

Cryptographic Engineering

working draft

Matthew D. Green
Johns Hopkins University

Excerpt: Chapter 5: Symmetric encryption and authentication
§5.3.1 (attacks on unauthenticated encryption), §5.3.2, §5.3.4, §5.3.6, §5.4.5–5.4.6
(book pages 73–77, 79–84, 89–89, 94–96)

Posted for 601.445/601.645 Practical Cryptographic Systems, Fall 2026
reading for Monday, September 14, 2026

<https://practicalcrypto.org>

This is an excerpt from an unfinished book draft: expect gaps, rough edges, and errors, and note that section numbers and page numbers may change in later drafts. Cross-references to other chapters point to material that is not included here. Copyright © 2026 Matthew D. Green. All rights reserved. Posted for the students of the course above; **please do not redistribute** — link to the course site instead. Built 2026-09-02.

Attacks on unauthenticated encryption

Once upon a time, cryptographers believed that confidentiality and authenticity were largely unrelated: that is, both properties were necessary and desirable, but they didn't necessarily interact. Cryptographers have learned differently, often to their dismay. This is best illustrated by examining the history of active attacks on unauthenticated encryption schemes.

A common threat facing real-world encryption schemes occurs when the attacker can access a *decryption oracle*, which is some system that possesses the decryption key and is willing to decrypt (or at least, partially decrypt) ciphertexts that the attacker submits. This might seem unrealistic, but it isn't! For example, access to a decryption oracle is almost unavoidable in network encryption protocols, given that one side of the connection is almost certainly receiving and decrypting all packets it receives. An active attacker can make its own fresh packets, or capture encrypted packets from the legitimate sender and modify them; in either case it can then send the new packets onwards to the decryptor. The main open question is what, exactly the attacker can learn from this experiment.

Interestingly, a problem with many unauthenticated encryption schemes is that attackers can learn quite a lot. Even if the remote decryptor does not send back the full decryption of a given packet (which would be surprising), the decryptor will reveal information like “is this decrypted message correctly formatted, or should I report an error.” These small bits of information can often be enough to gradually work out the contents of a legitimate encrypted message, if the attacker is allowed to modify the target ciphertext and re-transmit the many changed copies.¹³

Padding oracle attacks. One class of chosen-ciphertext attacks are called *format oracle* attacks. Here the decryptor first decrypts a received message, and then validates that the underlying plaintext is correctly formatted; if not, it exposes some visible information (such as an error response) that indicates that the formatting is invalid. Format oracles could involve everything from verifying a checksum, to de-compressing a message, or even (in unusual cases) something as bizarre as running an encrypted software update contained within the message!¹⁴

One extremely common subclass of format oracle is the *padding oracle*: this applies to encryption schemes such as CBC mode, where the message will typically be padded with additional bytes to bring it to a multiple of the cipher's block length. For example, in the PKCS #7 padding standard [180, §10.3] mentioned above, an 11-byte message encrypted using AES will have five bytes (“octets”) appended to it, each containing the length of the pad.¹⁵ Here we represent the full block in hexadecimal, with MM representing message bytes:

¹³Of course, there are also cases where an attacker is allowed to see the full decryption of a message! Usually this is more common in public-key encryption schemes, which we'll discuss later, but it could happen if an attacker is briefly granted access to a decryption machine.

¹⁴Note that I've never seen anyone exploit an unauthenticated software update, but I'm still waiting!

¹⁵For messages that are already aligned to the cipher's block length, an entire additional block of padding is added to ensure there is never any confusion.

MM MM MM MM MM MM MM MM MM MM MM MM 05 05 05 05 05

Under ordinary circumstances, a recipient will first decrypt the message, then it will examine the final byte of the message to learn the padding length. It will typically verify that each byte of the padding contains the same value (here 05), and then strip away the padding to reveal the actual message. If it encounters a surprise during any step of this processing, many implementations will produce an error response, or even some visible reaction, that the attacker might be able to observe.

Exploiting a padding oracle. When the encryption scheme used is *malleable* to the XOR operation (as in CTR and CBC mode), an attacker can capture an unknown ciphertext sent by a legitimate encryptor, and exploit the padding oracle in order to learn its plaintext contents. This is done by mauling the ciphertext to change the contents of the underlying plaintext, and submitting the new mauled ciphertexts to the oracle. A common strategy is as follows:

