Message Authentication Codes

6.1600 Course Staff: Henry Corrigan-Gibbs, Yael Kalai, Ben Kettle (TA), Nickolai Zeldovich

Fall 2022

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

So far, we have talked about authenticating *people* and authenticating *files*. In this section, we will discuss authenticating *communication*. If we have two parties that are communicating over the network, we want some way to guarantee to each party that the message they received really came from the other party and was not tampered with along the way.

At a first glance, this seems impossible. If there is some eavesdropper Eve in between the two parties, they can just replace the message with one of their own choosing and the other party will have no idea. To make this possible, we need to relax the scenario a bit and add an assumption—that the two parties share some secret key k.

With this shared key k between the two parties, our goal will be to add some "tag" onto the message that validates its authenticity. Necessarily, this tag will be a function of this shared key k. If this were not the case, the eavesdropper would be able to compute a valid tag herself—the secret k is the only information in this scenario that Eve does not know.

1 *Defining message authentication codes*

Syntax. A message authentication code (MAC) over key space $\mathcal{K}$, message space $\mathcal{M}$, and tag space $\mathcal{T}$ is an efficient algorithm $\mathsf{MAC}\colon \mathcal{K} \times \mathcal{M} \to \mathcal{T}$. In order for a MAC to be useful, it must be *secure*, in the following sense. We first give the definition and then explain why it is a useful one:

Definition 1.1 (MAC Security: Existentially unforgeability against adaptive chosen message attacks)**.** A MAC MAC over key space $\mathcal{K}$ and message space $\mathcal{M}$ is secure (existentially unforgeable against adaptive chosen message attacks) if any poly-time adversary $\mathcal{A}$ wins the following game with at most negligible probability:

In practice, "a poly-time adversary" means "any real-life adversary". But we need to place some mathematical bound on real-life to make the proofs work out.

- The challenger samples a MAC key $k \xleftarrow{R} \mathcal{K}$.
- For $i = 1, 2, \ldots$ (polynomially many times)
  - The adversary sends any message $m_i \in \mathcal{M}$ to the challenger
  - The challenger responds with $\mathsf{MAC}(k, m_i)$.
- The adversary sends the challenger a message-tag pair (m^*, t^*).
- The adversary wins the game if $\mathsf{MAC}(k, m^*) = t^*$ and $m^* \notin \{m_1, m_2, \ldots, m_n\}$.

1.1 Intuition for the security definition

To formulate our security notion, we need to define the adversary's goal and the adversary's power.

The adversary's goal in this definition is to compute a valid MAC of *any* message $m \in \mathcal{M}$ of its choice. It's not entirely obvious why we care about the adversary producing a valid MAC on *any* message: "If the adversary MACs a message that is jibberish, they are unlikely to be able to do any harm with it," you might think. But there will certainly be applications that authenticate messages that violate whatever definition of "non-jibberish" we define. So allowing the adversary to forge a MAC tag on any message makes the definition as broadly applicable as possible.

A subtlety of this definition is that, even if the MAC scheme is secure under this definition, it is possible for an adversary, given a valid message-tag pair (m, t) to produce a second valid message-tag pair (m, t') on the same message without knowing the secret key.

As far as the adversary's power goes: we, as usual in cryptography, restrict the adversary to be efficient (i.e., to run in polynomial time). But in the MAC security game we also allow the adversary to obtain MAC tags on messages of its choice. This captures the reality that in many systems, an adversary can trick an honest system into MACing adversarial messages. For example, if an email-backup system MACs every email that a user receives, an adversary may be able to obtain MAC tags on messages of its choice by sending emails to the backup system.

This has some interesting implications—importantly, the adversary can store these messages along with their MAC and replay them later.

1.2 MACs require pseudorandomness

The fact that it is even possible to construct a MAC seems a bit surprising—in effect, for a MAC to satisfy the definition, the tag has to effectively be random. But the only "randomness" that we have is the key k—to generate tags for arbitrarily many messages, we need much more randomness than one key's worth. This seems impossible. How can we generate a large number of random-looking tags from only a single short random key?

We get ourselves out of this conundrum by observing that the adversary must be an *efficient* algorithm. So while we cannot generate a large number of truly random bits from a short key, we can—under appropriate and reasonable cryptographic assumptions—generate a large number of bits that *look* truly random from the perspective of any efficient algorithm. We call these bits *pseudorandom*.

This surprising and powerful idea leads us to our next cryptographic primitive...

