How to Abuse and Fix Authenticated Encryption Without Key Commitment

Ange Albertini[1], Thai Duong[1], Shay Gueron[2,3], Stefan Kölbl[1], Atul Luykx[1], and Sophie Schmieg[1]

[1]Security Engineering Research, Google
[2]University of Haifa
[3]Amazon

Abstract

Authenticated encryption (AE) is used in a wide variety of applications, potentially in settings for which it was not originally designed. Recent research tries to understand what happens when AE is not used as prescribed by its designers. A question given relatively little attention is whether an AE scheme guarantees "key commitment": ciphertext should decrypt to a valid plaintext only under the key that was used to generate the ciphertext. As key commitment is not part of AE's design goal, AE schemes in general do not satisfy it. Nevertheless, one would not expect this seemingly obscure property to have much impact on the security of actual products. In reality, however, products do rely on key commitment. We discuss three recent applications where missing key commitment is exploitable in practice. We provide proof-of-concept attacks via a tool that constructs AES-GCM ciphertext which can be decrypted to two plaintexts valid under a wide variety of file formats, such as PDF, Windows executables, and DICOM. Finally we discuss two solutions to add key commitment to AE schemes which have not been analyzed in the literature: one is a generic approach that adds an explicit key commitment scheme to the AE scheme, and the other is a simple fix which works for AE schemes like AES-GCM and ChaCha20Poly1305, but requires separate analysis for each scheme.

1 Introduction

Authenticated Encryption. Symmetric-key encryption (SKE) has been the source of many attacks over the years. The main culprit is the use of malleable, unauthenticated schemes like CBC, and their susceptibility to padding oracle [Vau02] and related attacks. Such attacks are found with as much regularity against systems designed in the 90's as they are today; recent research [FIM20] shows that CBC continues to be an attack vector.

Beck et al. [BZG20] cite flaws in Apple iMessage, OpenPGP, and PDF encryption as examples to argue that practitioners are often only convinced that unauthenticated SKE is insecure when they see a proof-of-concept exploit. Similar efforts are deemed necessary to demonstrate the exploitability of cryptographic algorithms such as SHA-1 [SBK+17].

The vast majority of applications should default to using authenticated encryption (AE) [BN00, KY00], a well-studied primitive which avoids the pitfalls of unauthenticated SKE with relatively small performance overhead. AE schemes are used in widely adopted protocols like TLS [Res18], standardized by NIST [NIS07a, NIS07b] and ISO [ISO09], and are the default SKE option in modern cryptographic libraries such as NaCl [nac] and Tink [tin].

With AE more widely used, recent research focuses on its security guarantees in settings which push the boundaries and assumptions of conventional AE, such as understanding nonces [RS06], multiple decryption errors [BDPS13], unverified plaintext [ABL+14], side channel leakage [BMOS17], multi-user attacks [BT16], boundary hiding [BDPS12], streaming AE [HRRV15], and variable-length tags [RVV16]. Furthermore, constructions and security models have received additional scrutiny due to two recent competitions focusing on AE: CAESAR [CAE14] and the NIST lightweight cryptography competition [nis].

Key Commitment. Among the extended, desirable properties explored is the relatively little-studied of AE *key commitment*, which we intuitively explain as follows.

One of the defining design goals of AE is to provide ciphertext integrity: if recipient A decrypts a ciphertext with the key K_A into a valid plaintext, meaning authentication succeeds, then A knows that the ciphertext has not been modified during transmission. Intuitively, one might mistakenly assume that the integrity guarantee extends to keys, i.e., if some other recipient B decrypts the same ciphertext with their key K_B, then decryption would fail. However, this is neither an AE design goal, nor a guaranteed property, and there are secure and globally deployed AE schemes where both recipients can successfully decrypt the same ciphertext.

Key commitment guarantees that a ciphertext can only be decrypted under the same key used to produce it from some

plaintext. Schemes where it is possible to find a ciphertext which decrypts to valid plaintexts under two different keys do *not* commit to the key.

Initially studied and formalized for AE by Farshim et al. [FOR17] under the name "robustness", key commitment might seem like an academic pursuit. Yet Dodis et al. [DGRW18] and Grubbs et al. [GLR17a] show how to exploit AE schemes which do not commit to the key in the context of abuse reporting in Facebook Messenger. Nevertheless, AE key commitment has not received much attention and the concept can be overlooked during deployment.

1.1 Contributions

Facebook Messenger might seem like a niche use of AE which implicitly relies on key commitment, and was exploitable. However, we show that is not the case. We conduct a thorough study of AE key commitment to understand:

What settings or products are vulnerable to attacks that exploit AE schemes which do not commit to the key?
We found three settings in the past year: key rotation in key management services, envelope encryption, and Subscribe with Google [Albb] (see Section 2). In concurrent work, Len et al. [LGR] found another. We expect there to be more.

How easy is it to exploit lack of key commitment in practice?
We study deployed and standardized AE schemes — such as AES-GCM, AES-GCM-SIV, ChaCha20Poly1305, OCB — summarize known key commitment attacks, and introduce new ones. The cryptographic attacks place restrictions on adversarially generated plaintext and ciphertext (see Section 3.3); we exploit this is by using *binary polyglots*, files whose contents are valid according to two different file formats. Whereas Dodis et al. [DGRW18] demonstrate binary polyglots for JPEG and BMP, we show how to do the same for many other file formats: we create a tool[1] to mix files of specific file formats which support and identifies more than 40 formats, then tries to combine the input contents following various layouts, resulting in working binary polyglots made of more than 250 format combinations (see Section 4). Combined with another tool we made, we demonstrate how to turn the binary polyglot into AES-GCM ciphertext.

Are there simple and efficient ways to add key commitment to AE schemes?
Farshim et al. [FOR17], Grubbs et al. [GLR17a], and Dodis et al. [DGRW18] present both generic and optimized encryption algorithms which include key commitment. However, none achieve the efficiency of AES-GCM, and require changes to the cryptographic algorithms used[2].

We propose simple solutions which have not been analyzed in the literature — amounting to black-box use of the AE schemes, with one additional block output — and analyze their security. One solution simply prepends a constant block of all zero's to the plaintext and encrypts the padded plaintext as normal; decryption looks for the presence of a leading block of zero's to verify the correct key was used. This padding solution does not necessarily work for any AE scheme and must be analyzed on a case-by-case basis, which we do for AES-GCM and ChaCha20Poly1305.

Another solution applies a generic composition to any given AE: the scheme's key is first used to derive a key commitment string and an encryption key; the encryption key is then used in the underlying AE scheme; the scheme outputs the ciphertext and the commitment string.

An instantiation of our generic composition is already publicly deployed as part of the latest version (2.0) of the AWS Encryption SDK [AWSa], an open source client-side encryption library. Key commitment is included in its default configuration. More details can be found in [Tri].

2 Real-World Settings

We highlight several real-world scenarios, where lack of key-commitment of an AE scheme could lead to vulnerabilities. Note that these attacks do not break any properties of the underlying AE scheme, but rely on the fact their applications implicitly assume that these schemes are key-committing. Any vulnerabilities found as a result of this work have been responsibly disclosed (CVE pending).

Key Rotation. A key management service (KMS) creates, removes, controls access to, and audits usage of cryptographic keys. In such a service a cryptographic key is typically accessed through a URI, which the user needs to identify and access a key. One important feature here is key rotation, which allows to update a key in order to limit the amount of data encrypted under a single key and reduce damage in case of a compromise.

After a key rotation, the old key should still be able to decrypt old ciphertexts but not be used to encrypt any new data. This implies that different versions of a key exist and there must be some mechanism to decide which key is used for encryption/decryption. If the AE used for encrypting the data is not committing to a key, then this can potentially be exploited by an attacker. A user could assume that a ciphertext will decrypt to the same plaintext, independent of key rotations happening, however this might not be the case in practice.

The scenario we are interested in here is, multiple users are accessing a key through a URI. One of the users is malicious and wants to distribute e.g., a malicious file and the AE

[1] https://github.com/corkami/mitra

[2] In fact, as explained by Dodis et al. [DGRW18], performance was the primary reason Facebook Messenger used AES-GCM for attachments, making Facebook Messenger vulnerable to attack, despite the fact that Grubbs et al. [GLR17a] had already proposed secure alternatives.

scheme used is AES-GCM. The attacker proceeds as follows. First, create two keys K_1, K_2 and produce a ciphertext C which decrypts to a *"good"* file M under K_1 and to a *"bad"* file M' under K_2. Next, import K_1 into the KMS and send everyone the ciphertext C which they can store and decrypt to M when needed. At a later point in time, the adversary imports K_2 to the KMS.

Now at this stage the question arises which key will be used to decrypt C if a user calls the KMS API with the key URI. There are different options how this can be implemented:

1. Add metadata to the ciphertext to identify the exact key which is used. This adds overhead to the ciphertext, but if the metadata is ensured to be authentic this can be used to ensure that K_1 will be used to decrypt C.

2. Try out the keys until one successfully decrypts, starting with oldest version. In this case K_1 would successfully, decrypt and reveal M.

3. Try out the keys until one successfully decrypts, starting with newest version. In this case K_2 would successfully, decrypt and reveal the malicious file M'.

4. Allow the user to select the key version used to decrypt.

Note that if the adversary may delete/disable old key versions, a solution relying on (2) can still cause a decryption of C to the malicious file M'. This gives the adversary a simple trigger to enable/disable when the ciphertext should be decrypted to harmful content. The user will not detect that a different key was used, as the decryption is authentic.

Envelope Encryption. As cloud computing continues to see significant growth, a wider variety of applications come to rely on it, in more diverse settings. As a result, the encryption mechanisms used by cloud services need to be scrutinized to ensure they can be used securely in settings for which they might not have been designed.

Take for example envelope encryption, where anytime data is sent or stored, a symmetric key is generated to encrypt the data, and the key is in turn encrypted under multiple recipient keys, which could be either symmetric or asymmetric (i.e. a KEM). All major cloud service providers use envelope encryption, and typically use an AE scheme like AES-GCM for the symmetric encryption; see for example AWS [awsb] and Google Cloud [goo].

Envelope encryption users often — intuitively — expect that if the recipients receive the same ciphertext, then all will decrypt to the same plaintext. However this expectation is false: cloud services without key commitment can fall victim to attacks, where the same ciphertext will decrypt to different plaintexts under different keys. The AWS encryption SDK has been vulnerable to this and as a result the option for a key commitment has been added [Tri].

The encryption of a message for two users can be summarized as follows. First, a random data encryption key K_{DEK} is generated and wrapped by the two users keys which are provided through the `encrypt` API. Next, a per-message AES-GCM key K is derived using HKDF from K_{DEK}, a randomly generated message ID and fixed algorithm ID. A header is formed from the wrapped keys, the encryption context and other metadata. The header is authenticated using AES-GMAC with K and zero IV. The message M is then encrypted using AES-GCM with K, non-zero IV and fixed associated data. In the end this gives us a ciphertext which consists of a header H, header tag H_T, encrypted message C and authentication tag T. In order to decrypt, the SDK loops over the wrapped keys and returns the first one which it can successfully unwrap, which is then used to decrypt the ciphertext and obtain the message.

An attacker which wants to send a different messages M, M' to two recipients, can do so by exploiting the lack of key commitment in GCM/GMAC. The attacker generates a random pair of $(K_{\text{DEK}}, K'_{\text{DEK}})$, derives (K, K') and encrypts (M, M') such that they form a single ciphertext C with a single valid authentication tag T (see Section 3.3 for details on this). The attacker then wraps K_{DEK} for one user and K'_{DEK} for the other user. At last there is still the header H and tag H_T which need to be valid. For this we can use the same approach as for the ciphertext encryption, as GMAC is used for authentication. The encryption context can be used as an additional block and allows to *correct* the authentication tag, such that it is valid for both K and K'.