1. First, learn the padding length by starting at the rightmost byte and mauling exactly one byte of the plaintext. If the ciphertext produces a padding error, this indicates that the mauling affected the padding. Start with the original ciphertext and repeat this experiment with the next byte to the left, to see if that is also padding. When no error is produced, you know the length of the padding L .
2. With the padding length L in hand, the attacker now knows the exact content of the rightmost L bytes. It can now maul these padding bytes so that each one contains the integer value $L + 1$.
3. Submit the resulting mauled ciphertext to be decrypted. If the final byte of the (unpadded) message M happens to satisfy $M = L + 1$, this experiment will *not* produce a padding error. This immediately reveals the value of the last byte of the message!
4. If this experiment does produce a padding error, then you've learned that $M \neq L + 1$. To find the last byte of the message M , perform the following process: for $i = 1$ to 255, maul the ciphertext again by XORing the byte value i into the original (unmodified) ciphertext at the position of M . This will convert the value of the final message to be $M \oplus i$. Submit the resulting ciphertext for decryption. If no padding error is received, you now know that $M \oplus i = L$, which reveals the value of M .
5. Once the attacker learns the final byte of the message contents M , it can maul the ciphertext such that the final byte of the message contains $L + 1$ (*i.e.*, it appears to be padding) and it can continue the attack starting from step (2) to recover the next message byte to the left. When all bytes in the final block of the message are known, the entire final block of the ciphertext can be truncated away and the process can repeat with the new final block of the ciphertext.

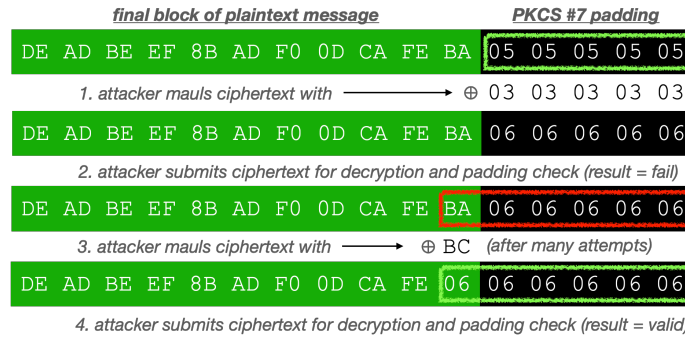


Figure 5.4: Illustration of several steps of a padding oracle attack. Each row shows the decrypted contents of the final block of a padded ciphertext. The attacker first learns the padding length (not shown), then mauls the ciphertext to increment the value of the padding bytes (step 1.) It submits the resulting ciphertext for decryption (step 2), which produces a padding error because the final message byte doesn’t fit the padding. For $i = 0x01$ to $0xFF$ it mauls the final byte of plaintext by XORing i at this position (one example shown in step 3), until it identifies a value (here $i = 0xBC$) where the padding check succeeds (step 4.) This reveals that the final plaintext byte $M = 0x06 \oplus 0xBC = 0xBA$.

Who’s afraid of format oracles? I wish I could tell you that format and padding oracles are just a teaching exercise, but over my career I’ve run into dozens of exploitable cases. For selfish reasons my favorite example is one that appeared in Apple’s iMessage encryption system way back in 2015, where a nifty format oracle involving `gzip` compression made it possible for my students to decrypt attachments sent between iPhones [129]. But they also turned up in SSL/TLS (up through v1.2) [11] and many other protocols that contain legacy cryptography [212, 169]. I’m sure there are still a few out there waiting to be caught.

For what it’s worth I hope you never have to actually worry about format or padding oracles in new systems, because you shouldn’t need to: proper use of *authenticated encryption* completely prevents all of these attacks. That’s because in an authenticated encryption scheme, any change to the ciphertext will cause the decryptor to immediately reject it *regardless of the plaintext contents*. I’m not describing this attack here because I hope you have to think about it (although it can be fun to implement as an exercise) but as a clear warning of why we should never, ever use unauthenticated encryption in production systems. In the next sections, we’ll talk about how authenticated encryption works under the hood.

5.3.2 Authenticated modes of operation

Authenticated encryption (AE) combines the confidentiality properties of an encryption scheme with the authentication properties of a MAC. Indeed, one can construct authenticated

Mode	Type	Cipher ops	Online?	Expansion	Standardization
CTR + HMAC-SHA-2	Generic	$\approx 2N^\dagger$	Yes	$t = \text{MAC} $	Components NIST [111, 239]
CCM	Two-pass	$\approx 2N + 2$	No	$t \leq b$	NIST [112], ISO/IEC [164]
EAX	Two-pass	$\approx 2N + 2$	Yes	$t \leq b$	<i>not</i> NIST/ISO; ISO/IEC [164]
GCM	Hash-based	$\approx N + 1^*$	Yes	$t = b$	NIST [113], ISO/IEC [164]
ChaCha20-Poly1305	Hash-based	$\approx N + 1^\dagger$	Yes	$t = 128$ bits	<i>not</i> NIST/ISO; IETF [242]
OCB3	“Tweak”-based	$\approx N + 2$	Yes	$t \leq b$	<i>not</i> NIST/ISO; IETF [189]

Table 5.1: Comparison of AE(AD) modes. b is the block size of the underlying block cipher; N is the number of data blocks in the plaintext (of size b ; for ChaCha20, $b = 512$). Entries count only block/stream-cipher invocations per message; constants absorb per-message setup. $*$ GCM also performs $\approx N + 1$ multiplications in $\text{GF}(2^b)$; the $E_K(0^b)$ subkey is per-key. † One extra ChaCha20 block derives the Poly1305 key; Poly1305 work is linear in 16 B chunks of AAD/ciphertext. ‡ This simplifies by assuming HMAC-SHA2 cost is about the same as the cipher, which isn’t necessarily true (more precisely: HMAC adds about $2N_h$ hash compression calls, where N_h depends on the hash’s block size).

encryption “generically” by combining the two primitives, but I would strongly discourage using this approach. Instead I recommend you use a modern *authenticated mode of operation* that provides the full functionality in a single efficient and standardized package. We’ll discuss several of these below; see Table 5.1 for a quick summary.

