CMSC 28400 Introduction to Cryptography **Autumn 2021**

Notes #11: Message Authentication

Instructor: David Cash

So far we've been concerned with secrecy of messages, and have considered ciphers and randomized encryption with various secrecy security goals. These notes consider a different security goal: *authenticity of messages*. Intuitively, we'd like tools that help prevent adversaries from changing or inserting messages on communications links. A typical setting is a compromised connection on the internet, where an untrusted router controls all of the messages being sent. Another setting is a cell connection, where an adversary with a powerful antenna may interfere with traffic between a device and a cell tower. While in those settings secrecy is also a goal, we're going to focus exclusively on authenticity first, and then return to how it may combined with secrecy.

11.0.1 Message Authentication Codes (MACs)

Let us start with an abstract setting. A sender and receiver are connected via a channel that is controlled by an adversary $\mathcal{A}$ who may change their messages (See Figure 11.1). If the endpoints don't share *something*, then there's not much they can do: The bits arriving at the receiver could just as well have come from the sender as the adversary. So we'll assume, like we did with encryption, that the two endpoints share a key k, and that they'll try use k to somehow stop $\mathcal{A}$, who doesn't know k.

A first idea might be to encrypt m using a cipher or randomized encryption scheme, and send c instead. The receiver could then decrypt and consider the output as the received message. (See Figure 11.2.) Assuming encryption is secure (say many-time CPA secure), does this ensure that when $\hat{m}$ is received, it was really sent by the sender? The answer is "not always" (at least). This is because *malleability attacks* may be possible, where an adversary can change the contents of a ciphertext even when it does not know the key. Whether or not the change is meaningful is a separate issue, but we can say with confidence that sometimes $\mathcal{A}$ may sneak in a $\hat{m}$ that wasn't actually sent.

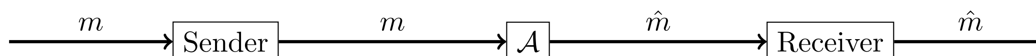

Figure 11.1: An unprotected channel.

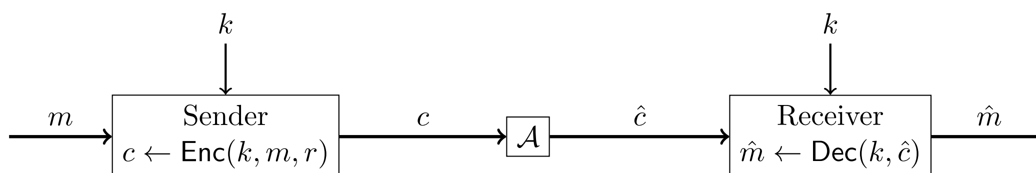

Figure 11.2: An attempt at authentication using encryption.

Instead of trying to use encryption schemes for authentication, we're going to define a new primitive, called a *message authentication code* (MAC) that is purpose-built. The term MAC is very standard, to the point where we almost never call them by the full name, so we'll just say MAC everywhere below.

Our plan to prevent messages changes is to require that all received messages be accompanied by a *tag*. A tag is a just a message-specific bitstring, which intuitively should only be computable by the sender who knows the secret key k. For security, we want that it should be hard to "forge" a tag when someone doesn't know the key.

We start with a definition, which is very simple.

Definition 11.1. *A function*

$$\text{Mac} : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$$

is called a MAC *with key-space $\mathcal{K}$, message-space $\mathcal{M}$, and tag-space $\mathcal{T}$.*

This doesn't say anything about security, but the names at least guide usage (see Figure 11.3). The idea is that the endpoints share a random $k \in \mathcal{K}$. When they want to send a message $m \in \mathcal{M}$, they compute $t \leftarrow \text{Mac}(k, m)$ and send m and t together. Upon receiving some (possibly modified) message and tag pair $(\hat{m}, \hat{t})$, the receiver checks if the tag is correct, meaning it checks if $\hat{t} \stackrel{?}{=} \text{Mac}(k, \hat{m})$. If not, then it throws an error, and otherwise it accepts $\hat{m}$ as authentic.

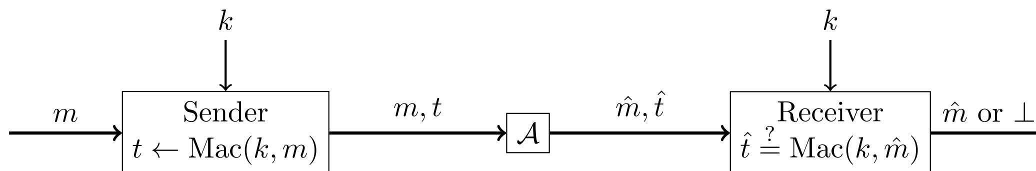

Figure 11.3: A channel protected with a MAC.

