8. Message Authentication Codes (MACs)

8. Message Authentication Codes (MACs)

8.1. Integrity and Authenticity

When building cryptographic schemes that guarantee integrity and authentication, the threat we're concerned about is adversaries who send messages pretending to be from a legitimate participant (*spoofing*) or who modify the contents of a message sent by a legitimate participant (*tampering*). To address these threats, we will introduce cryptographic schemes that enable the recipient to detect spoofing and tampering.

In this section, we will define *message authentication codes (MACs)* and show how they guarantee integrity and authenticity. Because MACs are a symmetric-key cryptographic primitive, in this section we can assume that Alice and Bob share a secret key that is not known to anyone else. Later we will see how Alice and Bob might securely exchange a shared secret key over an insecure communication channel, but for now you can assume that only Alice and Bob know the value of the secret key.

8.2. MAC: Definition

A MAC is a keyed checksum of the message that is sent along with the message. It takes in a fixed-length secret key and an arbitrary-length message, and outputs a fixed-length checksum. A secure MAC has the property that any change to the message will render the checksum invalid.

Formally, the MAC on a message M is a value $F(K, M)$ computed from K and M; the value $F(K, M)$ is called the *tag* for M or the MAC of M. Typically, we might use a 128-bit key K and 128-bit tags.

When Alice wants to send a message with integrity and authentication, she first computes a MAC on the message $T = F(K, M)$. She sends the message and the MAC $\langle M, T \rangle$ to Bob. When Bob receives $\langle M, T \rangle$, Bob will recompute $F(K, M)$ using the M he received and check that it matches the MAC T he received. If it matches, Bob will accept the message M as valid, authentic, and untampered; if $F(K, M) \neq T$, Bob will ignore the message M and presume that some tampering or message corruption has occurred.

Note that MACs must be deterministic for correctness–when Alice calculates $T = F(K, M)$ and sends $\langle M, T \rangle$ to Alice, Bob should get the same result when he calculates $F(K, M)$ with the same K and M.

MACs can be used for more than just communication security. For instance, suppose we want to store files on a removable USB flash drive, which we occasionally share with our friends. To protect against tampering with the files on our flash drive, our machine could generate a secret key and store a MAC of each file somewhere on the flash drive. When our machine reads the file, it could check that the MAC is valid before using the file contents. In a sense, this is a case where we are "communicating" to a "future version of ourselves," so security for stored data can be viewed as a variant of communication security.

8.3. MAC: Security properties

Given a secure MAC algorithm F, if the attacker replaces M by some other message M', then the tag will almost certainly[1] no longer be valid: in particular, $F(K, M) \neq F(K, M')$ for any $M' \neq M$.

[1]Strictly speaking, there is a very small chance that the tag for M will also be a valid tag for M'. However, if we choose tags to be long enough—say, 128 bits—and if the MAC algorithm is secure, the chances of this happening should be about $1/2^{128}$, which is small enough that it can be safely ignored.

More generally, there will be no way for the adversary to modify the message and then make a corresponding modification to the tag to trick Bob into accepting the modified message: given M and $T = F(K, M)$, an attacker who does not know the key K should be unable to find a different message M' and a tag T' such that T' is a valid tag on M' (i.e., such that $T' = F(K, M')$). Secure MACs are designed to ensure that even small changes to the message make unpredictable changes to the tag, so that the adversary cannot guess the correct tag for their malicious message M'.

Recall that MACs are deterministic–if Alice calculates $F(K, M)$ twice on the same message M, she will get the same MAC twice. This means that an attacker who sees a pair $M, F(K, M)$ will know a valid MAC for the message M. However, if the MAC is secure, the attacker should be unable to create valid MACs for messages that they have never seen before.