AE vs. AEAD. Way back at the start of this chapter, I recommended that you use AEAD schemes, and gave a high-level overview of how they worked. In this section we’re discussing authenticated encryption which protects ciphertext (and implicitly plaintext it contains.) However in many circumstances it’s useful to be able to protect additional data that isn’t actually encrypted within the ciphertext. Many authenticated encryption schemes support this by taking an *associated data* string as input during both encryption and decryption. That data isn’t actually encrypted into the packet, but the resulting ciphertext does authenticate it: that is, if the associated data passed in during encryption and decryption is different, decryption will fail with an error. Several schemes below will support this feature.

Generic authenticated encryption

As mentioned above, it’s possible to construct authenticated encryption “generically” from an unauthenticated encryption scheme along with a secure MAC. Let $\text{Encrypt}(K, M; \text{IV})$ be a secure unauthenticated encryption mode such as CBC or CTR mode (here we will specify the IV explicitly as an input to the algorithm.) Let MAC be a secure MAC algorithm. We will require that the parties possess two separate keys $K = (K_{\text{ENC}}, K_{\text{MAC}})$.¹⁶

¹⁶As a general rule we don’t recommend using the same key for encryption and MAC because there might be interactions between the two algorithms if the same key is used. This is particularly true when the encryption and MAC scheme both use the same underlying ingredient, such as a block cipher.

Encryption. To encrypt a message M , the encryptor first generates an appropriate IV and then follows these steps:

1. Compute $C' \leftarrow \text{Encrypt}(K_{\text{ENC}}, M; IV)$.
2. Compute $T \leftarrow \text{MAC}(K_{\text{MAC}}, IV \| C')$.
3. Output the authenticated ciphertext $C \leftarrow (IV, C', T)$.

Decryption. To decrypt the ciphertext C , the decryptor follows these steps:

1. Parse C into (IV, C', T) .
2. Verify that $T \stackrel{?}{=} \text{MAC}(K_{\text{MAC}}, IV \| C')$. If not, cease further processing and output **error**.
3. Output the message $M \leftarrow \text{Decrypt}(K_{\text{ENC}}, C'; IV)$.

A critical element of the construction above is that the decryptor never uses K_{ENC} unless it has previously verified the MAC tag T over the ciphertext and IV . This ensures that this key is only exposed to authentic ciphertexts (*i.e.*, ciphertexts created by a legitimate encryptor.) Hence all adversarial tampering with the ciphertext (including the IV) should be caught and rejected before the decryption key is used. In the literature, this approach is sometimes called “Encrypt-then-MAC”, and it provably meets strong notions of AE security if the encryption and MAC are each secure [42].

Insecure alternatives. It’s worth noting that there are other ways to combine encryption and MACs that are *not* necessarily secure [42]. These include:

1. **MAC then encrypt (insecure).** Some older versions of TLS compute a MAC tag T over a plaintext message M , and then encrypt the combined string $(M \| T)$. Unfortunately this can lead to real-world attacks, particularly when vulnerable encryption padding is added by the encryption scheme. For example, padding oracles (discussed above) can potentially decrypt these ciphertexts entirely.
2. **Encrypt and MAC (insecure).** Another common mistake is to compute the MAC tag T over a plaintext M , encrypt M to produce a ciphertext C' , and then compute the final ciphertext as $C = (C', T)$. The problem with this approach twofold: first, MACs aren’t guaranteed to provide *confidentiality*, and so the tag T could leak useful information about the message. Even in the best case, the MAC algorithm is not *randomized*, and hence an attacker may be able to tell whether two ciphertexts encrypt the same plaintext.¹⁷

¹⁷Some legacy protocols like SSH (2.0) use Encrypt-and-MAC, but they concatenate the message with a non-repeating per-message nonce before computing the MAC [41]. When using an appropriate MAC this will block the confidentiality leak mentioned above, but overall it’s better not dance on the lip of a volcano.

received plenty of formal security analysis [175, 126, 162].