Subscribe with Google [Albb]. SwG is a service which allows users to subscribe to publications using Google accounts. Users pay to access "premium" content. Paying users see the content immediately, while others might see a preview, or nothing. Either the publisher or third party authorizers give users access to the premium content; examples of third party authorizers include a search indexer, content distribution network, or a third-party paywall service.

Publishers include both premium and preview content in a single document, with the premium content encrypted [Cry]. To do so, the publisher creates a random symmetric key, the *document key*, and a structure that includes the document key with access requirements, the *document crypt*. The document key encrypts the premium content using an AE scheme, and the document crypt is encrypted under the authorizers' public keys. The encrypted document crypts are placed in the document's header.

Whenever a client requests authorization, the authorizer decrypts the document crypt and checks the access requirements. If a client may access the premium content, then the document key is used to decrypt the content.

Analogous to the envelope encryption setting, if the AE scheme used to encrypt the premium content with the document key does not include a key commitment, then malicious

publishers can display different content to different authorizers: prepare multiple document keys and a ciphertext which decrypts to different plaintexts under those keys; place the different document keys in different document crypts; when an authorizer decrypts its document crypt, it will receive its own document key, and therefore will see its own view of the decrypted premium content.

Initially, SwG was designed to use an AE scheme which did not have a key commitment. This issue was caught before launch and fixed by including a key commitment.

3 Authenticated Encryption and Key Commitment

3.1 Authenticated Encryption Schemes

Authenticated encryption with associated data, which we simply call AE, consists of stateless, deterministic encryption (Enc) and decryption (Dec) algorithms, where the decryption algorithm may output either plaintext or a single, pre-defined error symbol:

$$\mathsf{Enc} : \mathsf{K} \times \mathsf{N} \times \mathsf{A} \times \mathsf{M} \to \mathsf{C}, \tag{1}$$

$$\mathsf{Dec} : \mathsf{K} \times \mathsf{N} \times \mathsf{A} \times \mathsf{C} \to \mathsf{M} \cup \{\perp\}, \tag{2}$$

with K the keys, N the nonces, A the associated data, M the messages, C the ciphertexts, and $\perp$ an error symbol not contained in M, which represents verification failure. It must be the case that for all $K \in \mathsf{K}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$,

$$\mathsf{Dec}(K,N,A,\mathsf{Enc}(K,N,A,M)) = M \quad . \tag{3}$$

Appendix B reviews AE security.

3.2 AE Key Commitment Definition

AE schemes with a key commitment are 'collision resistant' in the sense that it is (computationally) difficult to find two keys which, equivalently, either

1. encrypt two plaintexts to the same ciphertext, or
2. decrypt the same ciphertext to two plaintexts.

We follow Farshim et al.'s [FOR17] formalization. Since we focus on concrete bounds, we only define an adversary's advantage in breaking an AE scheme's key commitment and refrain from saying when a scheme commits to the key. Our results allow users to pick parameters according to their security needs.

Definition 1 (Key Commitment Advantage)**.** Let $\Pi = (\mathsf{Enc}, \mathsf{Dec})$ denote an AE scheme. Let **A** be an adversary interacting with Π; let $Q_1, Q_2, \ldots$ denote the sequence of queries **A** makes to either Enc or Dec, where $Q_i = (K_i, N_i, A_i, M_i, C_i)$ and $\mathsf{Enc}(K_i, N_i, A_i, M_i) = C_i$ or $\mathsf{Dec}(K_i, N_i, A_i, C_i) = M_i$. Then **A**'s q-KC advantage against Π is the probability that there are two queries Q_i and Q_j where $K_i \neq K_j$, $C_i = C_j \neq \perp$, $M_i \neq \perp$, $M_j \neq \perp$, and $i, j \leq q$.

3.3 Absence of Key Commitment in AE schemes

We show that several commonly used AE schemes AES-GCM, ChaCha20Poly1305, AES-GCM-SIV and OCB3 do not commit to their keys. Note that this is not a security goal of these schemes and we also do not have to violate any security property of these schemes to achieve this.

Apart from OCB, all these schemes produce the ciphertext by generating a (pseudorandom) key stream and XORing it with the plaintext. Two different keys K_1, K_2 produce different key streams S_1, S_2, and for a given ciphertext C this decrypts to $M_1 = S_1 + C$ and $M_2 = S_2 + C$. To mount a successful attack, we have to ensure that the authentication passes and that the given C and authentication tag T are valid.

For AES-GCM, an attack has already been shown in [DGRW18], which we briefly describe in a general form here for schemes which use a polynomial MAC (like ChaCha20Poly1305) over the ciphertext. The general construction is as follows:

1. Derive two keys (r_1, s_1) from K_1 (resp. (r_2, s_2) from K_2).
2. Split the ciphertext in blocks $C[1], \ldots, C[m]$.
3. Compute the tag as $T = s_1 + \sum_{i=1}^{m} C[i] \cdot r_1^{m-i}$, where addition/multiplication is done over a finite field.

To generate valid tags with such an authentication scheme, we have to ensure that the given ciphertext leads to the same tag being computed under K_1 and K_2. We fix all ciphertext blocks apart from a single block $C[j]$, which gives us the following equation:

$$\begin{aligned} s_1 + C[j] \cdot r_1^{m-j} + \sum_{i=1, i\neq j}^{m} C[i] \cdot r_1^{m-i} = \\ s_2 + C[j] \cdot r_2^{m-j} + \sum_{i=1, i\neq j}^{m} C[i] \cdot r_2^{m-i}. \end{aligned} \tag{4}$$

All the variables here are known to the adversary, therefore this equation can be solved for $C[j]$

$$\begin{aligned} C[j] = (r_1^{m-j} + r_2^{m-j})^{-1} + s_1 + s_2 + \\ \sum_{i=1, i\neq j}^{m} C[i] \cdot r_1^{m-i} + C[i] \cdot r_2^{m-i}, \end{aligned} \tag{5}$$

which fully determines C and T.

For AES-GCM-SIV the construction is a bit different, as the polynomial MAC is computed over the plaintext, which is then XORed with the nonce and encrypted to get the tag T (see [GLL19]). T is then further used as the first counter

block for encryption. In this case we will first fix T in order to fix the corresponding key streams S_1, S_2. Next, we decrypt the tag with K_1, K_2 and XOR the nonce to obtain T_1 and T_2:

$$T_1 = \sum_{i=1}^{m} M_1[i] \cdot r_1^{m-i} \quad \text{and} \quad T_2 = \sum_{i=1}^{m} M_2[i] \cdot r_2^{m-i}. \tag{6}$$

Additionally, we have the condition that the ciphertext should be equal after adding the key streams, therefore we get m equations of the form

$$\begin{aligned} M_1[1] + S_1[1] &= M_2[1] + S_2[1] \\ M_1[2] + S_1[2] &= M_2[2] + S_2[2] \\ &\vdots \\ M_1[m] + S_1[m] &= M_2[m] + S_2[m]. \end{aligned} \tag{7}$$

In total this gives us $m+2$ linear equations in $2m$ variables (the plaintext blocks), which we can find a solution for if $m > 1$. In general this still gives us a lot of freedom in the message blocks as for longer messages we can fix parts and still find a solution to the system of linear equations.

As a final example we consider OCB3 [KR14], which does not follow the paradigm of creating a key stream and is therefore a particularly interesting case. It is also one of the most efficient AE schemes and has become popular in the variant θCB using a tweakable block cipher. For example Deoxys [JNP15] from the final CAESAR portfolio uses a similar mode and several candidates in the ongoing NIST Lightweight Competition are based on it.

The tag is computed as a simple checksum which is then encrypted. We can start with a similar approach to AES-GCM-SIV and will first fix the tag T and the message length m in order to be able to compute all the mask values Z. We can then decrypt the tag under the two keys and apply the mask which gives us

$$T_1 = \sum_{i=1}^{m} M_1[i] \quad \text{and} \quad T_2 = \sum_{i=1}^{m} M_2[i]. \tag{8}$$

Each message block $M[i]$ is encrypted as $C[i] = E_K(M[i] \oplus Z[i]) \oplus Z[i] = E_{K,Z[i]}(M[i])$ for some mask values $Z[i]$ (the concrete values for $Z[i]$ are not important for the attack here, and therefore it can also be instantiated with a tweakable block cipher). Hence we get equations of the form

$$\begin{aligned} E_{K_1,Z[1]}(M_1[1]) &= E_{K_2,Z[1]}(M_2[1]) \\ E_{K_1,Z[2]}(M_1[2]) &= E_{K_2,Z[2]}(M_2[2]) \\ &\vdots \\ E_{K_1,Z[m]}(M_1[m]) &= E_{K_2,Z[m]}(M_2[m]), \end{aligned} \tag{9}$$

if we want to have the same ciphertext. However as these equations are non-linear the approach used for AES-GCM-SIV can not work here.

The total message length is m, and we will now split the message blocks up into $t+1$ blocks which we will need to control for the attack, and $m-t-1$ blocks for the actual message content. As a first step, we will ensure that T_1 is correct, by adding a message block $M_1[m-t] = T_1 \oplus \sum_{i=1}^{m-t-1} M_1[i]$.

As long as $\sum_{i=m-t+1}^{m} M_1[i] = 0$ we get the correct tag T in the end for M_1. To get the correct T_2 we can do the following:

- Generate two sets of messages A_1, A_2 of length t, where $A_1[i] = 0$ and $A_2[i] = 1$. We require here that t is even, in order to have a checksum of 0.
- We encrypt those messages with K_1, decrypt them with K_2 and add them pairwise to obtain the values

$$\begin{aligned} &\gamma_j[i+1] = \\ &E^{-1}_{K_2,Z[m-t+2i+1]}(E_{K_1,Z[m-t+2i+1]}(A_j[2i+1])) + \\ &E^{-1}_{K_2,Z[m-t+2(i+1)]}(E_{K_1,Z[m-t+2(i+1)]}(A_j[2(i+1)])), \\ &\forall i, j : 0 \le i < t/2, j \in \{1,2\}. \end{aligned} \tag{10}$$

- The next step is to find values $x[i] \in \mathbb{F}_2$, such that

$$\gamma_{x[1]} + \ldots + \gamma_{x[t/2]} = T_2. \tag{11}$$

- This gives us the following equation

$$\sum_{i=1}^{t/2} \gamma_1 x[i] + \gamma_2 (1 - x[i]) = T_2. \tag{12}$$

- We introduce a new variable $\bar{x}$ and denote $\gamma[j]$ as the jth bit of γ. This gives us the following system of linear equations over $\mathbb{F}_2$

$$\begin{aligned} \sum_{i=1}^{t/2} \gamma_1[j] x[i] + \gamma_2[j](\bar{x}[i]) = T_2 \quad 1 \le j \le b \\ x[i] + \bar{x}[i] = 1 \quad 1 \le i \le t/2. \end{aligned} \tag{13}$$

  Here, b is the blocksize of E. This gives us $t/2 + b$ equations in $2b$ unknowns, therefore if we set $t/2 > b+1$ we get a solution with a probability > 0.5.

- Finally, we set

$$M_1[m-t+i] = \begin{cases} 0, \text{ if } x[\lfloor i/2 \rfloor] = 1 \\ 1, \text{ if } \bar{x}[\lfloor i/2 \rfloor] = 1 \end{cases} \qquad 1 \le i \le t, \tag{14}$$

  and compute the corresponding values for M_2, which leads to both M_1, M_2 giving us the correct tag T.

