of F, running against $\mathcal{O}_F$. $\mathbf{B}$ runs $\mathbf{A}$. For every query (N_1, N, A, M) that $\mathbf{A}$ issues, $\mathbf{B}$ queries $\mathcal{O}_F$ with the values $L_1 \parallel N_1$ and $L_2 \parallel N_1$ to obtain the response $X \in \{0,1\}^\kappa$ and $Y \in \{0,1\}^c$. Subsequently, $\mathbf{B}$ computes $Enc(X, N, A, M) = (C, T)$ and returns C, T, Y to $\mathbf{A}$. When $\mathbf{A}$ outputs a bit b', $\mathbf{B}$ outputs b' and stops. Denote the sequence of queries issued by $\mathbf{A}$ until it outputs b' by SEQ, and let $P_\$$ be the probability that $\mathbf{A}$ outputs $b' = 1$ if SEQ is replied with uniformly random responses (of the expected lengths). We have, by the definition

$$Adv^{PRF}_F(\mathbf{B}) = \left|Prob(b' = 1|\ b = 1) - Prob(b' = 1|\ b = 0)\right| \tag{3}$$

Note that if $b = 1$, the responses returned to $\mathbf{A}$ simulate (real) responses to SEQ from $\mathcal{O}_{\mathsf{CommitKey}\Pi}$ for $\mathsf{CommitKey}\Pi$. If $b = 0$, these responses simulate (real) responses to SEQ from $\mathcal{O}_{\mathsf{CommitKey}\Pi'}$ (in the fresh-multi-key setting). Therefore,

$$\left|Prob(b'=1|\ b=1) - Prob(b'=1|\ b=0)\right| =$$
$$\left|Prob(b'=1|\ b=1) - P_{\$} - \left(Prob(b'=1|\ b=0) - P_{\$}\right)\right| \geq$$
$$Adv^{priv}_{\mathsf{CommitKey}\Pi}(\mathbf{A}) - Adv^{priv}_{\mathsf{CommitKey}\Pi'}(\mathbf{A}) \tag{4}$$

and so, we have

$$Adv^{priv}_{\mathsf{CommitKey}\Pi}(\mathbf{A}) \leq Adv^{PRF}_{F}(\mathbf{B}) + Adv^{priv}_{\mathsf{CommitKey}\Pi'}(\mathbf{A}) \tag{5}$$

We now build an adversary $\mathbf{A}'$ against $\mathcal{O}_{mk-\Pi}$ (i.e., Π in the fresh-multi-key setting). $\mathbf{A}'$ runs $\mathbf{A}$. For every query (N_1, N, A, M) that $\mathbf{A}$ issues, $\mathbf{A}'$ queries $\mathcal{O}_{mk-\Pi}$ with (N, A, M) and obtains the response C, T. It then generates a fresh value $Y \leftarrow_{\$} \{0,1\}^c$ and returns C, T, Y to $\mathbf{A}$. When $\mathbf{A}$ outputs a bit b', $\mathbf{A}'$ outputs b' and stops. Since Y is a uniformly random value, we have

$$Adv^{priv}_{\mathsf{CommitKey}\Pi'}(\mathbf{A}) = Adv^{priv}_{mk-\Pi}(\mathbf{A}')$$

The number of steps that $\mathbf{B}$ and $\mathbf{A}'$ run, and the lengths of the queries, are clear from the above description. □

The Key commitment property of $\mathsf{CommitKey}\Pi$**.** The $\mathsf{CommitKey}\Pi$ construction is designed to address the following scenario:

> A polynomial time Adversary $\mathbf{A}''$ against the key identification string $\mathsf{K_C}$ chooses distinct main keys K_1, K_2, and a tuple $(N_1, N, A, C, T, \mathsf{K_C})$. It wins if $(N_1, N, A, C, T, \mathsf{K_C})$ passes the $\mathsf{CommitKey}\Pi$ authentication under K_1 and also under K_2, as main keys.

Claim. If adversary $\mathbf{A}''$ produces a winning tuple $(N_1, N, A, C, T, \mathsf{K_C})$ for keys $K_1 \neq K_2$, then $\mathbf{A}''$ has found a collision (on $\mathsf{K_C}$), i.e.,

$$F_{[c]}(K_1, L_2 \parallel N_1) = F_{[c]}(K_2, L_2 \parallel N_1) \tag{6}$$

Remark 5. It may be possible (or even easy) to find a tuple $(N_1, N, A, C, T, \mathsf{K_C})$ and two main keys $K_1 \neq K_2$, such that (N, A, C, T) passes Π authentication under the $\mathsf{K_E}$ values that are derived from K_1 and from K_2. This ability depends on the properties of the underlying AEAD Π. The introduction of $\mathsf{K_C}$ (as in $\mathsf{CommitKey}\Pi$) adds the requirement (6) for the full tuple $(N_1, N, A, C, T, \mathsf{K_C})$.