The major criticisms of CCM fall into two categories. For one thing, CCM is less efficient than some of the other AEAD schemes we’ll discuss further below. This is due to the fact that each plaintext block must be processed twice with the block cipher, once for encryption and once to compute the MAC. Secondly, the scheme is not “online”: this means that the length of the associated data and plaintext message must be known *before* encryption can begin, which could make CCM inappropriate if you’re encrypting streaming data where the length isn’t known at the start of a message. CCM also does not allow pre-processing of the associated data string. Finally, since CCM is based on CTR-mode encryption, the accidental re-use of a nonce/IV can potentially result in the loss of confidentiality for both messages encrypted this way, although this will *not* affect the security of other ciphertexts where nonces aren’t re-used.²⁰

EAX mode. Another two-pass mode is called EAX [45]. EAX is essentially what CCM mode would have been, if it had been developed using best practices: it is an “Encrypt-then-MAC” composition that uses CTR mode to first encrypt, then applies a CBC-based MAC (CMAC/OMAC) to the resulting ciphertext. The key design challenge is to use a single key for both primitives, which EAX achieves by being careful about what it feeds into the MAC function. EAX has some real advantages over CCM, namely, that it is *online* and also allows pre-processing of the associated data. Sadly, nobody seems to use EAX and it isn’t standardized by NIST, though an ISO standard does cover it [164]. Even more sadly, the ANSI standards body *did* standardize a variant called EAX’, which was apparently used in smart electricity meters. EAX’ was horrifically broken in 2012 [223], which reinforces my view that nothing good has ever come from ANSI (or electricity meters.)

5.3.4 Hashing-based modes: GCM, GHASH and Poly1305

While two-pass modes like CCM do the job, these modes require two invocations of the block cipher for each block of plaintext. This is unsatisfying for applications with tight performance requirements, such as embedded applications where every cycle is at a premium. To address this need, some alternative approaches have been proposed. The most popular of these employs *polynomial hashing* in order to construct a very efficient authenticated encryption mode. The most popular examples of this approach are GCM-mode (typically used with AES) and the ChaCha20-Poly1305 scheme.

Intuition: universal hashing. Many *non-cryptographic* hash functions are used in computer science. One important class are the *universal hash functions* (§4.7), which are a

²⁰This is a contrast with schemes such as GCM and Poly1305-based schemes, where a few instances of nonce re-use can leak the scheme’s internal authentication key and thus allow arbitrary forgeries. See [162] for more analysis and security proofs.

special class of keyed functions $h_k()$ that have excellent properties when used in non-adversarial settings. As we discuss in §4.7, a powerful property of universal hashing is that, on average, these functions “distribute” their outputs well across the function’s digest space. For example, given any two input messages $M_1 \neq M_2$ and a key k chosen randomly (and independently of the messages), then the probability of a collision $h_k(M_1) = h_k(M_2)$ should be bounded. A bit more precisely, a keyed hash family $\{h_k\}_{k \in \mathcal{K}}$ is called *ϵ -almost universal* if, for every fixed pair of distinct messages $M_1 \neq M_2$, the following condition holds:²¹

$$\Pr_{k \leftarrow \mathcal{K}} [h_k(M_1) = h_k(M_2)] \leq \epsilon.$$

As a side benefit, universal hash functions can be built using relatively “cheap” ingredients, such as simple arithmetic in prime or binary fields.

These functions might sound almost too good to be true, and they (sort of) are. The challenge with using them in cryptography is that they tend to be *wildly insecure* against malicious behavior. An attacker who learns the key k can typically find collisions with very little effort. You might try to keep the key secret; unfortunately, when a single key is used to hash many messages, it’s typically possible to recover the key given a relatively small number of distinct input/output pairs $(M, H_k(M))$.²²

From universal hash to symmetric MAC. Despite these weaknesses, some clever researchers realized that they could use universal hash functions to build very efficient MACs, as long as they were careful to take two precautions:

1. The key k must look random and must be kept secret from the attacker.
2. The attacker must never be allowed to see any output $h_k(M)$ from the hash function.

Fortunately in the world of symmetric cryptography, both goals are easy to achieve: we’re already in a model where the honest parties have shared secret keys, and the attacker is assumed not to know them. Moreover, with access to a cipher we can *encrypt* any hash outputs before transmitting them to the decryptor/verifier: this ensures that the adversary will never learn these values. Done properly, this “encrypted hash using a secret key” is a very secure MAC, and one that’s often screamingly efficient to compute, since the more expensive cipher only needs to be invoked once to protect the digest. The security of these MACs then comes down to a question of whether the attacker might somehow be able to modify or “maul” the encrypted MAC to a new useful value. If the MAC is encrypted with something like CTR-mode, then we can defend ourself against any XOR-based mauling if the function possesses an additional property called *XOR-universality* (see §4.7). Given a

²¹The probability is taken over the uniform random choice of k , while M_1 and M_2 are fixed independently of that choice. If the function has D possible digest values, a universal family conventionally achieves the bound $\epsilon = 1/D$; families with a larger specified bound are called almost universal.