More generally, secure MACs are designed to be secure against known-plaintext attacks. For instance, suppose an adversary Eve eavesdrops on Alice's communications and observes a number of messages and their corresponding tags: $\langle M_1, T_1\rangle, \langle M_2, T_2\rangle, \ldots, \langle M_n, T_n\rangle$, where $T_i = F(K, M_i)$. Then Eve has no hope of finding some new message M' (such that $M' \notin \{M_1, \ldots, M_n\}$) and a corresponding value T' such that T' is the correct tag on M' (i.e., such that $T' = F(K, M')$). The same is true even if Eve was able to choose the M_i's. In other words, even though Eve may know some valid MACs $\langle M_n, T_n\rangle$, she still cannot generate valid MACs for messages she has never seen before.

Here is a formal security definition that captures both properties described above. We imagine a game played between Georgia (the adversary) and Reginald (the referee). Initially, Reginald picks a random key K, which will be used for all subsequent rounds of the game. In each round of the game, Georgia may query Reginald with one of two kinds of queries:

- **Generation query:** Georgia may specify a message M_i and ask for the tag for M_i. Reginald will respond with $T_i = F(K, M_i)$.
- **Verification query:** Alternatively, Georgia may specify a pair of values $\langle M_i, T_i\rangle$ and ask Reginald whether T_i is a valid tag on M_i. Reginald checks whether $T_i \stackrel{?}{=} F(K, M_i)$ and responds "Yes" or "No" accordingly.

Georgia is allowed to repeatedly interact with Reginald in this way. Georgia wins if she ever asks Reginald a verification query $\langle M_n, T_n\rangle$ where Reginald responds "Yes", and where M_n did not appear in any previous generation query to Reginald. In this case, we say that Georgia has successfully forged a tag. If Georgia can successfully forge, then the MAC algorithm is insecure. Otherwise, if there is no strategy that allows Georgia to forge (given a generous allotment of computation time and any reasonable number of rounds of the game), then we say that the MAC algorithm is secure.

This game captures the idea that Georgia the Forger can try to observe the MAC tag on a bunch of messages, but this won't help her forge a valid tag on any new message. In fact, even if Georgia carefully selects a bunch of chosen messages and gets Alice to transmit those messages (i.e., she gets Alice to compute the MAC on those messages with her key, and then transmit those MAC tags), it still won't help Georgia forge a valid tag on any new message. Thus, MACs provide security against chosen-plaintext/ciphertext attacks, the strongest threat model.

8.4. AES-EMAC

How do we build secure MACs?

There are a number of schemes out there, but one good one is AES-CMAC, an algorithm standardized by NIST. Instead of showing you AES-CMAC, we'll look at a related algorithm called AES-EMAC. AES-EMAC is a slightly simplified version of AES-CMAC that retains its essential character but differs in a few details.

In AES-EMAC, the key K is 256 bits, viewed as a pair of 128-bit AES keys: $K = \langle K_1, K_2\rangle$. The message M is decomposed into a sequence of 128-bit blocks: $M = P_1 \| P_2 \| \ldots \| P_n$. We set $S_0 = 0$ and compute

$$S_i = \mathrm{AES}_{K_1}(S_{i-1} \oplus P_i), \qquad \text{for } i = 1, 2, \ldots, n.$$

Finally we compute $T = \text{AES}_{K_2}(S_n)$; T is the tag for message M. Here is what it looks like:

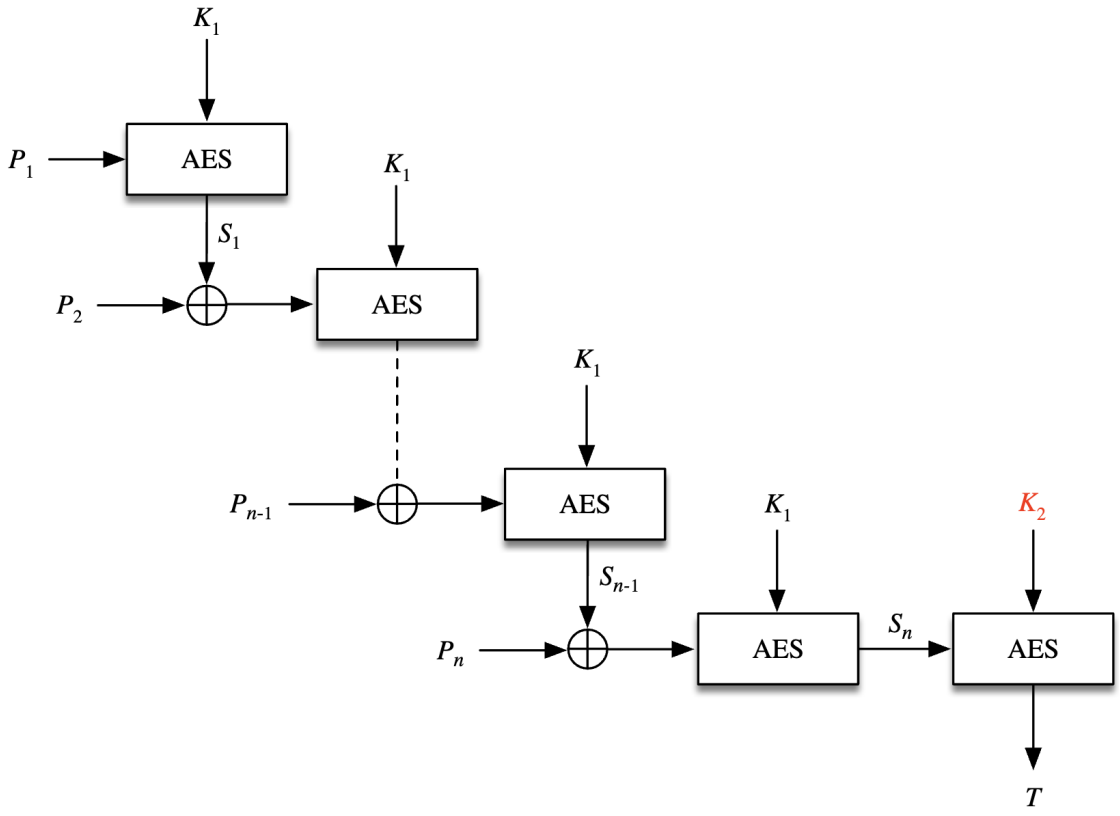

Figure 1: AES-EMAC block diagram, with K2 highlighted as the final encryption step

Assuming AES is a secure block cipher, this scheme is provably secure, using the unforgeability definition and security game described in the previous section. An attacker cannot forge a valid AES-EMAC for a message they haven't seen before, even if they are allowed to query for MACs of other messages.

8.5. HMAC

One of the best MAC constructions available is the HMAC, or Hash Message Authentication Code, which uses the cryptographic properties of a cryptographic hash function to construct a secure MAC algorithm.

HMAC is an excellent construction because it combines the benefits of both a MAC and the underlying hash. Without the key, the tag does not leak information about the message. Even with the key, it is computationally intractable to reconstruct the message from the hash output.

There are several specific implementations of HMAC that use different cryptographic hash functions: for example, HMAC_SHA256 uses SHA256 as the underlying hash, while HMAC_SHA3_256 uses SHA3 in 256 bit mode as the underlying hash function. The choice of underlying hash depends on the application. For example, if we are using HMACs with a block cipher, we would want to choose an HMAC whose output is twice the length of the keys used for the associated block cipher, so if we are encrypting using AES_192 we should use HMAC_SHA_384 or HMAC_SHA3_384.

The output of HMAC is the same number of bits as the underlying hash function, so in both of these implementations it would be 256 bits of output. In this section, we'll denote the number of bits in the hash output as n.

To construct the HMAC algorithm, we first start with a more general version, NMAC:

$\text{NMAC}(K_1, K_2, M) = H(K_1 \| H(K_2 \| M))$

In words, NMAC concatenates K_2 and M, hashes the result, concatenates the result with K_1, and then hashes that result.

Note that NMAC takes two keys, K_1 and K_2, both of length n (the length of the hash output). If the underlying hash function H is cryptographic and K_1 and K_2 are unrelated[2], then NMAC is provably secure.