Before going further, let's stop to note one attack that we won't prevent: if $\mathcal{A}$ observes a message/tag pair m, t being sent, then later it can *replay* this pair by sending it again. In our diagram above there's nothing the MAC can do stop this, since it really is the correct t. Thus we'll have to deal with replay attacks by changing the message we send, or by other means like timestamps in the messages themselves.

11.0.2 MAC Security: Unforgeability

It's clear that we need some security property from MAC in order for it to do any good. We'd like to somehow say that producing a tag for a message is hard if you don't know the key. As with our encryption definitions, we've going to think "worst case". Namely, we'll want the following properties of a security definition:

1. An adversary is trying to win an experiment by "forging" a tag on a message.
2. We'll declare an adversary to have won if it can produce a tag on *any* message $\hat{m} \in \mathcal{M}$ that was not tagged by the sender. It doesn't matter if $\hat{m}$ is "meaningful" or not (because we can't formalize this; what's meaningful will depend on the application).

3. We won't count replay attacks as winning, for the reasons discussed above.

4. We'll assume that the adversary can observe several message/tag pairs to help it try to forge a tag. In practice these examples come from some protocol run by the endpoints. In our worst-case thinking, we'll let $\mathcal{A}$ pick these example messages to be tagged. That way, no matter protocol the endpoints run, we'll have confidence that forging is hard.

5. We'll assume that the adversary can try injecting messages and see if the receiver rejects the tags or not. In practice an adversary can often tell whether or not a server rejects a message, so the definition will give the ability explicitly.

With those points in mind, here's the definition.

Definition 11.2. *Let* $\mathrm{Mac} : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$ *and let* $\mathcal{A}$ *be an algorithm. Define algorithm* $\mathbf{Expt}_F^{\mathrm{uf}}(\mathcal{A})$ *as*

Alg $\mathbf{Expt}_F^{\mathrm{uf}}(\mathcal{A})$

01 *Pick* $k \xleftarrow{\$} \mathcal{K}$
02 *Run* $\mathcal{A}^{\mathrm{Mac}(k,\cdot),\mathrm{Vrfy}_k(\cdot,\cdot)}$*, where the second oracle is given below.*
03 *If* $\mathcal{A}$ *ever queried* $\mathrm{Vrfy}(\cdot,\cdot)$ *on an input* $(\hat{m}, \hat{t})$ *such that*
 (1) $\hat{t} = \mathrm{Mac}(k, \hat{m})$*, and*
 (2) $\hat{m}$ *was not previously queried to the first oracle,*
 then output 1.
04 *Else: Output 0*

Oracle $\mathrm{Vrfy}_k(\hat{m}, \hat{t})$

 If $\hat{t} = \mathrm{Mac}(k, \hat{m})$*: Return 1*
 Else: Return 0

Define the unforgeability advantage of $\mathcal{A}$ against Mac *as*

$$\mathbf{Adv}_{\mathrm{Mac}}^{\mathrm{uf}}(\mathcal{A}) = \Pr[\mathbf{Expt}_{\mathrm{Mac}}^{\mathrm{uf}}(\mathcal{A}) = 1].$$

We'll only need one security definition for MACs this quarter.

The $\mathrm{Mac}(k, \cdot)$ oracle in the experiment gives the adversary the ability to look at whatever tag it wants. The second oracle $\mathrm{Vrfy}_k(\cdot, \cdot)$ serves two purposes: First, the adversary can probe it to see if a message/tag pair is acceptable to the receiver. Second, it uses this oracle to win the game, but submitting a message/tag pair $\hat{m}, \hat{t}$ that counts as a *forgery*, meaning that $\hat{m}$ was not previously tagged by the first oracle and yet $\mathcal{A}$ cooked up an acceptable tag anyway. The rationale for these oracle follows the desired features we listed before the definition.

To solve the next exercise, use the ideas from the CCA notes to mount a malleability attack, but phrase it in terms of an $\mathcal{A}$ that fits the syntax of the unforgeability experiment.

Exercise 11.1. *Let* $\mathrm{Mac} : \{0,1\}^\ell \times \{0,1\}^\ell \to \{0,1\}^\ell$ *be the* ℓ*-bit one-time page (*$\mathrm{Mac}(k, m) = k \oplus m$*). Give a simple adversary issuing one query to each oracle such that* $\mathbf{Adv}_{\mathrm{Mac}}^{\mathrm{uf}}(\mathcal{A}) = 1$.

11.0.3 Constructing a Fixed-Length MAC

The exercise above gives a non-example of a MAC. Constructing an example of a secure MAC is much harder; Actually *proving* any particular function Mac is secure is well beyond our current abilities, so we'll need to start with some other object as a foundation. For MACS, a block cipher is a natural starting point. In fact, a block cipher that has good PRP security will already be a good MAC!

One can intuitively sum up the relation between MAC and PRP security as: MAC security is about *predicting* a function F on a point before you query it. But if the output space $\mathcal{T}$ is large, and you can predict a function like that, then you must actually be distinguishing it from a random permutation.