²²One classical approach to solving this problem is to use keys that are as long as all the messages authenticated: this is called a Carter-Wegman authenticator (§4.7.1.) This approach provides *information-theoretic* security, with some obvious downsides for efficiency.

random key and a CTR-encrypted copy of $h_k(M)$, this property ensures that the adversary should be hard-pressed to find a string Δ such that $h_k(M) \oplus \Delta = h_k(M')$ for any chosen M' .

Review: polynomial hash functions. Universal hashing for arbitrary-length messages can be performed using a *polynomial* hash function as discussed in §4.7.1. These can hash large quantities of data using only relatively lightweight arithmetic operations in a prime or binary extension field. Polynomial hashing powers GCM-mode’s **GHASH** and the Poly1305 scheme we’ll describe below.

Let $\mathbb{F}$ be a field, and let $k \in \mathbb{F}$ be a non-zero random secret key. At a high level, a standard polynomial hash function works by first chopping the message M into a series of field elements $M_1, \dots, M_{n-1} \in \mathbb{F}$ (padding the last block if needed) and then encoding the bit-length of the message into a final element M_n . It then treats the values $M_1, \dots, M_n$ as coefficients of a degree- n polynomial $p()$, where the constant term of the polynomial M_0 does not contain a message block.²³ Next, it evaluates the polynomial at the point specified by the secret key k . For example:

$$h_k(M) = p(k) = \sum_{i=1}^n M_i \cdot k^i$$

With some optimizations, these polynomials can be evaluated iteratively using Horner’s method (§4.2) at the cost of one field addition and one field multiplication per plaintext block.²⁴ To optimize the speed of these operations, specific fields are used in real constructions: **GHASH** uses a binary extension field, while Poly1305 uses a prime field that’s chosen for speed in software. Modern processors also contain instructions for carry-less field multiplication to speed up GCM mode.

Warnings. As mentioned above, a key danger (pun intended!) in polynomial hash functions is that an attacker who learns a few pairs $(M, h_k(M))$ can potentially recover the key k . Even learning simple algebraic functions of these hash outputs, such as $M_1, M_2, h_k(M_1) + h_k(M_2)$ can lead to key recovery, given a few such examples. Encryption schemes using these hashes must be very careful to ensure that these hash outputs are always securely encrypted. As foreshadowing: this means that if you use something like CTR-mode to encrypt the hash, you must be careful never to reuse a nonce!

²³It’s absolutely critical for our purposes that, when this polynomial is evaluated at some point k , no part of the message be included in the “constant” term, *i.e.*, every part of the message must be multiplied by some power of k . This structure is needed to retain XOR-universality, as mentioned above. For a simple counterexample, if you place the first block of the message into the constant term at M_0 , then the hash of a one-block message might be $h_k(M_0) = M_0$, which will make it very easy to “maul” the hash of one message into a different message using XOR only.

²⁴A convention used by **GHASH** and Poly1305 is to assign the first encoded message block the highest power of the hash key. For encoded blocks $M_1, \dots, M_n$, setting $P_0 = 0$ and computing $P_i \leftarrow (P_{i-1} + M_i)k$ in the relevant field yields $P_n = M_1k^n + \dots + M_nk$. In **GHASH**, field addition is XOR; in Poly1305, each encoded message block includes its appended 1 bit and arithmetic is performed modulo $2^{130} - 5$.

GCM mode

With these ingredients in hand, we’re ready to discuss GCM mode [113, 213]. GCM is an AEAD that requires the encryptor to specify a nonce, which can never be re-used across two different messages. While GCM can in principle be used with many block ciphers, it was primarily designed to be used with AES: hence the standardized description I’ll give below is optimized for AES’s $b = 128$ -bit block size. This includes the design of the GHASH hash function (given in Algorithm 1), which operates over a field $\text{GF}(2^{128})$ defined by the polynomial $x^{128} + x^7 + x^2 + x + 1$.

Encryption. Encryption requires users to provide a unique nonce N for each message to be encrypted. The user can also provide an optional *associated data* string AD that will be authenticated by the algorithm. Here we will provide a simplified description that assumes the nonce is 96 bits, and authentication tags are 128-bits long.²⁵ The function $\text{zeropad}_{\text{bits}}(S)$ outputs a sufficient number of zero padding bits to pad S to a multiple of 128 bits. We’ll also define the function $\text{len_bits}_{64}(\cdot)$ to output a 64-bit unsigned integer containing the length of a string in bits. Given (K, N, AD, M) , encryption proceeds as follows:

1. Compute a GHASH key $H \leftarrow \text{E}_K(0)$.
2. Assuming $|N| = 96$ bits: initialize a 32-bit unsigned counter $\text{ctr} \leftarrow \bar{0}1$.²⁶
3. Set $J_0 \leftarrow (N \parallel \text{ctr})$.
4. Let C' be the CTR-mode encryption of M under K , using initial counter $J_0 + 1$.
5. Now compute:

$$S \leftarrow \text{GHASH}_H(\text{AD} \parallel \text{zeropad}_{\text{bits}}(\text{AD}) \parallel C' \parallel \text{zeropad}_{\text{bits}}(C') \parallel \text{len_bits}_{64}(\text{AD}) \parallel \text{len_bits}_{64}(C'))$$