Remark 6. We may relax the requirement on $\mathbf{A}''$ and allow a choice of different N_1 values. Here, $\mathbf{A}''$ can choose distinct main keys K_1, K_2, and two tuples $(N_1, N, A, C, T, \mathsf{K_C})$, $(N_1', N, A, C, T, \mathsf{K_C})$. $\mathbf{A}''$ wins if $(N_1, N, A, C, T, \mathsf{K_C})$ passes the $\mathsf{CommitKey}\Pi$ authentication under K_1 as the main key, and
$(N_1', N, A, C, T, \mathsf{K_C})$ passes the $\mathsf{CommitKey}\Pi$ authentication under K_2 as the main key. If that case, to win, $\mathbf{A}''$ needs to find a collision

$$F_{[c]}(K_1, L_2 \parallel N_1) = F_{[c]}(K_2, L_2 \parallel N_1') \tag{7}$$

Collision resistance requirements from F. To meet the $\mathsf{CommitKey}\Pi$ design goal, the pseudorandom function $F(\cdot,\cdot)$ should be chosen in a way that an adversary with an assumed (reasonable) compute time has a negligible probability to produce a *collision* of type (6), even with its (adversarial) chosen keys. In particular, c should be sufficiently large so that brute force attempts until a collision occurs is practically unfeasible. Note that the choice $F(K,L) = \mathtt{SHA256}(K \parallel L)$ (shown in Section 3.2) is based on a collision resistant hash function. It satisfies the requirement, under the standard assumption on SHA256. Note also that SHA256 prevents even a collision of the type (7), under the same assumption.

5 Discussion

We give an example for using the bounds of Theorem 1 for a scenario of interest. Consider the case where $\kappa_0 = \kappa = c = 256$ and Π is the standard AES-GCM (with a 96-bit nonce). Suppose that a main key K is used $q \leq 2^{32}$ times, with different nonces $N_{1_1}, \ldots N_{1_q}$ and derived values $\mathsf{K_E}_{1_1} \ldots \mathsf{K_E}_{1_q}$ and $\mathsf{K_C}_{1_1} \ldots \mathsf{K_C}_{1_q}$. We assume that all the derived values are distinct. Every nonce from $N_{1_1}, \ldots$ N_{1_q}(and the respective derived key is used for encrypting a payload with the following characteristics. Payload j consists of $\bar{q}_j$ chunks of data. Every chunk is encrypted with AES-GCM under the key $\mathsf{K_E}_j$, using a different AES-GCM nonce (N). The total number of blocks encrypted with $\mathsf{K_E}_j$ is σ_j. For $\mathsf{CommitKey}\Pi$, we make the assumption that AES behaves like an ideal cipher in the multi key scenario, and ignore the PRP advantage of distinguishing AES from a random permutation on $\{0,1\}^{128}$. With probability at most $(2q)^2/2^{\kappa+1}$, we may assume that the q values of $\mathsf{K_E}$ and the q values of $\mathsf{K_C}$ are distinct. With T_0 key guessing attempts (for either K or a derive $\mathsf{K_E}$), correct guessing succeeds with probability $(T_0 q)/2^{\kappa}$. Therefore, we can upper bound the advantage of a privacy adversary against $\mathsf{CommitKey}\Pi$ by

$$\max Adv_F^{PRF} + \frac{4q^2}{2^{\kappa+1}} + T_0 \frac{q}{2^{\kappa}} + \sum_{j=1}^{q} \frac{(\sigma_j + \bar{q}_j + 1)^2}{2^{129}} \tag{8}$$

where $\max Adv_F^{PRF}$ is the maximum distinguishing advantage for F, with $2q$ queries.