3.4 Restrictions on the plaintext

Although we can demonstrate how to generate ciphertext which can decrypt to valid plaintexts under different keys

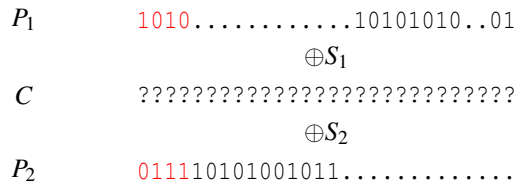

Figure 1: Example of constructing two plaintexts P_1, P_2 from the generated key streams S_1, S_2 and the conditions on the single bits. The keystreams are fixed and the adversary can choose the ciphertext C in order to determine the plaintexts. A **?** denotes that this bit can be freely chosen by the adversary, a **.** that this can be any value in the plaintext and **1**, **0** that the corresponding bit should have this value. In this example the conditions on the first 4 bits (red), would have to be fulfilled by finding the two key stream S_1, S_2, while all the other conditions can simply be solved by choosing the corresponding bits in C.

in a cryptographic sense, a natural question is whether it is possible to construct meaningful plaintexts. In the settings discussed in Section 2 the adversary wants to find a single ciphertext C and two keys K_1 and K_2 such that $\mathsf{Dec}(K_1, C) = M_1$ and $\mathsf{Dec}(K_2, C) = M_2$ are *meaningful* messages.

The condition here is that $C = P_1 \oplus S_1 = P_2 \oplus S_2$, where S_1, S_2 are known to the adversary. This means that if the adversary wants to fix a single bit in P_1, then this will fully determine the corresponding bit in C resp. P_2. Therefore, if we want to control the content of a bit at a position in either P_1 or P_2, we can just set it to the corresponding value. However, if we want to control the same bit position in both P_1 and P_2, then we will have to find a collision in the key stream of S_1 and S_2 at this position. In the case of OCB this has to be done on a block size level.

The attacks outlined in Section 3.3 give us additional other restrictions. In the case of AES-GCM and ChaCha20Poly1305 we need a single block fixed in both P_1 and P_2 to *repair* the tag, while for AES-GCM-SIV this will typically require 2 controlled blocks. To summarize these are the restrictions:

1. If we have to set the value of the ith bit in both plaintexts, then we have to brute-force the key stream to get the correct bit in both plaintexts. In the case of OCB we have to do this on block-size level, therefore this will typically be infeasible and we have to ensure there are no overlapping conditions.

2. We need to be able to put random blocks of in both plaintexts at the same position. For AES-GCM we only need a single block, for AES-GCM-SIV typically 2 blocks and for OCB on average $b+1$ blocks where b is the blocksize of the block cipher used.

4 Creating Meaningful Plaintexts

In practice, we need to be able to generate plaintext which are interpreted as valid files. To that end, we seek to craft a file whose contents are valid according to two different file formats — a binary polyglot — and where the two file formats' data do not overlap.

As many file formats start at offset zero with their magic signatures and headers, making such pair of plaintexts coexist at the same offset is not possible, and brute-forcing both sets of magic and headers is not practical.

We describe how we can exploit certain file formats characteristics. There are many file formats with their own requirements and restrictions, but we found more than 250 working combinations of formats. We refer to [fil] for a description of the file formats referenced below.

4.1 File Format Characteristics

In this section, we introduce the aspects of file formats that are important in generating binary polyglots.

Enforced offset Most formats require files to start at offset zero, but some formats allow files to start at any position. Pure compressors — software which compress one block of data with no notion of file such as Bzip2, Gzip or XZ — typically start at offset zero, as does storage software such as TAR or Unix Archive. Typically, archive formats such as ZIP, RAR, 7z or Arj and flexible web-oriented formats such as Html and PHP allow files to start at any offset.

Pre-cavity Some formats start with a cavity that can take any content. This could be by design, as with the raw dump of sectors of an ISO image, or by courtesy, as for DICOM or PDF, or by abuse, such as archiving a null-named file with TAR.

Appended data Most file formats have one or more ways of determining whether a file is complete:

- The file size or the number of elements is declared in advance, such as RIFF or Java Class.

- Enough elements have been parsed in the file to declare it valid, and the parser gives up when it meets incorrect data afterward. Anything after that is ignored.

- The format has some kind of terminator or footer to officially declare that the format is valid. Sometimes the terminator is just an element declaring itself as the last one with a specific bit set or its pointer to a next element is set to null. Some formats like XZ actively check that the file ends up with its footer, but in practice, most parsers just process the file until a terminator is encountered and all subsequent data is ignored.

Parasite Most file formats allow for *parasitic data* that is left as-is and not parsed. We explore here the typical strategies depending on the kind of format:

- Archive formats are like a stack of labelled storage boxes (cf. Figure 2). To add parasitic data to such a file, just store it, i.e. keep as-is without any compression (see Figure 3). Optionally prevent the extra file to show like the other ones in the archive listing, by for example corrupting a checksum or giving a null file name. Note that some archive formats like XZ always process the data with some light compression, and therefore modify data even at their lowest compression level, but often they implement storage without any form of processing.

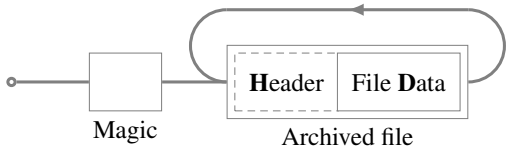

Figure 2: Layout of an archive format (like AR).

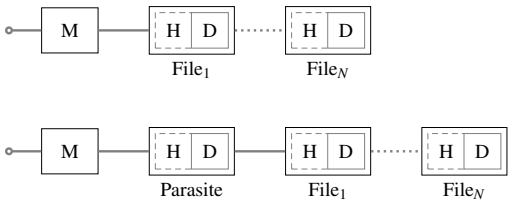

Figure 3: Adding parasitic data to an archive format (like Figure 2).

- Sequence-based formats are like trains: one locomotive for the header, and one or several wagons for the chunks (cf. Figure 4). To add anything in that train, just load your goods on another wagon, and insert it in the train at any wagon boundary (see Figure 5). For such formats, use a comment/junk block. While a comment is typically expected to be text, short and unique, such chunks can in practice contain any content, with length which only limitation is how it's stored, and be repeated: parsers just treat the comment as data to ignore. If the format does not have such a kind of element, it is still possible to rely on redundant or unused element, such as an extra ILDA palette, a picture in an RTF, or just a block of data in PDF. These chunks typically declare before their data their type and size — pre-wrapping — and occasionally store some extra information — post-wrapping — after their data such as CRC, size (redundant for error detection), chunk terminator (see Figure 6).

- Some formats such as ICO are like books. They have at the start a table of content that points to each chapter to read. If you just add more pages, just update the pointers in the table of content. Some formats such as TIFF or BMP are like towed dinghies, where a tugboat just has a rope - a pointer - to the next boat, and each boat is linked to the next by another rope. If you want to carry more, just put something between two boats, and make the rope longer. In practice, you can for example make some space that will be ignored by moving format data further and adjusting all pointers accordingly (see Figure 7).

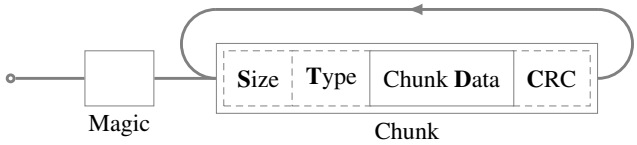

Figure 4: Layout of a sequence format (like PNG).

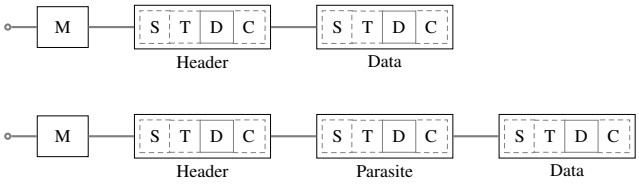

Figure 5: Adding parasitic data to a sequence format (like Figure 4).

Stopping parsers The parasite payload might be executed but some trailing bytes might still be executed. It might be better or even required to break out of the hosting format (JavaScript) or to terminate parsing forcibly with some specific keyword in Ruby `__END__` or PostScript `stop` or some tricks such as forcing recursion and exhausting the parser.

Wrappending Note that some formats do not tolerate appended data, notably those that are parsed as specific structures until the end of file, but it is still possible to add a trailing structure wrapping a parasite, as most format structures are declared before the data they contain. Such appended data wrapped in a structure can be called wrappended.

Wrappending can be required if a format that doesn't tolerate appended-data is used as a parasite into another one, such as DICOM/PNG polyglots: PNG starts at offset zero, DICOM at 128, so DICOM is a parasite of PNG, yet DICOM doesn't tolerate appended data, so the body of the PNG can't be just following the DICOM parasite (cf. Table 4).

4.2 Combination strategies

Knowing the typical characteristics of a file formats, the following strategies make it easy to have a file with more than one valid format:

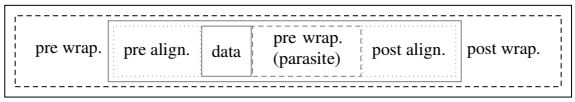

Figure 6: Generic layout of a parasitic chunk.

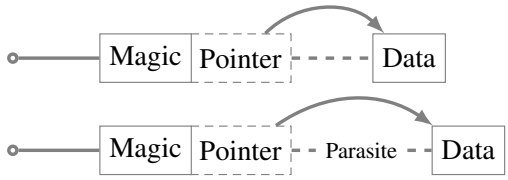

Figure 7: Adding parasitic data to a pointer-based format.

- Combining a format C that starts with a cavity and another format A that tolerates appended data or wrappending. The only extra requirement is that the cavity should fit the other payload.
- Appending a format that is valid at any offset to a format tolerating appended data, or wrappending it.
- Inserting a format P valid at any offset as a parasite inside another format H. A chunk of H must be able to fit all of P. Otherwise, it may still be possible to split P in several pieces, making it a zipper: the pieces of each format are parasites to the other.

Zippers Some formats like GIF start at offset zero and only tolerate parasites of very limited length, as its comment's length is encoded on a single byte, limited to 255. So it's likely not possible to make a parasite to contain the complete other payload. A workaround for that is to split the parasite payload in headers and parasite declaration, so that the body of the host itself is a parasite to the hosted file.

Therefore, both payload's bodies are parasite to each other. They both set up the structure to tolerate the other's body, exactly like the teeth of each side of a zipper embrace the other side's teeth.

Results These combined strategies make that the vast majorities of formats can be combined in one way or another within the same file.

It can be noted that Java classes are executable and can be malicious, and are packaged in JAR files which are just ZIP, which can be appended to any other format and still be valid, which was used again in August 2020 — CVE-2020–1464 — to exploit people in the wild by piggy-packed to a signed MSI file, making the whole file malicious yet apparently validated by Microsoft, so this kind of abuse is still potentially dangerous nowadays.

4.3 Binary polyglots

We see that it's usually possible for two different formats to coexist in the same file without any overlap, therefore we can fulfill the restriction 1) stated in Section 3.4 without additional costs. In practice, very few formats — two to our knowledge: ID3v1 and XZ — can't be made to coexist with any other format: They enforce parsing at offset zero. They actively enforce a footer, preventing appended data. They are very restrictive, preventing any form of parasite.

These binary polyglot files are instant to make, even generically: some data has to be moved around, and some pointers and checksums updated. To generate binary polyglots given 2 input files with no required pre-processing, we wrote a tool which identifies supported file formats, then tries different layouts and generates the final binary polyglot file.

4.3.1 Generic polyglot protocol

There are several layouts of generic protocol: pre-cavity based, appended-data based, and parasite/zipper based. From a generic perspective, a working binary polyglot is a protocol exchange between 2 formats - here with payload P_1 and P_2 of format F_1 and F_2.

Pre-cavity based: F_1 always has or can optionally have a pre-cavity C. F_2 tolerates appended data or wrappended data. $\text{Size}(P_2) \leq \text{Size}(C)$.

Appended-data based: F_1 tolerates appended data or wrappended data. $\text{Size}(F_1) \leq \text{StartOffset}(P_2)$.