6. Let $T \leftarrow \text{E}_K(J_0) \oplus S$.
7. Return $C \leftarrow (C' \parallel T)$.

Decryption. Decryption is largely the same as encryption. On input (K, N, AD, C) , the decryptor parses C into (C', T) and then re-computes the GHASH key H and initial counters as above. It applies GHASH to the associated data AD and ciphertext C' and re-computes an authentication tag T' using the same procedure used during encryption. If the check $T' \stackrel{?}{=} T$ does not succeed, it outputs *error*. Otherwise it decrypts C' using the appropriate counter, and outputs the plaintext.

Implementation warnings and security. GCM is provably secure under the assumption that the block cipher is a pseudorandom permutation [213], and that it is used correctly. The biggest footgun when using GCM is that encryptors must never, ever re-use the same nonce N to encrypt two different messages/associated data strings. Partly this is due to the

²⁵In practice GCM can use other nonce lengths and shorter authentication tags. See [113].

²⁶If the nonce length is not 96 bits, a more complex process is used. This involves hashing the nonce using GHASH and using the result to construct the value J_0 . See [113, Algorithm 4].

Algorithm 1 GHASH [113, 213]

Require: Hash key $H \in \text{GF}(2^{128})$

Require: A message comprising m 128-bit blocks $X_1, \dots, X_m$.

Ensure: 128-bit authentication tag Y

```

1:  $Y \leftarrow 0$   $\triangleright$  Initialize the accumulator
2: for  $i \leftarrow 1$  to  $m$  do
3:    $\triangleright \oplus$  is addition,  $\otimes$  is multiplication in  $\text{GF}(2^{128})$  with reduction mod  $x^{128} + x^7 + x^2 + x + 1$ .  $\triangleleft$ 
4:    $Y \leftarrow (Y \oplus X_i) \otimes H$ 
5: return  $Y$ 

```

weakness of CTR-mode encryption when nonces are repeated. But the issue in GCM is much worse than that. Notice that the GHASH output S is also encrypted using CTR-mode. If an attacker sees tags T_1, T_2 from different messages, where both are encrypted with the same key and nonce, they can XOR the tags to obtain $S_1 \oplus S_2$ where S_1, S_2 are the individual GHASH outputs. Since $\oplus$ is the addition operation in the field $\text{GF}(2^{128})$, a few known input/hash pairs can be used to recover the hash key H . With knowledge of H , an attacker can forge tags on new messages as long as the key K is used. This completely breaks the authenticity guarantees offered by GCM.

A related problem with GCM is that the standard uses a 96-bit nonce N by default. If this tag is selected randomly, then after approximately 2^{48} ciphertexts the probability of a nonce repeat is quite high.²⁷ For this reason, NIST specifies that implementations should re-key after encrypting approximately 2^{32} (about 4 billion) messages under a given key.²⁸ While 4 billion messages is a very large number, many network encryption protocols will routinely encrypt huge numbers of messages, particularly if a shared key is used across multiple devices. I have been in design meetings where this limit has posed a real problem for systems that you probably use.

To address both of these issues, new modes like GCM-SIV have been developed [143]. These modes are *misuse-resistant*, which means that these vulnerabilities are prevented by design. Unfortunately they require slightly more computation, and are not yet standardized by NIST.

Don't use short authentication tags. When GCM was originally specified, it was proposed that the authentication tags could be shortened, for applications where bandwidth was at a premium. Subsequently, researchers pointed out that using tags shorter than 128 bits was a bad idea, since it makes forgeries more likely than the naïve attack where an attacker just

²⁷This is a simple consequence of the birthday bound (§4.3.1.) If encryptors can keep state, it's also possible to select nonces deterministically, *e.g.*, by starting from $N = 0$ and incrementing the nonce. As long as you avoid overflows and don't lose track of state, this approach can encrypt larger numbers of messages.

²⁸This limit was chosen in order to preserve a safety margin such that the probability of a repeated nonce is below 2^{-32} [113, §8.3].

guesses the correct authentication tag [119, 178]. NIST still allows short authentication tags under the bizarre justification that some applications can handle forged packets better [113, Appendix C]. I respectfully disagree: just use normal-length tags.

ChaCha-Poly1305

The ChaCha20-Poly1305 AEAD construction [242] is built from components developed by Daniel J. Bernstein. It’s conceptually quite similar to GCM-mode. Encryption is performed with the ChaCha20 stream cipher in CTR-mode. The associated data is padded, then concatenated with the ciphertext and string lengths, and hashed using the Poly1305 MAC function [48], which (internally) encrypts the hash to produce the authentication tag. There are a few differences: instead of a single hash key per key K , as in GCM, this construction derives a new hash key for every pair (K, N) . Moreover, the unique structure of the field Poly1305 uses means that some bits of the hash key are unused (“clamped” to zero.)