2 Pseudorandom Functions

A pseudorandom function is defined over a keyspace $\mathcal{K}$, and input spacei $\mathcal{X}$ and output space $\mathcal{Y}$. To be useful a pseudorandom function must satisfy the following security definition:

Definition 2.1 (Pseudorandom Function, PRF)**.** A function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a pseudorandom function if all efficient algorithms $\mathcal{A}$ win the following game with probability $\frac{1}{2}$ + "negligible":

- The challenger samples a random bit $b \leftarrow \{0,1\}$ and a key $k \xleftarrow{R} \mathcal{K}$.
- If $b = 0$, the challenger sets $f(\cdot) := F(k, \cdot)$.
- If $b = 1$, the challenger sets $f(\cdot) \xleftarrow{R} \mathsf{Funs}[\mathcal{X}, \mathcal{Y}]$.
- Then for $i = 1, 2, \ldots$ (polynomially many times):
  - The adversary sends the challenger a values $x_i \in \mathcal{X}$.
  - The challenger responds with $y_i \leftarrow f(x_i) \in \mathcal{Y}$.
- The adversary outputs a guess $\hat{b}$ at the bit b.
- The adversary wins if $b = \hat{b}$.

Here, Funs is the set of all functions from $\mathcal{X}$ to $\mathcal{Y}$.

First, the challenger will sample a random $b \leftarrow \{0,1\}$ and a key $k \leftarrow \mathcal{K}$.

The adversary can trivially win this game with probability $\frac{1}{2}$ by just guessing the bit b at random. This definition asserts that no efficient adversary can do much better than that.

If we have such a pseudorandom function F, we could easily construct a MAC—we can just use the message as the input to the pseudorandom function along with the key: $\mathsf{MAC}(k, m) := F(k, m)$.

2.1 Constructing pseudorandom functions from one-wayness

It is not at all obvious that pseudorandom functions should exist at all! They seem like a very magical primitive indeed.

One surprising fact is that if there exists *any* function that is "hard to invert," in a sense we will define, then pseudorandom functions exist. For example, if you believe that factoring large numbers is difficult (as many people do), then pseudorandom functions exist.

In particular the following definition captures the notion of a function that is hard to invert:

Definition 2.2 (One-Way Function)**.** A function $f\colon \mathcal{X} \to \mathcal{Y}$ is a *one-way function* if for all efficient adversaries $\mathcal{A}$,

$$\Pr[f(\mathcal{A}(f(x))) = f(x)\colon x \xleftarrow{R} \mathcal{X}] \le \text{"negligible"}.$$

Having defined one-way functions, we now have the following surprising and non-obvious result:

Notice that if $\mathsf{P} = \mathsf{NP}$, one-way functions do not exist, and therefore psuedorandom functions do not exist.

Theorem 2.3. *Psueodorandom functions exist if and only if one-way functions exist.*

In practice, we assume that:

- the function $f(x) := \text{SHA256}(x)$ is a one-way function where the domain is the set of 256-bit strings,
- the function $f(x) := \text{AES}(x, 0^{128})$ is a one-way function, where the domain is the set of 128-bit strings, and
- the function $f(x) := 2^x \bmod p$ is a one-way function on domain $\{1, \dots, p\}$, for a sufficiently large prime p.

2.2 *Pseudorandom functions in practice*

In practice, we use the Advanced Encryption Standard (AES) as a pseudorandom function. The AES function on key length $\kappa \in \{128, 192, 256\}$ has the type signature $\text{AES} : \{0,1\}^\kappa \times \{0,1\}^{128} \to \{0,1\}^{128}$. That is, it takes a 128-bit input and generates a 128-bit output.

We don't have any mathematical proof that AES is a pseudorandom function. However, it has undergone a tremendous amount of cryptanalysis and the best attacks on AES are only marginally better than the obvious brute-force attacks.

3 *From pseudorandom functions to MACs*

MACs for short messages. Using AES as a pseudorandom function on a 128-bit domain, we can build a MAC for 128-bit messages as described above : $\mathsf{MAC}(k, m) := \mathsf{AES}_k(m)$. However, since AES takes only 128 bits as input, using AES directly, we can only authenticate 128-bit messages.

Insecure ways to construct a MAC for long messages. A bad way to construct a MAC for long messages from a pseudorandom function F for 128-bit messages is just to chop our message m up into 128-bit blocks $m = (m_1, m_2, \dots)$ and MAC each block separately. Our tag, then, would look something like $(F(k, m_1), F(k, m_1))$. However, there is a problem! Given the tag $t = (t_1, t_2)$ for a message $m = (m_1, m_2)$, we can easily generate a valid tag $t' = (t_2, t_1)$ for a different message $m' = (m_2, m_1)$.