Parasite based: P_1 tolerates a parasite Pa. Pa tolerates appended or wrappended data. $\text{StartOffset}(Pa) \leq \text{StartOffset}(P_2)$. $\text{Size}(P_2) + \text{wrapping} \leq \text{MaxSize}(Pa)$.

4.3.2 Beyond standard generic polyglots

Other possible techniques (cf. [Alb15, SBK+17, LP20, AAE+14, Alba]) are not generic to all file formats.

We could easily take the next PDF article that you would want to open, combine it with malware — even without the source — and turn it into a standard PDF that once encrypted with the right key and then decrypted with another given key (both known in advance), will result in the original malware.

It is easy to turn such a binary polyglot file into two valid plaintexts that will be combined as the same ciphertext, using the offsets where the file contents change side, and each side doesn't depend on the contents of the other one (except checksums of parasite chunks). It is like slicing two sausages at the same locations and mixing their contents.

Note that our attack can make it possible for two different contents to overlap on a few bytes.

4.4 Crypto-polyglots

It is possible to craft files with additional cryptographic properties that will alter their content depending on some specific state. For example, the two files of a hash collision pair, or a file changing its type via encryption (see [AA14]). These files are not necessarily binary polyglots (they can be for example of the same type, but with different contents), but we call polyglot files with such properties crypto-polyglots.

Since our attack can make some bytes of each content overlap and depending on the decryption key we used, we can craft some crypto-polyglots with the following benefits:

- Two formats starting at offset zero to coexist. It's particularly powerful when one of the format starts with a very few bytes, such as Windows' PE, which only needs 2 bytes 'MZ' at offset zero, then 58 unchecked bytes.

- Two files with the same format but with different data, with each plaintext's data hosting the other's: basically a crypto-zipper. This is possible for example with 2 JPG plaintexts, as it has a short magic and a short comment segment declaration, which can host foreign data.

- Hiding one plaintext from the other. This is particularly useful to bypass blacklisting: as a security measure, Adobe Reader checks if a PDF file starts with another common format signature and will refuse to open it. In our case, a PDF/PNG polyglot will be opened as the PNG payload will be wrongly decrypted when the PDF payload is in clear. This has also the interesting aspect of hiding your malicious code from scanning and preventing exploit code to be leaked in advance, unlike standard polyglots and reusable hash collisions.

As most formats often require many bytes of data to declare their type and headers before they can host a parasite (for example, parasitic data in a GIF image can only be present at offset 16 or more), they are non-practical for overlapping contents.

4.4.1 Tag correction

In the case of AES-GCM, one block needs to be used to correct the authentication tag (see Section 3.4), respectively more blocks are required for AES-GCM-SIV and OCB. In practice, most formats support appended data, so just appending that block is enough. For the few formats that don't tolerate appended data, wrappending, increasing the size of the internal parasite, or using a small space of a cavity are valid tricks. They all depend on the formats combination used in the file.

5 Adding Key Commitment to AE

5.1 Hash Function Use in Prior Work

Recall that key committing AE schemes $(\mathsf{Enc}, \mathsf{Dec})$ need to be collision resistant, that is, it should be difficult to find two inputs $X = (K,N,A,M)$ and $X' = (K',N',A',M')$ such that $\mathsf{Enc}(X) = \mathsf{Enc}(X')$. As we discuss below, all prior work relies on schemes which explicitly or implicitly contain hash functions to achieve collision resistance.

Farshim et al. [FOR17], the first to study key commitment which they call "robustness", propose generic composition — like encrypt-then-MAC [BN08] — which send either the entire message or ciphertext into a collision-resistant pseudorandom function (PRF). As a practical instantiation, they propose using a hash function with a key, for example HMAC [BCK96] or KMAC [NIS16].

Grubbs et al. [GLR17a] design *compactly committing* AE, where a small portion of the ciphertext commits to the message. Due to differences in security definitions, GLR's compactly committing AE does not formally guarantee FOR's robustness, yet GLR need collision resistance and, like FOR, they propose using collision resistant PRFs which process the entire message or ciphertext.

Dodis et al. [DGRW18] design *encryptment* schemes, which they propose as a building block to achieve robust or compactly committing AE. Their schemes are more efficient than FOR and GLR's, yet they still need to process the message through a hash function, and even prove that block cipher-based encryptment schemes cannot be more efficient than hash functions. They also conjecture that block cipher-based key robust schemes as defined by FOR cannot be more efficient than hash functions.

There are two drawbacks to these approaches:

1. Since commonly used AE algorithms like AES-GCM and ChaCha20Poly1305 do not follow the above hash-based designs, avoiding attacks requires using less widely deployed algorithms, or entirely new ones.

2. The performance of hash-based designs is limited by the fact that commonly used hash functions are serial, whereas widely used AE schemes are parallelizable. This becomes an issue when the message or ciphertext is large, and in fact led Facebook Messenger to rely on AES-GCM to encrypt message attachments, exposing the application to attack [DGRW18].

Ideally, a solution would require minimal changes to widely deployed, highly efficient AE schemes like AES-GCM.

5.2 Overview of Our Solutions

We observe that the message or ciphertext does not need to be processed as in a hash-based design: if the ciphertext contains a commitment to just the key, verified during decryption, then

the adversary cannot generate ciphertext valid under two keys. We propose the following:

Padding Fix. Let X denote an ℓ-bit string of 0's. Prepend X to the message M for each encryption, $\mathsf{Enc}(K,N,A,X \| M)$, and check for the presence of X at the start of the message after decryption; decryption fails if X is not present. This solution is not generic, and must be analyzed per scheme. Furthermore, tt is implicitly assumed that $X \| M$ is a legitimate input to Enc, i.e., that it is still shorter than the longest legitimate message.

Generic solution. Given a key K, derive an encryption key and a commitment using collision-resistant PRFs: $K_{\text{enc}} = F_{\text{enc}}(K)$ and $K_{\text{com}} = F_{\text{com}}(K)$. The ciphertext is a combination of the normal ciphertext computed with K_{enc} and Enc, and K_{com}: $(\mathsf{Enc}(K_{\text{enc}},N,A,M),K_{\text{com}})$. A nonce N' can be used to compute $K_{\text{enc}} = F_{\text{enc}}(K,N')$ or $K_{\text{com}} = F_{\text{com}}(K,N')$. The presence or absence of N' to derive K_{enc} and K_{com} results in four constructions named in Table 1.

5.3 Padding Fix

Our padding solution $\mathsf{Pad}_\ell\Pi = (\mathsf{Pad}_\ell\Pi^{\mathsf{Enc}}, \mathsf{Pad}_\ell\Pi^{\mathsf{Dec}})$ for some predetermined integer $\ell > 0$ and AE scheme $\Pi = (\mathsf{Enc},\mathsf{Dec})$ is algorithmically described in Appendix E. We discuss the security of $\mathsf{Pad}_\ell\Pi$ when Π is instantiated with 96-bit-nonce AES-GCM and ChaCha20Poly1305, followed by performance considerations.

AE Security. Since the padding fix uses the underlying AE scheme in a black-box manner, conventional AE security (defined in Appendix B) follows immediately. Note that the AE security bounds change since the plaintext length increases by ℓ bits. However, for all practical values of ℓ, e.g. one or two block lengths, the difference is negligible.

Key Commitment Security. Both of our solutions, the padding fix and generic solution, send the encryption key K through collision-resistant functions, and include the output of a collision-resistant function in the ciphertext. However, the padding fix relies on the internals of the AE scheme for collision resistance, which is why security requires a case-by-case analysis. We analyze 96-bit-nonce AES-GCM and ChaCha20Poly1305, and leave analysis of other schemes for future work.

AES-GCM. We use the notation and definitions of Appendix A and Section C.1. Let $\mathsf{E} : \mathsf{K} \times \mathsf{X} \to \mathsf{X}$ denote AES with any key size. Let (N,A,M,C) be GCM input. GCM uses the nonce N to produce an initial value $I = N \| \mathsf{str}_{32}(1)$, which in turn is used to generate AES input in CTR mode. When prepending a string of ℓ zero's to M, GCM's ciphertext includes CTR mode output of length ℓ. Using the notation of Section C.1, we let $F := \mathsf{E}_K(\mathsf{msb}_{96}(I) \| \cdot)$ denote an evaluation of E_K with the first 96 bits fixed to $\mathsf{msb}_{96}(I)$, and define L to be the smallest multiple of 128 not less than ℓ, then the following is added to GCM's ciphertext:

$$\mathsf{msb}_\ell\Big(\mathsf{CTR}[F](\mathsf{lsb}_{32}(I),L)\Big). \tag{15}$$

In particular, a key commitment adversary would have to find two different keys such that the first ℓ bits of CTR mode under initial value I are the same under those two different keys, otherwise the GCM ciphertext would not be the same and the adversary would not have broken the key commitment. Letting $I_1,\ldots,I_L$ denote the AES inputs used in the above CTR mode call, the adversary needs to find K_1 and K_2 such that

$$\begin{aligned} &\mathsf{E}_{K_1}(I_i) = \mathsf{E}_{K_2}(I_i) \quad \text{for } i = 1,\ldots,L-1 \quad \text{and} \\ &\mathsf{msb}_{\ell-128(L-1)}(\mathsf{E}_{K_1}(I_L)) = \mathsf{msb}_{\ell-128(L-1)}(\mathsf{E}_{K_2}(I_L)). \end{aligned} \tag{16}$$

If we model AES as an *ideal cipher*, that is, as a random variable chosen uniformly at random from the set of all block ciphers with interface $\mathsf{K} \times \mathsf{X} \to \mathsf{X}$, then the probability that an adversary finds K_1 and K_2 such that Equation 16 holds for any given $I_1,I_2,\ldots,I_L$ is at most $p^2/2^\ell$, where p is the number of queries the adversary makes to the ideal cipher. This leads to the following result.

Theorem 1. *Let Π denote GCM with 96-bit nonces using ideal cipher $\pi : \mathsf{K} \times \mathsf{X} \to \mathsf{X}$. Consider an adversary $\mathbf{A}$ with access to π. Then $\mathbf{A}$'s q-KC advantage against $\mathsf{Pad}_\ell\Pi$ is at most $(q+p)^2/2^\ell$, where $\mathbf{A}$ makes at most p queries to π.*

ChaCha20Poly1305. Analysis of ChaCha20Poly1305 is similar to AES-GCM since ChaCha20Poly1305 uses CTR mode (see Section C.2), but with the ChaCha20 block function instead of AES. If we idealize the ChaCha20 block function CC as a random variable over all functions $\{0,1\}^{256} \times \{0,1\}^{32} \times \{0,1\}^{96} \to \{0,1\}^{512}$, then we achieve the same result as with AES-GCM:

Theorem 2. *Let Π denote ChaCha20Poly1305 using ideal random function $\rho : \{0,1\}^{256} \times \{0,1\}^{32} \times \{0,1\}^{96} \to \{0,1\}^{512}$. Consider an adversary $\mathbf{A}$ with access to ρ. Then $\mathbf{A}$'s q-KC advantage against $\mathsf{Pad}_\ell\Pi$ is at most $(q+p)^2/2^\ell$, where $\mathbf{A}$ makes at most p queries to ρ.*

Assumptions on the Underlying Primitives. GLR [GLR17b] and DGRW [DGRW19] justify security assuming either key-dependent message security, related-key security, or by modelling the primitives as ideal. Similarly, our analysis assumes the primitives are ideal.

To build a conventional AE scheme (cf. Appendix B) with a block cipher or hash function, it suffices to assume that the underlying primitive behaves like a PRP or PRF when

keyed with a uniformly random key unknown to the adversary. In contrast, supporting AE key commitment requires understanding what happens when the adversary can choose the key used in the block cipher or hash function. As a result, practical instantiations require a stronger assumption on the underlying primitives. Since the adversary can choose the key, related-key attacks [Bih94] and known-key [KR07] or chosen-key attacks become relevant.