Encryption. The scheme uses the ChaCha20 cipher with a 256-bit key K , a 96-bit nonce N , and a 32-bit block counter. We’ll denote this function as $E_K()$ for consistency. Messages are assumed to be byte-aligned. The function $\text{zeropad}_{\text{bytes}}(S)$ outputs a sufficient number of zero padding bytes to pad the string S to a multiple of 16 bytes. The function $\text{len_bytes}_{64}()$ outputs the length of a string in bytes, as a 64-bit little-endian unsigned integer. Given (K, N, AD, M) , encryption proceeds as follows:

1. Set $H \leftarrow E_K(0^{32}, N)$ and parse the first 256 bits into two 128-bit strings r, s .
2. Set the 32-bit initial block counter $\text{ctr} \leftarrow 1$.
3. Compute C' by applying ChaCha20 to M under key K , nonce N , and initial block counter ctr .
4. Compute:

$$T \leftarrow \text{Poly1305}_{r,s}(\text{AD} \parallel \text{zeropad}_{\text{bytes}}(\text{AD}) \parallel C' \parallel \text{zeropad}_{\text{bytes}}(C') \parallel \text{len_bytes}_{64}(\text{AD}) \parallel \text{len_bytes}_{64}(C'))$$

5. Output $C \leftarrow (C' \parallel T)$.

An important difference from GCM is that the Poly1305 algorithm accepts a one-time key s that is internally used as a one-time pad to encrypt the hash. This isn’t really very different than GCM, but it explains why we do not need to explicitly encrypt the output of the function, and also also why we require a different s for every nonce.

Decryption. Decryption is fairly simple. Given (K, N, AD, C) , first parse C into (C', T) . Follow the steps of the encryption algorithm to derive the Poly1305 key (r, s) . Use Poly1305 to re-compute the tag T' and verify that $T \stackrel{?}{=} T'$. If not, output **error**. Otherwise, decrypt the ciphertext C' using nonce N and initial block counter 1, and output the resulting plaintext.

Security. ChaCha20-Poly1305 has been analyzed for security under the assumption that ChaCha20 is a PRF [261, 95]. As with GCM, this assumes that encryptors use it correctly:

5.3.6 Nonces and misuse-resistance

AEAD schemes require the encryptor to select a nonce prior to encrypting each message. As mentioned above, the same nonce should never be used twice with a given key. However, mistakes do happen! Since implementation errors are common, cryptographers have sought to minimize the damage caused by such a bug. The results are called *misuse-resistant* schemes [276, 144, 22].

The main goal of a misuse-resistant AEAD scheme is to prevent catastrophic failure in the event that nonces are re-used to encrypt messages. AEAD schemes can be classified into three categories:

1. **Not misuse-resistant.** Here, the re-use of a small number of nonces can have catastrophic effects on the security of all messages encrypted with that key. For example, both of the popular GCM and ChaCha-Poly1305 schemes leak the plaintext of messages encrypted with the same nonce; however (and even worse), they will leak the scheme’s internal authentication key if a small number of nonces are re-used. This exposes the whole scheme to tampering attacks [178].
2. **Partially misuse-resistant.** If nonces are re-used, confidentiality may be lost entirely for the specific messages where nonce re-use occurs. However, all other ciphertexts where nonces are *not* repeated will remain fully secure, and authenticity is maintained. CCM mode is an example of a mode that provides this property.
3. **Fully misuse-resistant.** When the same nonce is re-used to encrypt two messages M_1, M_2 the most information that an attacker will learn is whether $M_1 = M_2$. All other security properties of the scheme remain, including full confidentiality for M_1, M_2 when $M_1 \neq M_2$. An example of a scheme with this property is GCM-SIV [143, 144].

The results in the “partial” category typically stem from the use of a stream cipher to encrypt data. Here the re-use of a nonce to encrypt messages M_1, M_2 will produce ciphertexts of the form $M_1 \oplus KS, M_2 \oplus KS$ where KS is the specific keystream produced for that nonce. An attacker can XOR these values to obtain $M_1 \oplus M_2$, which may leak information about either message.

Synthetic IVs. Fully misuse-resistant schemes can avoid this problem by constructing a second “synthetic” IV that’s a function of the key, plaintext and nonce (as well as possibly the associated data.) The benefit is that even if the caller repeats a nonce, the synthetic IV should only repeat if *exactly* the same data is encrypted (including the associated data), which means that the attacker does not get the benefit of seeing different messages (or hash values) encrypted using the same IV. Schemes like GCM-SIV [143, 144] compute this IV very efficiently. Unfortunately there’s at least some cost to this safety benefit: computing the synthetic IV increases the cost of encrypting a message.

number of local cipher operations is $2^{\kappa+1}$, or 2^{57} for DES, and table storage is identical to the standard meet-in-the-middle attack. Even better, the process of building the table can be done offline, and the same table can be re-used to attack many different encryptors (using different keys), so at least part of the work can be performed once and used over and over again.