Given a MAC that is secure for more than one message block is quite hard (this is in contrast to randomized encryption, where something simple like CTR mode is easy to state and not too hard to analyze). The next exercise shows that some simple attempts don't work.

Exercise 11.2. *Let* AES *be the 128-bit AES block cipher as usual. Consider* $F_1 : \mathcal{K} \times \mathcal{M} \to \mathcal{T}_1$ *and* $F_2 : \mathcal{K} \times \mathcal{M} \to \mathcal{T}_2$*, with* $\mathcal{K} = \{0,1\}^{128}$, $\mathcal{M} = \{0,1\}^{128} \times \{0,1\}^{128}$, $\mathcal{T}_1 = \{0,1\}^{256}$*, and* $\mathcal{T}_2 = \{0,1\}^{128}$*, defined by*

$$F_1(k, m_1 \| m_2) = \mathsf{AES}(k, m_1) \| \mathsf{AES}(k, m_2),$$
$$F_1(k, m_1 \| m_2) = \mathsf{AES}(k, m_1) \oplus \mathsf{AES}(k, m_2).$$

Give a simple adversaries $\mathcal{A}_1, \mathcal{A}_2$ *such that* $\mathbf{Adv}^{\mathrm{uf}}_{F_1}(\mathcal{A}_1) = 1$ *and* $\mathbf{Adv}^{\mathrm{uf}}_{F_2}(\mathcal{A}_2) = 1$.

11.0.4 A MAC for Long, Variable-Length Messages: CBC-MAC

In practice, something called CBC-MAC is often used for multi-block messages. We won't even attempt to analyze it, as the proof is a bit too complicated. But it's worth knowing in case you ever come across it. Here is the code: It takes message-space $\mathcal{M}$ consisting of strings of at most $2^{128} - 1$ bits long; This is almost, but not quite, infinite, and is just a technical condition. The key-space and tag-space are $\mathcal{K} = \mathcal{T} = \{0,1\}^{128}$. The construction works as follows:

Alg Mac(k, m)

- Let $\mathsf{len} \in \{0,1\}^{128}$ be the bit-length of m, encoded as a 128-bit integer.
- $\overline{m} \leftarrow \mathsf{pad}_{\mathrm{cbc}}(m)$
- Parse $\overline{m}[1] \| \cdots \| \overline{m}[L] \leftarrow \overline{m}$ *(128-bit blocks)*
- $v \leftarrow \mathsf{len}$
- For $i = 1, \ldots, L$:
  - $v \leftarrow \mathrm{AES}(k, v \oplus \overline{m}[i])$
- Output v

The padding function $\mathsf{pad}_{\mathrm{cbc}}$ here is different from the PKCS padding we saw before (I'm not sure why, actually). Luckily, $\mathsf{pad}_{\mathrm{cbc}}$ is simple: it just appends a 1 bit followed by the required number of zero bits to get the resulting output to be a multiple of 128 bits long.

As the name suggests, CBC-MAC is similar to CBC encryption. The main differences are that

1. CBC-MAC is deterministic. Instead of randomness, the first block is set to an encoding of the length.

2. Only the final value of v is output. Outputting the intermediate values of v results in an insecure MAC (a good exercise!). You can check this even for messages such that $L = 2$.

Exercise 11.3. *Consider a version of CBC-MAC where the length block is not used. More precisely, define* Mac_0 *to be exactly like* Mac *above, change the line just before the "for" loop to* $v \leftarrow 0^{128}$ *instead of* $v \leftarrow \mathsf{len}$*. Give a simple* $\mathcal{A}$ *such that* $\mathbf{Adv}^{\mathrm{uf}}_{\mathrm{Mac}_0}(\mathcal{A}) = 1$*. (Hint: A single, one-block query to* $\mathrm{Mac}_0(k, \cdot)$ *is enough. The simplest attack I can think of then forges a tag on a two-block message* $\hat{m}$*.) This shows that we can't simply ignore then length block!*

Exercise 11.4. *Requiring the length block at the start of the message is annoying, since it means you can't start processing until you know many bits you're sending. It'd be nice if we could move the length block to the end, but unfortunately this makes it insecure!*

Concretely, give a simple attack against the following MAC:

Alg $\mathrm{Mac}(k, m)$

Let $\mathsf{len} \in \{0,1\}^{128}$ *be the bit-length of* m*, encoded as a* 128*-bit integer.*
$\overline{m} \leftarrow \mathsf{pad}_{\mathrm{cbc}}(m)$
Parse $\overline{m}[1] \| \cdots \| \overline{m}[L] \leftarrow \overline{m}$ (128-bit blocks)
$v \leftarrow 0^{\ell}$ *//start with zeros instead of length*
For $i = 1, \ldots, L$*:*
  $v \leftarrow \mathrm{AES}(k, v \oplus \overline{m}[i])$
$v \leftarrow \mathrm{AES}(k, v \oplus \mathsf{len})$ *// length block last*
Output v