In fact AE schemes might not achieve key commitment when instantiated with *weak* primitives. Take for example HMAC, which is commonly used to build AE with e.g. CTR-mode. HMAC does not require a collision resistant hash function, therefore the use of HMAC-SHA-1 could be justified, and it is still used in TLS in practice. However, if an adversary can find a collision efficiently for the hash function it is possible to find two different tags under two different keys to break the key commitment. As chosen-prefix collisions are practical for SHA-1 [LP20], HMAC-SHA1 is insufficient to provide key commitment while this is not the case for HMAC used with a collision resistant hash function.

In particular, the padding fix with AES-GCM assumes an ideal cipher, and therefore raises the following interesting problem: *Is it possible to find two keys k_1, k_2 such that $AES_{k_1}(0) = AES_{k_2}(0)$ in less than $\approx 2^{64}$ trials*. If the key-size is larger than the blocksize, then such a pair of keys must exist. While there has been some work on the chosen-key setting [FJP13] or using AES in a hashing mode [Sas11], we are not aware of any results on this specific problem.

Performance. The performance overhead of the Padding solution is minimal. To estimate this performance overhead, let $T_{GCM}(a,p)$ denote the performance (e.g., in processor cycles, where smaller is better) for AES-GCM encryption with a 128-bit key, over an input with AAD A and message M. For convenience, assume that A and M consist of full (128-bit) blocks, and set $|A| = 128a$ and $|M| = 128p$ for some $a \geq 0$, $p \geq 0$. With this notation, the performance of the Padding solution is $T_{\mathsf{Pad}_\ell\ GCM}(a,p) = T_{GCM}(a,p+1)$. The actual differences (and relative difference) depends on multiple factors, including the computing platform, and potentially also the values of a and p. For example, well aligned buffers may fit better in the caches, and can be accessed more efficiently. To illustrate, we consider $a = 0$ (no AAD) and measurement carried out on OpenSSL (version 1.0.2m). This code is highly optimized to leverage the potential pipelining that the processor can offer. We ran the code on a 7th Generation Intel Core i7-7700 processor ("Kaby Lake"). On this processor, the latency of the `AESENC` instruction is 4 cycles. This means that the latency for AES encryption of one block is ~ 41 cycles (the throughput is 10 cycles). For $p = 128$ (a 2048 bytes message), we measured $T'_{\mathsf{Pad}_\ell\ GCM}(0,128) = 1,739$ cycles and $T_{GCM}(0,128) = 1,665$ cycles, indicating an overhead of 74 cycles for the Padding solution and relative impact of $\sim 4.4\%$. With $p = 127$ (a 2032 bytes message), we measured $T'_{\mathsf{Pad}_\ell\ GCM}(0,128) = 1,665$ cycles and $T_{GCM}(0,128) = 1,636$ cycles. In this case, The overhead is 29 cycles, and the relative impact is $\sim 1.8\%$. For a longer message we measured $T'_{\mathsf{Pad}_\ell\ GCM}(0,384) = 4,263$ cycles and $T_{GCM}(0,384) = 4,203$ cycles, with relative impact of $\sim 1.4\%$.

Algorithm 1: $\mathsf{CommitKey}_{IV}\Pi^{\mathsf{Enc}}(K,N',N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N' \in \{0,1\}^{\nu'}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\text{com}} \in \{0,1\}^c$

$K_{\text{enc}} \leftarrow F_{\text{enc}}(K,N')$
$K_{\text{com}} \leftarrow F_{\text{com}}(K,N')$
$C \leftarrow \mathsf{Enc}(K_{\text{enc}},N,A,M)$
return (C,K_{com})

5.4 Generic Solution

Table 1: The four types of key derivation for the generic solution. Each key is either derived with a nonce, or without.

		K_{com} fixed	K_{com} nonce
K_{enc} fixed		Type I	Type III
K_{enc} nonce		Type II	Type IV

Let $\Pi = (\mathsf{Enc},\mathsf{Dec})$ be an AE scheme where $\mathsf{K} = \{0,1\}^\kappa$ and $\mathsf{N} = \{0,1\}^\nu$. We describe the scheme $\mathsf{CommitKey}\Pi$ over Π. Let κ_0, ν', c be positive integers where, without loss of generality, $\kappa_0 \geq \max(\kappa,c)$. Let

$$F_{\text{enc}} : \{0,1\}^{\kappa_0} \times \{0,1\}^{\leq \nu'} \to \{0,1\}^\kappa \tag{17}$$

$$F_{\text{com}} : \{0,1\}^{\kappa_0} \times \{0,1\}^{\leq \nu'} \to \{0,1\}^c \tag{18}$$

be independent PRFs. Both schemes use the same key $K \in \{0,1\}^{\kappa_0}$, called the *main key*, but must guarantee that their outputs remain independent.

$\mathsf{CommitKey}\Pi$ has four types, depending on whether a nonce is used in F_{enc} or F_{com} (see Table 1). We describe Type IV in Algorithm 1 and Algorithm 2. The remaining types are described in Appendix D.

Table 2: The overheads compared to Π involved with the different flavors of $\mathsf{CommitKey}\Pi$, when encrypting or decrypting q payloads with the main key K.

Type	F_{enc} calls	F_{com} calls	Communication
I	1	1	c
II	q	1	$c+\nu'$
III	1	q	$c+\nu'$
IV	q	q	$c+\nu'$

Algorithm 2: $\mathsf{CommitKey}_{IV}\Pi^{\mathsf{Dec}}(K,N',N,A,C,K_{\text{com}})$

Input: $K \in \{0,1\}^{\kappa_0}$, $N' \in \{0,1\}^{v'}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $C \in \mathsf{C}$, $K_{\text{com}} \in \{0,1\}^{c}$

Output: $M \in \mathsf{M} \cup \{\perp\}$

$K'_{\text{com}} \leftarrow F_{\text{com}}(K,N')$

$K'_{\text{enc}} \leftarrow F_{\text{enc}}(K,N')$

$M \leftarrow \mathsf{Dec}(K'_{\text{enc}},N,A,C)$

if $K_{com} \neq K'_{com}$ *or* $M \stackrel{?}{=} \perp$ **then return** $\perp$

return M

Note that $\mathsf{CommitKey}\Pi$ includes a nonce N_1 in addition to the nonce N used for the underlying AE scheme Π. This is done for backwards compatibility, as Π might already be deployed and re-using Π's nonce for $\mathsf{CommitKey}\Pi$ might not be feasible. The security requirements for N_1 and N are the same, so if possible, they can be set to equal each other as long as uniqueness is guaranteed; however care must be taken to ensure the nonces are sufficiently long — $|N|$ and $|N'|$ may not be the same, and depending upon the exact requirements of the application (e.g. N' needs to be generated randomly), one might want a larger $|N'|$.

Using the Different $\mathsf{CommitKey}\Pi$ Types. The different $\mathsf{CommitKey}\Pi$ types have different incremental computational and bandwidth overheads over Π; see Table 2. $\mathsf{CommitKey}\Pi$ Type I and type II carry the lowest incremental overheads over Π as they use a fixed key identifier K_{com}. These are useful when leaking an identifier for the key used to produce ciphertext does not violate privacy requirements, for example, when a main key is used for only one session between the communicating parties. Deriving a nonce-dependent K_{com} value, as in Types III and IV, does not leak any key identifiers, but comes at some incremental cost.

Simple Instantiation of F_{enc} and F_{com} Let $\kappa_0 = \kappa = 256$, assume that $v_1 \leq 256$, and set $c = 256$. Let L_{enc} and L_{com} be fixed labels and define

$$F_{\text{enc}}(K,N) = \texttt{SHA256}(K \,\|\, L_{\text{enc}} \,\|\, N) \tag{19}$$

$$F_{\text{com}}(K,N) = \texttt{SHA256}(K \,\|\, L_{\text{com}} \,\|\, N) \tag{20}$$

For concreteness, we give examples of labels L_{enc} and L_{com} in Table 3. The different $\mathsf{CommitKey}\Pi$ types are encoded in the labels L_{enc}, L_{com}. With this choice,

$$\big|K \,\|\, L_{\text{enc}} \,\|\, N\big| = \big|K \,\|\, L_{\text{com}} \,\|\, N\big| \leq 576 \text{ bits}, \tag{21}$$

so deriving K_{enc} and K_{com} require for each computation at most two calls to the SHA256 compression function. Furthermore, for Type I, computing K_{enc} and K_{com} invokes the SHA256 compression function only once, and for Type II, computing K_{com} calls the compression function only once (but twice to compute K_{enc}). Appendix F demonstrates how to instantiate a Type I key committing AES-GCM.

Table 3: Sample labels for use in our instantiation of F_{enc} and F_{com}. Define some fixed label $L0$ of length 48 bits; for example $L0 =$ `0x436f6d6d6974`, which is `Commit` in hexadecimal notation.

Type	L_{enc}	L_{com}
I	$L0 \,\|\, 0x01 \,\|\, 0x01$	$L0 \,\|\, 0x01 \,\|\, 0x02$
II	$L0 \,\|\, 0x02 \,\|\, 0x01$	$L0 \,\|\, 0x02 \,\|\, 0x02$
III	$L0 \,\|\, 0x03 \,\|\, 0x01$	$L0 \,\|\, 0x03 \,\|\, 0x02$
IV	$L0 \,\|\, 0x04 \,\|\, 0x01$	$L0 \,\|\, 0x04 \,\|\, 0x02$

Key Commitment Security. To meet the $\mathsf{CommitKey}\Pi$ design goal, the PRFs F_{enc} and F_{com} must be collision-resistant (22); our instantiation achieves collision-resistance with SHA256. Furthermore, c should be large enough to make brute-force collision search impractical.

Claim 1. *If adversary* **A** *produces a winning tuple* (N,A,C,T,K_{com}) *for keys* $K_1 \neq K_2$*, then* **A** *has found a collision (on* K_{com}*), i.e.,*

$$F_{com}(K_1,N) = F_{com}(K_2,N)\,. \tag{22}$$

Note that the claim holds even if the adversary may freely choose two different nonces N_1 and N_2 as input.

AE Security. Say the PRF's used by $\mathsf{CommitKey}\Pi$ are secure, that is, each PRF output looks uniformly random and independent of other PRF output against computationally bounded adversaries, then:

1. Π is called using K_{enc}, which is uniformly random and independent, hence if Π is a secure AE scheme, then Π's output maintains confidentiality and integrity, and
2. K_{com} is uniformly random and independent of Π's output, hence K_{com} does not affect AE security.

As with other generic compositions involving key derivation functions, we can use a straightforward hybrid argument with the result that $\mathsf{CommitKey}\Pi$ preserves Π's AE security.

We state AE security for $\mathsf{CommitKey}\Pi$ Type IV. The statements and analysis for Types I, II, III are analogous. See Appendix B for definitions of PRF and AE advantage. Note that the PRF assumption in the theorem assumes that F_{com} and F_{enc} behave independently even though they accept the same main key as input — in practice, both PRFs are instantiated with the same function F and using domain separation.

Theorem 3 ($\mathsf{CommitKey}_{IV}\Pi$ AE Security)**.** *Let* **A** *be a nonce-respecting AE adversary against* $\mathsf{CommitKey}_{IV}\Pi$ *making at most q queries with associated data, message, and ciphertext length at most* ℓ*. Let* **B** *be a PRF adversary and* **C**

an AE adversary against Π*, then* $\mathbf{A}$*'s multi-key AE advantage with* μ *instances is*

$$\mu\text{-}\mathsf{AE}_{\mathsf{CommitKey}_{IV}\Pi}(\mathbf{A}) \leq \mathsf{PRF}_{F_{com},F_{enc}}(\mathbf{B}) + (\mu \cdot q)\text{-}\mathsf{AE}_{\Pi}(\mathbf{C}), \tag{23}$$

where $\mathbf{B}$ *makes at most* q *queries to each of its oracles, and* $\mathbf{C}$ *makes at most 1 query to each of its oracles with associated data, message, and ciphertext length at most* ℓ*.*

Appendix G shows how to use the bounds of Theorem 3.