A fix for this is to use 3DES with three independent keys, also known as 3TDEA or EDE3. However, variants of these attacks still work on even this mode, though the security loss is more moderate. As a result, NIST standards classified even 168-bit EDE3 3DES as providing at most 112 bits of security (although nobody today should be using 3DES for any security purpose: the standard was withdrawn by NIST in 2024.)

5.4.3 Time-memory tradeoffs

5.4.4 Attacks on ciphers

Modern ciphers are evaluated for a variety of attacks, which are often very specific to the nature of the cipher itself. A proper discussion of cipher attacks would require more than this chapter. Quite probably more than this book. Here we will provide only the briefest touch

5.4.5 Attacks on insecure modes and constructions

Blockwise chosen plaintext attacks

5.4.6 Committing and “key-committing” encryption

Occasionally a real-world protocol requires both the properties of a commitment scheme *and* an encryption scheme. More concretely, a sender may wish to encrypt a message to some recipient, such that the recipient (decryptor) is assured that the resulting ciphertext can be opened to one (and only one) possible plaintext. The challenge here is that we must now also treat the sender as a potential adversary, and assume that it is *trying* to construct a ciphertext C that can be opened to different messages. This represents a stronger threat model than we apply to standard symmetric encryption, where we generally assume the encryptor is honest.

Even if we assume that the sender is adversarial, this doesn’t tell the full story. If we are sending a ciphertext to a recipient who already possesses a decryption key k , then most (non-randomized) decryption algorithms will produce at most one possible decryption (or an error.) For this attack to be a concern, the sender must have some input to the decryption process they can change *after* the ciphertext has been transmitted. Two common patterns show up in real systems: in the first example, decryption may require a nonce/IV that is not delivered as part of the ciphertext, but might be delivered or influenced after the ciphertext has been sent. In a second pattern, the decryptor may receive the *decryption key* separately

from the ciphertext.³⁶ Both of these examples raise the possibility that an adversarial sender might produce an ambiguous ciphertext that can be decrypted into different messages, and that the sender might be able to influence this decryption by supplying chosen values for either the nonce or decryption key. Schemes that prevent the attacker from executing this attack are called *committing* or *key-committing* encryption schemes.

Examples of non-committing schemes. Standard encryption schemes (including AEADs) are designed under the assumption that the recipient will always decrypt with the correct key (*i.e.*, the same key used to encrypt.) When this assumption is violated, these schemes may not provide any guarantees. To give an extreme (and unrealistic) example of the phenomenon, consider a ciphertext encrypted using a one-time pad: $C = M \oplus K$. Here it's trivial for the sender to derive a new key K' that “opens” C to any message M' of the same length, simply by computing $K' = C \oplus M'$. While we don't generally use one-time pads in production, some (unauthenticated) block cipher modes of operation can also be affected by either replacing the nonce/IV or the decryption key. Intuitively you might think that AEAD schemes are more resilient to this attack, since they typically include an *authentication tag* that should fail to verify under the wrong key or nonce. Unfortunately this assumption is only valid when the encryptor is honest. Surprisingly, if the encryptor is malicious, they can formulate ciphertexts for schemes such as GCM and Poly1305 that will correctly decrypt to different ciphertexts under different keys or nonces, due to the fact that the polynomial MACs used in these schemes can generate tags that validate with multiple different keys [141].

Can you actually do anything with this attack?. An obvious counterargument is that attacks on real-world schemes can't change the decrypted result very substantially, at least without destroying the message. Since keys and nonces are relatively short compared to the length of larger encrypted messages (such as files), replacing these values only provides a limited amount of flexibility to the malicious encryptor. However, even small changes to a plaintext can do surprising amounts of damage. For example, some recent works have demonstrated that it's possible to generate *polyglot binaries* that effectively contain two different payloads (even files of different types!) where a small and surgical change to just the right portion of the file header will cause a parser to “choose” one of the two payloads and ignore the other [103, 12]. By carefully structuring the file and the encryption parameters, a malicious encryptor can encrypt such a file, then later “open” the ciphertext to the payload of their choosing.

Committing encryption AEAD modes. Several research papers have proposed fixes to the committing problem. At a high level, the typical solution is to extend an existing AEAD

³⁶A popular example of this use-case occurs in *message franking schemes* used in some end-to-end encryption systems, see §12.1.1.

by attaching a commitment as part of the ciphertext: here the commitment is computed over various parameters such as the key K , the nonce/IV, any associated data, and the authentication tag used the scheme. Two concrete schemes are the CTX extension proposed by Chan and Rogaway [81] and an approach due to Bellare and Hoang [40]. Some of these ideas are being considered for future encryption standards. My general advice in this area is that you should remember that this attack only applies in very specialized cases where you don't trust the encryptor. Hence you should address it on a case-by-case basis and try to use existing standards in places where you're sure this won't be an issue for your system.

DRAFT