HMAC is a more specific version of NMAC that only requires one key instead of two unrelated keys:

$\text{HMAC}(M, K) = H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$

The HMAC algorithm actually supports a variable-length key K. However, NMAC uses K_1 and K_2 that are the same length as the hash output n, so we first transform K to be length n. If K is shorter than n bits, we can pad K with zeros until it is n bits. If K is longer than n bits, we can hash K to make it n bits. The transformed n-bit version of K is now denoted as K'.

Next, we derive two unrelated keys from K'. It turns out that XORing K' with two different pads is sufficient to satisfy the definition of "unrelated" used in the NMAC security proof. The HMAC algorithm uses two hardcoded pads, *opad* (outer pad) and *ipad* (inner pad), to generate two unrelated keys from a single key. The first key is $K_1 = K' \oplus opad$, and the second key is $K_2 = K' \oplus ipad$. *opad* is the byte `0x5c` repeated until it reaches n bits. Similarly, *ipad* is the byte `0x36` repeated until it reaches n bits.[3]

In words, HMAC takes the key and pads it or hashes it to length n. Then, HMAC takes the resulting modified key, XORs it with the ipad, concatenates the message, and hashes the resulting string. Next, HMAC takes the modified key, XORs it with the opad, and then concatenates it to the previous hash. Hash this final string to get the result.

Because NMAC is provably secure, and HMAC is a special case of NMAC that generates the two unrelated keys from one key, HMAC is also provably secure. This proof assumes that the underlying hash is a secure cryptographic hash, which means if you can find a way to break HMAC (forge a valid HMAC without knowing the key), then you have also broken the underlying cryptographic hash.

Because of the properties of a cryptographic hash, if you change just a single bit in either the message or the key, the output will be a completely different, unpredictable value. Someone who doesn't know the key won't be able to generate tags for arbitrary messages. In fact, they can't even distinguish the tag for a message from a random value of the same length.

HMAC is also very efficient. The inner hash function call only needs to hash the bits of the message, plus n bits, and the outer hash function call only needs to hash $2n$ bits.

8.6. MACs are not confidential

A MAC does not guarantee confidentiality on the message M to which it is applied. In the examples above, Alice and Bob have been exchanging non-encrypted plaintext messages with MACs attached to each message. The MACs provide integrity and authenticity, but they do nothing to hide the contents of the actual message. In general, MACs have no confidentiality guarantees–given $F(K, M)$, there is no guarantee that the attacker cannot learn something about M.

As an example, we can construct a valid MAC that guarantees integrity but does not guarantee confidentiality. Consider the MAC function F' defined as $F'(K, M) = F(K, M) \| M$. In words, F' contains a valid MAC of the message, concatenated with the message plaintext. Assuming F is a valid MAC, then F' is also valid MAC. An attacker who doesn't know K won't be able to generate $F'(K, M')$ for the attacker's message M', because they won't be able to generate $F(K, M')$, which is part of $F'(K, M')$. However, F' does not provide any confidentiality on the message–in fact, it leaks the entire message!

[2]The formal definition of "unrelated" is out of scope for these notes. See this paper to learn more.

[3]The security proof for HMAC just required that ipad and opad be different by at least one bit but, showing the paranoia of cryptography engineers, the designers of HMAC chose to make them very different.

There is no notion of "reversing" or "decrypting" a MAC, because both Alice and Bob use the same algorithm to generate MACs. However, there is nothing that says a MAC algorithm can't be reversed if you know the key. For example, with AES-MAC it is clear that if the message is a single block, you can run the algorithm in reverse to go from the tag to the message. Depending on the particular MAC algorithm, this notion of reversing a MAC might also lead to leakage of the original message.

There are some MAC algorithms that don't leak information about the message because of the nature of the underlying implementation. For example, if the algorithm directly applies a block cipher, the block cipher has the property that it does not leak information about the plaintext. Similarly, HMAC does not leak information about the message, since it maintains the properties of the cryptographic hash function.