Design rationale and alternatives. We require $\mathsf{CommitKey}\Pi$ to use $\kappa_0 \geq \kappa$ to keep a key hierarchy: the derived encryption keys (K_{enc}) are not longer than the main key. Similarly, we require $\kappa_0 \geq |K_{\text{com}}|$ and set K_{com} to be sufficiently long 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 $\mathsf{CommitKey}\Pi$ encryption.

We point out that defining $F(K,L) = \mathtt{H}(K \parallel L)$ with any NIST standard collision-resistant hash function $\mathtt{H}$, with a sufficiently long digest, is an acceptable choice. This makes it is easy to choose a main key (K) of a desired length, and also to truncate the digests to c or κ bits, as needed. Note that it is implicitly assumed here that for this usage, $\mathtt{H}$ is invoked with equal-length arguments.

6 Related Work

Security, performance, and implementation demands from practice have pushed the design and analysis of AE schemes since its formalization by Bellare and Namprempre [BN00]. The inefficiency of generically composing an encryption scheme with a message authentication code algorithm led to the design of highly efficient, parallelizable schemes such as OCB [RBB03]. OCB's patents led NIST to standardize alternative AE schemes like GCM [MV04a], which in turn saw widespread adoption. ChaCha20Poly1305 [NL15] is popular on systems without AES hardware acceleration.

Hoang, Krovetz, and Rogaway introduce the concept of "robust AE" (RAE) [HKR15], formalizing one of the strongest types of security that an AE scheme can satisfy. We do not use the term robust in the sense of "robust AE."

Abdalla et al. [ABN10] initiate a provable-security treatment of robust encryption. Farshim et al. [FOR17] extend the concept of robustness to symmetric-key encryption. Canetti et al. [CKVW10] consider "wrong-key detection", which is similar to robustness.

The OPAQUE protocol [JKX18] requires an AE scheme with random key robustness: robustness where the attacker may not choose the two keys under which it finds a collision. An early draft of an OPAQUE protocol RFC describes a way to fix GCM similar to what we propose [Kra, Section 3.1.1], by appending a constant string to the plaintext. Subsequent drafts of the RFC remove mention of the fix.

Everspaugh et al. [EPRS17] discuss how to securely support key rotation without decryption in key management services via updatable AE; as part of their motivation, they discuss how Amazon and Google perform key rotation. To achieve ciphertext integrity while rekeying, they require the underlying symmetric encryption scheme to be *compactly robust*, where the adversary should not be able to find two keys and two ciphertexts with the same tag.

7 Conclusions

Beyond Facebook Messenger, Section 2 demonstrates products and settings where key commitment naturally arises and a lack thereof violates expectations, resulting in attacks. We conclude that key commitment is an important property to consider for AE schemes.

From our study of attacks, we see that collision-resistance, or rather, a lack thereof is important to being able to take advantage of AE schemes' lack of key commitment; the fastest AE schemes are often not collision resistant. We conclude that taking advantage of a lack of key commitment is easy to do for widely deployed AE schemes. Adversaries can choose encryption keys as they please, and the automated tools we have developed demonstrate how easy it is to generate binary polyglots with a wide variety of file formats.

However, we also conclude that it is easy to add key commitment to AE schemes. While prior work focused on hash-function based designs, we show that it is secure to add key commitment through a blackbox use of the underlying AE scheme. We note that, while the generic solution mainly relies on collision resistance of hash functions, the padding fix does rely on additional assumptions on its underlying primitives; for example collision resistance of AES.

Overall, we conclude that lack of key commitment is easy to take advantage of, but also easy to fix.

Acknowledgments

The authors would like to thank Daniel Bleichenbacher for highlighting the impact of binary polyglots, Jean-Philippe Aumasson, Maria Eichlseder and Marc Stevens for helping with crypto-polyglots, and Peter Valchev and Christoph Kern for their helpful feedback.

This research was partly 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.

Availability

The polyglot and GCM tools along with proof-of-concepts are available at https://github.com/corkami/mitra.

References

[AA14] Ange Albertini and Jean-Philippe Aumasson. A binary magic trick, angecryption. *International Journal of Proof-of-Concept or GTFO*, 0x03:37–41, 2014. https://archive.org/details/pocorgtfo03.

[AAE+14] Ange Albertini, Jean-Philippe Aumasson, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Malicious hashing: Eve's variant of sha-1. In Antoine Joux and Amr Youssef, editors, *Selected Areas in Cryptography – SAC 2014*, pages 1–19, Cham, 2014. Springer International Publishing.

[ABL+14] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. How to Securely Release Unverified Plaintext in Authenticated Encryption. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 105–125. Springer, 2014.

[ABN10] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In Daniele Micciancio, editor, *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, volume 5978 of *Lecture Notes in Computer Science*, pages 480–497. Springer, 2010.

[Alba] Ange Albertini. Hash collisions and exploitations. Date Accessed: Oct. 13, 2020. https://github.com/corkami/collisions.

[Albb] Jim Albrecht. Introducing Subscribe with Google. Date Accessed: Oct 4, 2020. https://blog.google/outreach-initiatives/google-news-initiative/introducing-subscribe-google/.

[Alb15] Ange Albertini. Abusing file formats. *International Journal of Proof-of-Concept or GTFO*, 0x07:18–41, 2015. https://archive.org/details/pocorgtfo07.

[AWSa] AWS Encryption SDK. Date Accessed: Oct. 13, 2020. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/introduction.html.

[awsb] AWS Key Management Service concepts - AWS Key Management Service. Date Accessed: Sep 1, 2020. https://docs.aws.amazon.com/kms/latest/developerguide/concepts.html.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.

[BDPS12] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. Security of symmetric encryption in the presence of ciphertext fragmentation. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 682–699. Springer, 2012.

[BDPS13] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. On Symmetric Encryption with Distinguishable Decryption Failures. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 367–390. Springer, 2013.

[Ber05] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2005.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. http://cr.yp.to/papers.html#chacha, 2008.

[Bih94] Eli Biham. New types of cryptanalytic attacks using related keys. *J. Cryptol.*, 7(4):229–246, 1994.

[BMOS17] Guy Barwell, Daniel P. Martin, Elisabeth Oswald, and Martijn Stam. Authenticated encryption in the face of protocol and side channel leakage. In

Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 693–723. Springer, 2017.

[BN00] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki 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 531–545. Springer, 2000.

[BN08] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *J. Cryptology*, 21(4):469–491, 2008.

[BT16] Mihir Bellare and Björn Tackmann. The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, volume 9814 of *Lecture Notes in Computer Science*, pages 247–276. Springer, 2016.

[BZG20] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. Automating the development of chosen ciphertext attacks. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1821–1837. USENIX Association, 2020.

[CAE14] CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, May 2014. http://competitions.cr.yp.to/caesar.html. Date Accessed: 14 Oct 2020.

[CKVW10] Ran Canetti, Yael Tauman Kalai, Mayank Varia, and Daniel Wichs. On symmetric encryption and point obfuscation. In Daniele Micciancio, editor, *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, volume 5978 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 2010.

[Cry] CrystalOnScript. Protect your subscription content with client-side encryption. Date Accessed: Oct 4, 2020. https://amp.dev/documentation/guides-and-tutorials/develop/monetization/content_encryption/.

[DGRW18] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra 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.

[DGRW19] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. Cryptology ePrint Archive, Report 2019/016, 2019. https://eprint.iacr.org/2019/016.

[EPRS17] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In Jonathan Katz and Hovav 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 98–129. Springer, 2017.

[fil] Just Solve the File Format Problem. Date Accessed: Oct 4, 2020. http://fileformats.archiveteam.org/.

[FIM20] Rintaro Fujita, Takanori Isobe, and Kazuhiko Minematsu. ACE in chains: How risky is CBC encryption of binary executable files? In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020, Proceedings, Part I*, volume 12146 of *Lecture Notes in Computer Science*, pages 187–207. Springer, 2020.

[FJP13] Pierre-Alain Fouque, Jérémy Jean, and Thomas Peyrin. Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 183–203. Springer, 2013.

[FOR17] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. *IACR Trans. Symmetric Cryptol.*, 2017(1):449–473, 2017.

[GLL19] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. RFC 8452, April 2019.

[GLR17a] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav 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.

[GLR17b] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. Cryptology ePrint Archive, Report 2017/664, 2017. https://eprint.iacr.org/2017/664.

[goo] Envelope encryption | Cloud KMS Documentation | Google Cloud. Date Accessed: Sep 1, 2020. https://cloud.google.com/kms/docs/envelope-encryption.

[HKR15] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. Robust Authenticated-Encryption AEZ and the Problem That It Solves. In Elisabeth Oswald and Marc 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.

[HRRV15] Viet Tung Hoang, Reza Reyhanitabar, Phillip Rogaway, and Damian Vizár. Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance. In Rosario Gennaro and Matthew 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 493–517. Springer, 2015.

[IOM12] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. Breaking and Repairing GCM Security Proofs. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 31–49. Springer, 2012.

[ISO09] Information technology — Security techniques — Authenticated encryption. Standard, International Organization for Standardization, Geneva, CH, February 2009.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486. Springer, 2018.

[JNP15] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Deoxys v1.3. CAESAR submissions, 2015. http://competitions.cr.yp.to/round2/deoxysv13.pdf.

[KR07] Lars R. Knudsen and Vincent Rijmen. Known-key distinguishers for some block ciphers. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2007.

[KR14] Ted Krovetz and Phillip Rogaway. The OCB Authenticated-Encryption Algorithm. RFC 7253, May 2014.

[Kra] Dr. Hugo Krawczyk. The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-krawczyk-cfrg-opaque-03, Internet Engineering Task Force. Work in Progress.

[KY00] Jonathan Katz and Moti Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In Bruce Schneier, editor, *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2000.

[LGR] Julia Len, Paul Grubbs, and Tom Ristenpart. Re: [Cfrg] Second RGLC on draft-irtf-cfrg-hpke. Date Accessed: Oct 4, 2020. https://mailarchive.ietf.org/arch/msg/cfrg/culAsRgMOvpPQcdvmXBrb8Muai8/.

[LP20] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles: First chosen-prefix collision on SHA-1 and application to the PGP web of trust. In *USENIX Security Symposium*, pages 1839–1856. USENIX Association, 2020.

[MV04a] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Anne Canteaut and Kapalee Viswanathan, editors, *INDOCRYPT*, volume 3348 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2004.

[MV04b] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode of Operation (Full Version). *IACR Cryptology ePrint Archive*, 2004:193, 2004.

[nac] NaCl: Networking and Cryptography library. Date Accessed: Oct 4, 2020. https://nacl.cr.yp.to/, version 2016.03.15 of the index.html web page.

[nis] Lightweight Cryptography. https://csrc.nist.gov/projects/lightweight-cryptography. Date Accessed: 14 Oct 2020.

[NIS07a] NIST Special Publication 800-38C. Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality. National Institute of Standards and Technology, 2007.

[NIS07b] NIST Special Publication 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. National Institute of Standards and Technology, 2007.

[NIS16] NIST Special Publication 800-185. SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash. National Institute of Standards and Technology, 2016.

[NL15] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.

[Pro14] Gordon Procter. A security analysis of the composition of chacha20 and poly1305. *IACR Cryptology ePrint Archive*, 2014:613, 2014.

[Pro15] Gordon Procter. *The Design and Analysis of Symmetric Cryptosystems*. PhD thesis, 2015.