MACs for long messages: The easy way. If we have a pseudorandom function F with an input space of 256-bits, we can construct a MAC on arbitrary-length messages using the "hash-and-sign" paradigm. In particular, we use a collision-resistant hash function $H\colon \{0,1\}^* \to \{0,1\}^{256}$ (**??**) and we define the MAC on message space $\{0,1\}^*$ as:

$$\mathsf{MAC}(k, m) := F(k, H(m)).$$

Notice that we cannot use AES as the pseudorandom function F in this construction, since AES only takes a 128-bit input. In this case, we would need a collision-resistant hash function $H\colon \{0,1\}^* \to \{0,1\}^{128}$, but it is always possible to find collisions in hash functions with 128-bit output in time 2^{64}. So such a MAC can never be secure against attackers running in time 2^{64}.

In practice, we typically do not construct MACs in this way because collision-resistant hash functions are typically more expensive to compute (per bit of input) than pseudorandom functions, such as AES.

3.1 MACs for long messages: Cipher-Block Chaining MAC

A common and secure way to construct a MAC for long messages from a MAC for short messages is to *chain* the output of each of these calls to the pseudorandom function. Given our chopped message $(m_1, m_2, \ldots, m_n)$, we will generate $t_1 = F(k, m_1)$ as before. When generating t_2, we will first XOR t_1 into the input: $t_2 = F(k, m_2 \oplus t_1)$. This continues until the end of the message, at which point have a tag t_n. Finally, we apply the PRF *with a different key* k' to the value t_n and output this tag $t \leftarrow F(k', t_n)$. This construction is called CBC-MAC or CMAC.

Applying the PRF to the last block using an independent random key is important. If we do not use a new key, an adversary can mount a length-extension attack. That is, if the adversary asks for $t = \mathsf{MAC}(k, m_1)$ and $t' = \mathsf{MAC}(k, t)$, t' is also a valid key for the original message with two zero blocks attached $\mathsf{MAC}(k, m_1 \| 0 \| 0)$. The chain of AES applications becomes equivalent, since zero blocks are equivalent to skipping the XOR and adding AES applications.

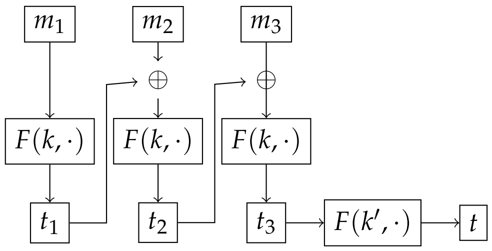

Figure 1: The CBC-MAC construction.

CBC-MAC is going out of favor for two reasons:

1. It is impossible to parallelize the MAC computation: the chaining procedure is inherently sequential so you cannot speed it up, even if you have a computer with many CPU cores.
2. Computing the MAC requires one PRF invocation *per block* of the message. There are even faster MACs that require only one PRF invocation *per message* total, plus a number of fast "non-cryptographic" operations per message block. These MACs can be faster than CBC-MAC on some processors. The GMAC construction we will see next is one example.

3.2 A parallelizable MAC: Carter-Wegman MAC

We now describe a different way to authenticate long messages. This MAC scheme is parallelizable and also requires only one single PRF invocation per message authenticated (independent of the message length). The construction is named the Carter-Wegman MAC, after its inventors.[1] Modern encryption schemes, including AES-GCM (**??**) use a Carter-Wegman-style MAC as a key ingredient.

[1] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3), 1981

For this construction, we will use the notation $\mathbb{Z}_p$ to indicate the set of integers modulo p with addition and multiplication modulo p. So $x + y \in \mathbb{Z}_p$ means that we add x and y as integers and reduce the result modulo p. Typically, we will think of p as a prime—of 64 bits, for example.

The MAC uses a fixed a prime number p as a parameter, where $p \approx 2^n$ on security parameter n. The MAC uses a pseudorandom function $F\colon \mathcal{K} \times \mathbb{Z}_p \to \mathbb{Z}_p$.

So in practice, we take $p \approx 2^{128}$ for 128-bit security.

Here, the input space of the pseudorandom function F is the set of integers in $\{0, \dots, p-1\}$. Given a pseudorandom function on bitstrings, it is indeed possible to construct one that operates on numbers in $\mathbb{Z}_p$ like this by interpreting each number as a bitstring.