We set the limits $q \leq 2^{32}$, $\bar{q}_j = 2^{30}$ and $\sigma_j = 2^{30}$, $j = 1, \ldots, q$, and assume $T_0 \leq 2^{96}$. This implies $(\sigma_j + \bar{q}_j + 1) < 2^{32}$, and consequently, the dominant term in (8) is at most $2^{32} \times 2^{-65} = 2^{-33}$. With a judicious choice for a PRF F (e.g., $F(K,L) = \mathtt{SHA256}(K \parallel L)$ as in Section 3.2), we can assume that the PRF distinguishing advantage with $2q$ queries (for F) is or order $O(4q^2/2^{257})$. The amount of data that can be encrypted using $\mathsf{CommitKey}\Pi$ and a main key K, is up to 2^{60} blocks (i.e., 2^{64} bytes), and the indistinguishability bound is at most $O(2^{-32})$.

Design rationale and alternatives. We require $\mathsf{CommitKey}\Pi$ to use $\kappa_0 \geq \kappa$ in order to keep a key hierarchy: the derived encryption keys ($\mathsf{K_E}$) are not longer than the main key. Similarly, we require $\kappa_0 \geq \mathsf{K_C}$ and set $\mathsf{K_C}$ to be sufficiently long in order to make brute force collision and pre-image search unfeasible. The power-of-two choice $\kappa_0 = \kappa = c = 256$ seems adequate and convenient. However, it is also reasonable to settle with $c = 192$ or 160 to reduce the overhead of $\mathsf{CommitKey}\mathit{\Pi}$ encryption. We point out that defining $F(K, L) = \mathtt{H}(K \parallel L)$ with any NIST standard cryptographic hash function H, with a sufficiently long digest, is an acceptable choice (the example in Section 3.2 uses SHA256). This makes it is easy to choose a main key (K) of a desired length, and also, under standard assumptions on the hash function, to truncate the digests to c or κ bits, as needed. Note that it is implicitly assumed here that for this usage, H is invoked with equal-length arguments. It is also possible to choose other designs where $F(K, L) = \mathtt{HMAC}(K, L)$ or $F(K, L) = \mathtt{HKDF}(K, L)$. Due to their construction that is based on a collision resistant hash function, these options also satisfy the collision resistance requirement mentioned in Section 4 (even with chosen keys). In such cases, the requirement for equal-length arguments can be relaxed. We point out that $\mathsf{CommitKey}\mathit{\Pi}$ does not require that the checks for T and for $\mathsf{K_C}$ (see Steps 4, 5, 6 in Figure 1) are executed in constant time or in a particular order. An implementation can choose to return $\perp$ as soon as one comparison does not match.

Finally, we point out a theoretical difference between the $\mathsf{CommitKey}\mathit{\Pi}$ constructions of Type I and of Type IV. For Type I, the collision analogous to (6) for Type IV, is $F_{[c]}(K_1, L_2) = F_{[c]}(K_2, L_2)$ (i.e., no N_1 nonce is involved). This means that for Type I, a selection of a pair K_1 K_2 already determines the existence of (or lack of) a collision. By contrast, for an adversary $\mathbf{A}''$ for Type IV $\mathsf{CommitKey}\mathit{\Pi}$, can choose a pair K_1 K_2, and then still have the freedom to select a value N_1 that yields a collision of the form (6).

Acknowledgments. This research was supported by: NSF-BSF Grant 2018640; The Israel Science Foundation (grant No. 3380/19); The BIU Center for Research in Applied Cryptography and Cyber Security, and the Center for Cyber Law and Policy at the University of Haifa, both in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

References

1. AWS: AWS Encryption SDK. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/message-format.html (2020)
2. Campagna, M., Gueron, S.: Key management systems at the cloud scale. Cryptogr. **3**(3), 23 (2019). https://doi.org/10.3390/cryptography3030023, https://doi.org/10.3390/cryptography3030023
3. Dodis, Y., Grubbs, P., Ristenpart, T., Woodage, J.: Fast message franking: From invisible salamanders to encryptment. Cryptology ePrint Archive, Report 2019/016 (2019), https://eprint.iacr.org/2019/016
4. Grubbs, P., Lu, J., Ristenpart, T.: Message franking via committing authenticated encryption. Cryptology ePrint Archive, Report 2017/664 (2017), https://eprint.iacr.org/2017/664

5. Gueron, S., Lindell, Y.: Better bounds for block cipher modes of operation via nonce-based key derivation. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 1019–1036. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3133992, https://doi.org/10.1145/3133956.3133992
6. Krawczyk, H.: The OPAQUE Asymmetric PAKE Protocol; draft-krawczyk-cfrg-opaque-03. https://tools.ietf.org/html/draft-krawczyk-cfrg-opaque-03#section-3.1.1 (2020)