[RBB03] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[Rog04] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[RS06] Phillip Rogaway and Thomas Shrimpton. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. *IACR Cryptology ePrint Archive*, 2006:221, 2006.

[RVV16] Reza Reyhanitabar, Serge Vaudenay, and Damian Vizár. Authenticated encryption with variable stretch. In Jung Hee Cheon and Tsuyoshi 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 396–425, 2016.

[Sas11] Yu Sasaki. Meet-in-the-middle preimage attacks on AES hashing modes and an application to whirlpool. In *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 378–396. Springer, 2011.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 570–596. Springer, 2017.

[tin] GitHub - google/tink: Tink is a multi-language, cross-platform, open source library that provides cryptographic APIs that are secure, easy to use correctly, and hard(er) to misuse. Date Accessed: Oct 4, 2020. https://github.com/google/tink.

[Tri] Alex Tribble. Improved client-side encryption: Explicit KeyIds and key commitment. Date Accessed: Oct. 13, 2020. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/introduction.html.

[Vau02] Serge Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In Lars 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 534–546. Springer, 2002.

A Notation

The set of strings of length not greater than x bits is $\{0,1\}^{\leq x}$, and the set of strings of arbitrary length is $\{0,1\}^*$. Unless specified otherwise, all sets are subsets of $\{0,1\}^*$. If $X, Y \in \{0,1\}^*$, then $|X|$ is the length of X, and $X \parallel Y$ and XY denote the concatenation of X and Y.

Let ε denote the empty string, and let 0^n denote the n-bit string consisting of only zeros. Given a block size n, the function $\mathsf{len}_n(X)$ represents the length of X modulo 2^n as an n-bit string, and $X0^{*n}$ is X padded on the right with 0-bits

Table 4: A PNG/DICOM zipper (shown in HexII).

Offset	Content	PNG	DICOM
000	89 .P .N .G \r \n ^Z \n	magic sig.	
+08	00 00 01 52 .c .O .M .M	comment chunk {	
080	.D .I .C .M		magic sig.
+04	02 00 00 00 .U .L 04 00 C6 00 00 00		metadata tags {
150	.K ._ .3 .6 .3 . |		} // tags
+06	09 00 00 00 .O .B 00 00 62 00		private tag
160	00 00		{
+02	A0 B7 45 37	} // chunk	
+04	00 00 00 0D .I .H .D .R 00 00	body {	
1B0+08	00 00 00 00 .I .E .N .D	}	
1C0	AE 42 60 82	// body	
+04	|		} // tag
	08 00 16 00 .U .I 1A 00 .1 .. .2 ..		body {
540	FF FF FF FF FF FF FF FF FF 00		} // body

to get a string of length a multiple of n. If $X \in \{0,1\}^*$, then $|X|_n = \lceil |X|/n \rceil$ is X's length in n-bit blocks. The operation

$$X[1]X[2]\cdots X[x] \xleftarrow{n} X \tag{24}$$

denotes splitting X into substrings such that $|X[i]| = n$ for $i = 1,\ldots,x-1$, $0 < |X[x]| \le n$, and $X[1]\|X[2]\|\cdots\|X[x] = X$.

The set of n-bit strings is also viewed as the finite field $GF(2^n)$, by mapping $a_{n-1}\ldots a_1a_0$ to the polynomial $a(\mathrm{x}) = a_{n-1} + a_{n-2}\mathrm{x} + \cdots + a_1\mathrm{x}^{n-1} + a_0\mathrm{x}^{n-1} \in GF(2)[\mathrm{x}]$, and fixing an irreducible polynomial which defines multiplication in the field. For $n = 128$, the irreducible polynomial is $1 + \mathrm{x} + \mathrm{x}^2 + \mathrm{x}^7 + \mathrm{x}^{128}$, the one used for GCM.

The function $\mathsf{int}(Y)$ maps the j-bit string $Y = a_{j-1}\ldots a_1a_0$ to the integer $i = a_{j-1}2^{j-1} + \cdots + a_1 2 + a_0$, and $\mathsf{str}_j(i)$ maps the integer $i = a_{j-1}2^{j-1} + \cdots + a_1 2 + a_0 < 2^j$ to the j-bit string $a_{j-1}\ldots a_1a_0$. Let $\mathsf{inc}_m(X)$ denote the function which adds one modulo 2^m to X when viewed as an integer:

$$\mathsf{inc}_m(X) := \mathsf{str}_m(\mathsf{int}(X) + 1 \bmod 2^m).$$

Define $\mathsf{msb}_j(X)$ to be the function that returns the j most significant bits of X, and $\mathsf{lsb}_j(X)$ the j least significant bits.

The expression $a \stackrel{?}{=} b$ evaluates to $\top$ if a equals b, and $\bot$ otherwise.

For a keyed function defined on a domain $\mathsf{K} \times \mathsf{X}$, we write $F(K,X)$ and $F_K(X)$ interchangeably. If the function has three or more inputs, $\mathsf{K} \times \mathsf{N} \times \mathsf{X}$, then the second input can be written as a superscript, $F(K,N,X) = F_K^N(X)$. If $E : \{0,1\}^n \to \{0,1\}^m$ is a function, then the notation

$$F \leftarrow E(C \,\|\, \cdot) \tag{25}$$

defines F to be the function from $\{0,1\}^{n-|C|}$ to $\{0,1\}^m$ which maps an element $X \in \{0,1\}^{n-|C|}$ to $E(C \,\|\, X)$.

B Security Definitions

An adversary $\mathbf{A}$ is an algorithm which interacts with an oracle O. Let $\mathbf{A}^O = 1$ be the event that $\mathbf{A}$ outputs 1 when interacting with O, then define

$$\underset{\mathbf{A}}{\Delta}(f;g) := \left|\mathbf{P}\left[\mathbf{A}^f = 1\right] - \mathbf{P}\left[\mathbf{A}^g = 1\right]\right|, \tag{26}$$

which is the advantage of $\mathbf{A}$ in distinguishing f from g, where f and g are viewed as random variables. The notation can be extended to multiple oracles by setting $O = (O_1,\ldots,O_l)$.

We assume that all keyed functions do not change their output length under different keys, that is, $|F_K(X)|$ is the same for all $K \in \mathsf{K}$. Given a keyed function F, define $\$_F$ to be the algorithm which, given X as input, outputs a string chosen uniformly at random from the set of strings of length $|F_K(X)|$ for any key K. When given the same input, $\$_F$ returns the same output. Often $\$_F$ is called a *random oracle*.

Let $F : \mathsf{K} \times \mathsf{X} \to \mathsf{Y}, F' : \mathsf{K} \times \mathsf{X}' \to \mathsf{Y}'$ be PRFs, then the PRF advantage of adversary $\mathbf{A}$ against (F,F') is

$$\mathsf{PRF}_{F,F'}(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}\left(F_K, F'_K; \$_F, \$_{F'}\right), \tag{27}$$

where K is chosen uniformly at random from K.

Let $\Pi_K = (\mathsf{Enc}_K, \mathsf{Dec}_K)$ be an AE scheme using key K. Let $\Pi^\$ = (\$_{\mathsf{Enc}}, \bot)$ be an idealized AE scheme with the same interface as Π, and let $\Pi^\$_1, \Pi^\$_2, \ldots$ denote independent, idealized copies. Then the multi-key AE advantage of adversary $\mathbf{A}$ against Π is

$$\mu\text{-}\mathsf{AE}_\Pi(\mathbf{A}) := \underset{\mathbf{A}}{\Delta}\left(\Pi_{K_1}, \Pi_{K_2}, \ldots, \Pi_{K_\mu}; \Pi^\$_1, \Pi^\$_2, \ldots, \Pi^\$_\mu\right), \tag{28}$$

where $K_1,\ldots,K_\mu$ are chosen independently and uniformly at random, and $\mathbf{A}$ is *nonce-respecting*, meaning the same nonce is never queried twice to Enc. Nonces may be repeated with Dec. Furthermore, $\mathbf{A}$ cannot use the output of an O_1^N query as the input to an O_2^N with the same nonce N, otherwise such queries result in trivial wins.

C Algorithm Descriptions

In this section we provide descriptions of GCM, ChaCha20Poly1305, and OCB. The descriptions are

only given to the level of detail sufficient for the paper. The notation is borrowed from various sources: the description of OCB by Rogaway, Bellare, and Black [RBB03], the description of GCM by Iwata, Ohashi, and Minematsu [IOM12], and the documents by Procter analyzing ChaCha20Poly1305 [Pro15, Pro14].

C.1 GCM

We describe the GCM mode of operation [MV04a, MV04b] with nonces of 96-bit length. We let $E : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ denote a block cipher. The function GHASH is defined in Algorithm 3. Algorithm 5 provides pseudocode for GCM encryption, which also uses the keystream generator CTR mode in Algorithm 4. See Figure 8 for an illustration.

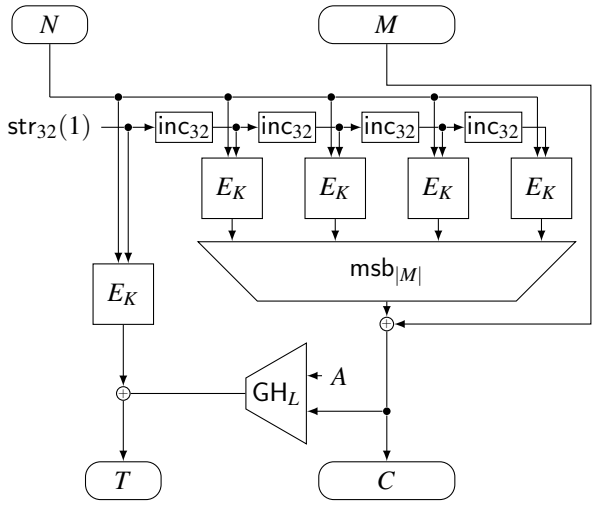

Figure 8: The GCM mode of operation with 96-bit nonces. GH is GHASH. The value L is $E_K(\mathsf{str}_n(0))$.

Algorithm 3: $\mathsf{GHASH}_L(A,C)$

Input: $L \in \{0,1\}^n$, $A \in \{0,1\}^{\leq n(2^{n/2}-1)}$, $C \in \{0,1\}^{\leq n(2^{n/2}-1)}$
Output: $Y \in \{0,1\}^n$

$X \leftarrow A0^{*n} \,\|\, C0^{*n} \,\|\, \mathsf{str}_{n/2}(|A|) \,\|\, \mathsf{str}_{n/2}(|C|)$
$X[1]X[2]\cdots X[x] \xleftarrow{n} X$
$Y \leftarrow 0^n$
for $j = 1$ **to** x **do**
  $Y \leftarrow L \cdot (Y \oplus X[j])$
end
return Y

Algorithm 4: $\mathsf{CTR}[F](X,m)$

Input: $F : \{0,1\}^x \to \{0,1\}^n, X \in \{0,1\}^x, m \in \mathbb{N}$
Output: $S \in \{0,1\}^{mn}$

$I \leftarrow X$
for $j = 1$ **to** m **do**
  $S[j] \leftarrow F(I)$
  $I \leftarrow \mathsf{inc}_x(I)$
end
$S \leftarrow S[1]S[2]\cdots S[m]$
return S

Algorithm 5: $\mathsf{GCM}_K(N,A,M)$

Input: $K \in \{0,1\}^{128}$, $N \in \{0,1\}^{128}$, $A \in \{0,1\}^{\leq 128\cdot 2^{32}}$, $M \in \{0,1\}^{\leq 128\cdot 2^{32}}$
Output: $(C,T) \in \{0,1\}^{\leq 128\cdot 2^{32}} \times \{0,1\}^{128}$

$L \leftarrow E_K(\mathsf{str}_{128}(0))$
$I \leftarrow N \,\|\, \mathsf{str}_{32}(1)$
$m \leftarrow |M|_{128}$
$F \leftarrow E_K\left(\mathsf{msb}_{96}(I) \,\|\, \cdot\right)$
$C \leftarrow M \oplus \mathsf{msb}_{|M|}\left(\mathsf{CTR}[F]\left(\mathsf{inc}_{32}(\mathsf{lsb}_{32}(I)), m\right)\right)$
$T \leftarrow E_K(I) \oplus \mathsf{GHASH}_L(A,C)$
return (C,T)

C.2 ChaCha20Poly1305

Our description of ChaCha20Poly1305 is taken from RFC 7539 [NL15], as well as Procter's analysis [Pro15, Pro14]. The combination of ChaCha20 [Ber08] and Poly1305 [Ber05] is similar to that of GCM, with the main differences being the fact that block cipher calls are replaced by ChaCha20's block function calls, and the key for Poly1305 is generated differently.