The keyspace for the MAC is $\mathcal{K}$, so the MAC key consists of a key for the pseudorandom function. The message space for the MAC is $\mathcal{M} = \mathbb{Z}_p^{\leq L}$, the set of vectors of integers of $\mathbb{Z}_p$ elements of length at most L where $L \ll p$. Here, assume that the message vector has length at least 1.

One other difference is that this MAC construction is *randomized*. So there are now two algorithms:

- $\mathsf{MAC.Sign}(k, m) \to t$, which takes as input a key k and message m and outputs a MAC tag t, and
- $\mathsf{MAC.Verify}(k, m, t) \to \{0, 1\}$, which takes as input a key k, message m, tag t, and outputs an accept/reject bit.

The security definition here is essentially the same as for deterministic MACs, except that we use different algorithms to generate and verify the MAC tags.

The Carter-Wegman MAC construction is then:

$\mathsf{MAC.Sign}(k, m \in \mathbb{Z}_p^{\leq L}) \to t$.

- Compute $v \leftarrow F(k, 0) \in \mathbb{Z}_p$.
- Parse the message into chunks as $(m_1, \dots, m_\ell) \leftarrow m \in \mathbb{Z}_p^\ell$.
- Compute $M(v) \leftarrow m_1 v + m_2 v^2 + m_3 v^3 \cdots + m_\ell v^\ell \in \mathbb{Z}_p$.
- Sample a nonce $r \xleftarrow{R} \mathbb{Z}_p$.
- Output $t \leftarrow \big(r, F(k, r) + M(v)\big) \in \mathbb{Z}_p^2$ as the MAC tag.

Essentially we are viewing the blocks of the message m as coefficients of a degree-t polynomial $M(\cdot)$. We then evaluate this polynomial at the secret point v determined by the MAC key.

$\mathsf{MAC.Verify}(k, m, t) \to \{0, 1\}$.

- Compute $v \leftarrow F(k, 0) \in \mathbb{Z}_p$.
- Parse the message into chunks as $(m_1, \dots, m_\ell) \leftarrow m \in \mathbb{Z}_p^\ell$.
- Compute $M(v) \leftarrow m_1 v + m_2 v^2 + m_3 v^3 + \cdots + m_\ell v^\ell \in \mathbb{Z}_p$.
- Parse the tag $(r, z) \leftarrow t \in \mathbb{Z}_p^2$.
- Output "1" if and only if $z - F(k, r) = M(v)$.

Security intuition. The security argument here goes as follows:

For a detailed treatment of Carter-Wegman security see Boneh and Shoup's textbook, *A Graduate Course in Applied Cryptography*, Section 7.4.

- First, we appeal to the PRF security property to argue that we can replace the values $F(k_F, r)$ used to generate the tags with truly random values.

- Next, we show that as long as the MAC.Sign algorithm never samples the same nonce r twice, the masking values $F(k,r)$ are independent random values that complete hide the values $M(v)$. So, the adversary learns no information on the secret point v by making MAC queries.
- Now, say that the adversary finds a forged message-tag pair (m^*, t^*). There are two cases:
  - Either the forgery uses a fresh random nonce r^* that did not appear as the response to any of the adversary's MAC queries. In this case, the forgery is only valid with probability $1/p$.
  - Alternatively, the forger could use a random nonce r^* that is equal to the nonce r returned from one of the adversary's MAC queries. In this case, we have the following relations, where message m polynomial M was the message the adversary queried of the challenger:

$$\begin{aligned} F(k,r) &= M(v) - z \\ F(k,r) &= M^*(v) - z^* \\ 0 &= \big(M(v) - M^*(v)\big) + (z - z^*). \end{aligned}$$

    Since $m \neq m^*$, we know $z \neq z^*$. So $(M(\cdot) - M^*(\cdot)) + (z^* - z)$ is a non-zero polynomial of degree at most t. Since such a polynomial can have at most $\ell \leq L$ zeros in $\mathbb{Z}_p$, and since the adversaries view is independent of the evaluation point $v \in \mathbb{Z}_p$, the probability that the adversary's forgery is valid is at most ℓ/p.

  In either case, the adversary's probability of forging is $O(L)/p = \text{poly}(\lambda) \cdot \text{negl}(\lambda) = \text{negl}(\lambda)$ on security parameter λ.

References

[1] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3), 1981.