In practice, we usually want to guarantee confidentiality in addition to integrity and authenticity. Next we will see how we can combine encryption schemes with MACs to achieve this.

8.7. Authenticated Encryption

An *authenticated encryption* scheme is a scheme that simultaneously guarantees confidentiality and integrity on a message. As you might expect, symmetric-key authenticated encryption modes usually combine a block cipher mode (to guarantee confidentiality) and a MAC (to guarantee integrity and authenticity).

Suppose we have an IND-CPA secure encryption scheme Enc that guarantees confidentiality, and an unforgeable MAC scheme MAC that guarantees integrity and authenticity. There are two main approaches to authenticated encryption: encrypt-then-MAC and MAC-then-encrypt.

In the *encrypt-then-MAC* approach, we first encrypt the plaintext, and then produce a MAC over the ciphertext. In other words, we send the two values $\langle \mathsf{Enc}_{K_1}(M), \mathsf{MAC}_{K_2}(\mathsf{Enc}_{K_1}(M)) \rangle$. This approach guarantees *ciphertext integrity*–an attacker who tampers with the ciphertext will be detected by the MAC on the ciphertext. This means that we can detect that the attacker has tampered with the message without decrypting the modified ciphertext. Additionally, the original message is kept confidential since neither value leaks information about the plaintext. The MAC value might leak information about the ciphertext, but that's fine; we already know that the ciphertext doesn't leak anything about the plaintext.

In the *MAC-then-encrypt* approach, we first MAC the message, and then encrypt the message and the MAC together. In other words, we send the value $\mathsf{Enc}_{K_1}(M \| \mathsf{MAC}_{K_2}(M))$. Although both the message and the MAC are kept confidential, this approach does not have ciphertext integrity, since only the original message was tagged. This means that we'll only detect if the message is tampered after we decrypt it. This may not be desirable in some applications, because you would be running the decryption algorithm on arbitrary attacker inputs.

Although both approaches are theoretically secure if applied correctly, in practice, the MAC-then-Encrypt approach has been attacked through side channel vectors. In a side channel attack, improper usage of a cryptographic scheme causes some information to leak through some other means besides the algorithm itself, such as the amount of computation time taken or the error messages returned. One example of this attack was a padding oracle attack against a particular TLS implementation using the MAC-then-encrypt approach. Because of the possibility of such attacks, encrypt-then-MAC is generally the better approach.

In both approaches, the encryption and MAC functions should use different keys, because using the same key in an authenticated encryption scheme makes the scheme vulnerable to a large category of potential attacks. These attacks take advantage of the fact that two different algorithms are called with the same key, as well as the properties of the particular encryption and MAC algorithms, to potentially leak information about the original message. The easiest way to avoid this category of attacks is to simply use different keys for the encryption and MAC functions.

8.8. AEAD Encryption Modes

There are also some special block cipher operation modes, known as AEAD (Authenticated Encryption with Additional Data) that, in addition to providing confidentiality like other appropriate block cipher modes, also provide integrity/authenticity.

The “additional data” component means that the integrity is provided not just over the encrypted portions of the message but some additional unencrypted data. For example, if Alice wants to send a message to Bob, she may want to include that the message is "From Alice to Bob" in plaintext (for the benefit of the system that routes the message from Alice to Bob) but also include it in the set of data protected by the authentication.

While powerful, using these modes improperly will lead to catastrophic failure in security, since a mistake will lead to a loss of both confidentiality and integrity at the same time.

One such mode is called AES-GCM (Galois Counter Mode). The specifics are out of scope for these notes, but at a high level, AES-GCM is a stream cipher that operates similarly to AES-CTR (counter) mode. The security properties of AES-GCM are also similar to CTR–in particular, IV reuse also destroys the security of AES-GCM. Since the built-in MAC in AES-GCM is also a function of the CTR mode encryption, improper use of AES-GCM causes loss of both confidentiality and integrity.

Some other modes include CCM mode, CWC mode, and OCB mode, but these are out of scope for these notes.