The ChaCha20 block function is denoted by

$$\mathsf{CC} : \{0,1\}^{256} \times \{0,1\}^{32} \times \{0,1\}^{96} \to \{0,1\}^{512}\,, \qquad (29)$$

which operates on keys of length 256 bits, a block number of length 32 bits, a nonce of length 96 bits, and with an output of size 512 bits. The Poly1305 universal hash function is denoted by

$$\mathsf{Poly} : \{0,1\}^{128} \times \{0,1\}^* \to \{0,1\}^{128}\,. \qquad (30)$$

The description of ChaCha20Poly1305 encryption is given in Algorithm 6.

C.3 OCB3

In this section we describe the OCB mode of operation [KR14, Rog04, RBB03]. We focus on OCB version 3 [RBB03], however our results extend to all versions of OCB. We do not include associated data as we do not need it for the OCB attacks.

Algorithm 6: $\mathsf{CC\&Poly}_K(N,A,M)$

Input: $K \in \{0,1\}^{256}$, $N \in \{0,1\}^{96}$,
$A \in \{0,1\}^{\leq 8\cdot(2^{64}-1)}$, $M \in \{0,1\}^{\leq 512\cdot(2^{32}-1)}$
Output: $(C,T) \in \{0,1\}^{|M|} \times \{0,1\}^{128}$

$F \leftarrow \mathsf{CC}_K(\cdot,N)$
$C \leftarrow M \oplus \mathsf{msb}_{|M|}\Big(\mathsf{CTR}[F]\,(\mathsf{str}_{32}(1),|M|_{512})\Big)$
$L \leftarrow \mathsf{msb}_{256}\,(F(\mathsf{str}_{32}(0)))$
$T \leftarrow \mathsf{lsb}_{128}(L) \oplus$
$\mathsf{Poly}_{\mathsf{msb}_{128}(L)}\Big(A0^{*128}\|C0^{*128}\|\mathsf{str}_{64}(|A|_8)\|\mathsf{str}_{64}(|C|_8)\Big)$
return (C,T)

The reference used for the figure, pseudocode, and notation below is from [RBB03]. Let $E : \mathsf{K} \times \{0,1\}^n \to \{0,1\}^n$ be a block cipher and let τ denote the tag length, which is an integer between 0 and n. Let $\gamma_1,\gamma_2,\ldots$ be constants, whose values depend on the version of OCB used; for example, in OCB1 [RBB03] these are Gray codes. Then Algorithm 7 gives pseudocode describing OCB encryption, and Figure 9 provides an accompanying diagram.

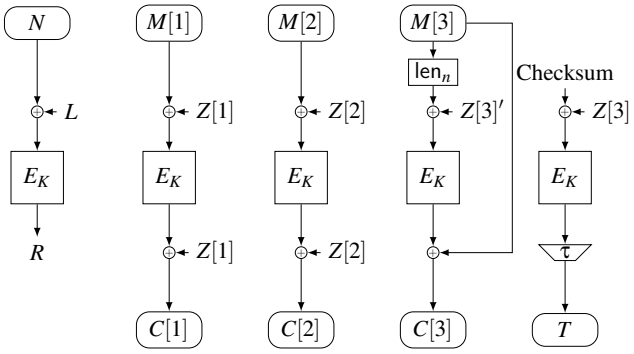

Figure 9: The OCB mode of operation applied to a plaintext of length at most three blocks. The value L is $E_K(0^n)$ and $Z[3]' = Z[3] \oplus L \cdot \mathrm{x}^{-1}$.

D Type I, II, and III $\mathsf{CommitKey}\Pi$ Encryption

E Algorithmic Descriptions of the Padding Fix

F 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' nonce, so $\nu' = 0$) and consider $\mathsf{CommitKey}_I\Pi$. Define F_{enc} and F_{com} as instantiated above with `SHA256`, with

Algorithm 7: $\mathsf{OCB}_K(N,M)$

Input: $K \in \{0,1\}^n$, $M \in \{0,1\}^*$
Output: $C \in \{0,1\}^*$

$M[1]M[2]\cdots M[m] \xleftarrow{n} M$
$L \leftarrow E_K(0^n)$
$R \leftarrow E_K(N \oplus L)$
for $i = 1$ **to** m **do**
  $Z[i] = \gamma_i \cdot L \oplus R$
end
for $i = 1$ **to** m **do**
  $C[i] \leftarrow E_K(M[i] \oplus Z[i]) \oplus Z[i]$
end
$X[m] \leftarrow \mathsf{len}_n(M[m]) \oplus L \cdot \mathrm{x}^{-1} \oplus Z[m]$
$Y[m] \leftarrow E_K(X[m])$
$C[m] \leftarrow Y[m] \oplus M[m]$
$\mathrm{Checksum} \leftarrow M[1] \oplus \cdots \oplus M[m-1] \oplus C[m]0^{*n} \oplus Y[m]$
$T \leftarrow \mathsf{msb}_\tau\Big(E_K(\mathrm{Checksum} \oplus Z[m])\Big)$
return $C[1]\cdots C[m]T$

Algorithm 8: $\mathsf{CommitKey}_I\Pi^{\mathsf{Enc}}(K,N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\mathrm{com}} \in \{0,1\}^c$

$K_{\mathrm{enc}} \leftarrow F_{\mathrm{enc}}(K)$
$K_{\mathrm{com}} \leftarrow F_{\mathrm{com}}(K)$
$C \leftarrow \mathsf{Enc}(K_{\mathrm{enc}},N,A,M)$
return (C,K_{com})

Algorithm 9: $\mathsf{CommitKey}_{II}\Pi^{\mathsf{Enc}}(K,N_1,N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N_1 \in \{0,1\}^{\nu_1}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\mathrm{com}} \in \{0,1\}^c$

$K_{\mathrm{enc}} \leftarrow F_{\mathrm{enc}}(K,N_1)$
$K_{\mathrm{com}} \leftarrow F_{\mathrm{com}}(K)$
$C \leftarrow \mathsf{Enc}(K_{\mathrm{enc}},N,A,M)$
return (C,K_{com})

Algorithm 10: $\mathsf{CommitKey}_{III}\Pi^{\mathsf{Enc}}(K,N_1,N,A,M)$

Input: $K \in \{0,1\}^{\kappa_0}$, $N_1 \in \{0,1\}^{\nu_1}$, $N \in \mathsf{N}$, $A \in \mathsf{A}$, $M \in \mathsf{M}$
Output: $C \in \mathsf{C}$, $K_{\mathrm{com}} \in \{0,1\}^c$

$K_{\mathrm{enc}} \leftarrow F_{\mathrm{enc}}(K)$
$K_{\mathrm{com}} \leftarrow F_{\mathrm{com}}(K,N_1)$
$C \leftarrow \mathsf{Enc}(K_{\mathrm{enc}},N,A,M)$
return (C,K_{com})

Algorithm 11: $\mathsf{Pad}_\ell\Pi^{\mathsf{Enc}}(K,N,A,M)$

Input: $K \in \mathsf{K}, N \in \mathsf{N}, A \in \mathsf{A}, M \in \mathsf{M}$
Output: $C \in \mathsf{C}$
return $\mathsf{Enc}(K,N,A,\mathsf{str}_\ell(0) \,\|\, M)$

Algorithm 12: $\mathsf{Pad}_\ell\Pi^{\mathsf{Dec}}(K,N,A,C)$

Input: $K \in \mathsf{K}, N \in \mathsf{N}, A \in \mathsf{A}, C \in \mathsf{C}$
Output: $M \in \mathsf{M} \cup \{\perp\}$
$M \leftarrow \mathsf{Dec}(K,N,A,C)$
if $M \neq \perp$ *and* $\mathsf{msb}_\ell(M) \stackrel{?}{=} \mathsf{str}_\ell(0)$ **then return** M
return $\perp$

$L_{\text{enc}} = L0 \,\|\, 0x01 \,\|\, 0x01$ and $L_{\text{com}} = L0 \,\|\, 0x01 \,\|\, 0x02$ where $L0 =$ `0x41455347434d`, which is `AESGCM` in hexadecimal notation. The resulting $\mathsf{CommitKey}_I\Pi$ is a "key committing AES-GCM". At setup, encryption requires AES key expansion for the encryption key K_{enc}, and also one computation of $\text{AES}(K_{\text{enc}}, 0^{128})$ for the GHASH key. This setup overhead is the same as the setup for AES-GCM.

G Instantiating the Bounds of Theorem 3

We show how to use the bounds of Theorem 3. Consider the case where $\kappa_0 = \kappa = c = 256$ and Π is the 96-bit-nonce AES-GCM. Suppose that a main key K is used $q \leq 2^{32}$ times, with different nonces $N'_1, \ldots N'_q$ resulting in derived values $K_{\text{enc}\,i} = F_{\text{enc}}(K, N'_i)$ and $K_{\text{com}\,i} = F_{\text{com}}(K, N'_i)$. Every nonce from $N'_1, \ldots, N'_q$ and the respective derived key is used to encrypt 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 $K_{\text{enc}\,j}$. The total number of blocks encrypted with $K_{\text{enc}\,j}$ 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 K_{enc} and the q values of K_{com} are distinct. With T_0 key guessing attempts (for either K or a derive K_{enc}), correct guessing succeeds with probability $(T_0 q)/2^{\kappa}$. Therefore, we can upper bound the advantage of an adversary against $\mathsf{CommitKey}\Pi$ by

$$\max \mathsf{PRF}(F_{\text{enc}}, F_{\text{com}}) + \frac{4q^2}{2^{\kappa+1}} + T_0\frac{q}{2^{\kappa}} + \sum_{j=1}^{q} \frac{(\sigma_j + \bar{q}_j + 1)^2}{2^{129}} \tag{31}$$

where $\max \mathsf{PRF}(F_{\text{enc}}, F_{\text{com}})$ 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 (31) is at most $2^{32} \times 2^{-65} = 2^{-33}$. With a judicious choice for a PRF F (e.g., `SHA256`), 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 $O(2^{-32})$.

H Example

This file is actually a proof of concept itself, containing this paper under its PDF form, but also a PDF viewer as a Windows executable. The parameters are :

- $K_1 =$ `0x4e6f773f00000000000000000000000`,
- $K_2 =$ `0x4c3474337221212100000000000000`,
- $N =$ `0x000000000000000000e7c6`,
- $A =$ `0x4d79566f69636549734d795061737321`.

If this PDF is encrypted with K_1 and decrypted with K_2, it will give you a fully working executable which is a PDF viewer. There is no widely available CLI for AES-GCM, but OpenSSL can be used to simulate the encryption/decryption process without authentication in the following way:

```
openssl enc -in paper.pdf -out ciphertext
  -aes-128-ctr
  -iv 000000000000000000e7c600000002
   -K 4e6f773f00000000000000000000000
openssl enc -in ciphertext -out viewer.exe
  -aes-128-ctr
  -iv 000000000000000000e7c600000002
   -K 4c3474337221212100000000000000
```