**ISA 562: Information Security, Theory and Practice**

Lecture 3

# 1 Defining Message authentication

## 1.1 Defining MAC schemes

In the last lecture we saw that, even if our data is encrypted, a clever adversary can still manipulate ciphertexts and learn from server responses to the manipulated messages. It is very important, therefore, that we also consider authenticating our messages. Intuitively, the goal is to enable two parties that share a key to send messages to one another, together with a proof that the person sending the message knows the secret key. We achieve this using message authentication codes (MACs), which, like encryption schemes, consist of three algorithms, in this case: $(\mathsf{Gen}, \mathsf{MAC}, \mathsf{Vrfy})$. It also has associated with it a keyspace, $\mathcal{K}$, a message space, $\mathcal{M}$, and a message tag space, $\mathcal{T}$.

$\mathsf{Gen}(1^n)$: outputs the shared key, $k$.
$\mathsf{MAC}(k, m)$: uses the key $k$ to compute a *tag* $t$ for the message $m$. It outputs $t$.
$\mathsf{Vrfy}(k, m, t)$: verifies that the tag $t$ is valid for message $m$. It outputs either 1 or 0.

Just as in the case of encryption, we have a correctness requirement – it should always be the case that a good MAC verifies. That is, we insist that honest parties that share a key will not have their messages rejected. Therefore, we require that for all $k \in \mathcal{K}$, for all messages $m \in \mathcal{M}$, $\Pr[\mathsf{Vrfy}(k, (m, \mathsf{MAC}(k, m))) = 0] = 0$. In many cases, the $\mathsf{Vrfy}$ algorithm is deterministic, always outputting the same tag $t$ for some fixed message $m$ and key $k$. In this case, we can verify a tag $t$ by simply recomputing it ourselves and comparing the result to the value $t$ that was provided. This is called *canonical verification*.

## 1.2 Defining security for MAC schemes

As in the case of message encryption, we define security for MACs through a security game between an adversary and a challenger. We consider a particular scheme secure if there does not exist any polynomial-time adversary that can succeed in this game (with more than negligible probability), whenever the challenger uses the suggested scheme. Equivalently, we can show that a particular MAC scheme is insecure by demonstrating that some adversary wins in this game whenever the challenger uses the proposed scheme.

$\mathsf{Mac\text{-}forge}(n)$:
1. $k \leftarrow \mathsf{Gen}(1^n)$
2. $\mathcal{A}(1^n)$ is given oracle access to $\mathsf{MAC}(k, \cdot)$. The adversary eventually outputs $(m, t)$.
   Let $Q$ denote the set of all queries that $\mathcal{A}$ asked its oracle
3. $\mathcal{A}$ succeeds iff a) $\mathsf{Vrfy}(k, m, t) = 1$ and b) $m \notin Q$. In that case, the experiment output is defined to be 1.

**Definition 1** *A MAC,* $\Pi = (Gen, Mac, Vrfy)$ *is* existentially unforgeable under and adaptive chosen-message attack *if for all probabilistic poly-time adversaries* $\mathcal{A}$ *there is a negligible function* $negl()$ *such that* $\Pr[\mathsf{Mac\text{-}forge}(n) = 1] \leq negl(n)$

One issue that we want to pay attention to when we define security is that the adversary might have an opportunity to see some valid message / tag pairs. We want to ensure that the adversary cannot use these pairs in some clever way to create *new* forgeries. This is captured in Step 2 of the security definition: when we say the adversary is given "oracle access to $\mathsf{MAC}(k, \cdot)$", we mean that he can submit arbitrary messages of his choice to the challenger, who responds by computing and sending the $\mathsf{MAC}$ on each submitted message, using the key $k$ that was chosen in Step 1. This allows the adversary to gather information, capturing whatever information he might be able to gather in the real world, when this scheme is eventually used in a real application. Note that we allow the adversary to choose these messages however he would like, because we do not know how this scheme will eventually be used, and so we do not know what message / tag pairs a true adversary might witness in the real world. By allowing the adversary to pick the messages in this security game, we are capturing the fact that, regardless of what messages the users choose to authenticate, the adversary cannot use the message / tag pairs that he observes to later create a forgery.

Intuitively, a secure MAC scheme should prevent a malicious party that doesn't know a key $k$ from generating *any* valid pair, $(m, t)$, such that $\mathsf{Vrfy}(k, (m, t))$ outputs 1. If the adversary can find *any* forgery, even a forgery on some random, meaningless sequence of bits, we should consider this to be a dangerous attack on our scheme. Indeed, we will see in the next lecture that many applications require the users to MAC random strings, so an adversary that is able to carry out even this seemingly minimal attack could cause a lot of damage. On the other hand, we can't prevent him from "replaying" any of the pairs that he witnessed previously: if he is listening when one party sends an authenticated message, he can always write it down, bring it back tomorrow, and claim that he is the one that holds the key and created the tag on that message. There is no universally acceptable way to handle replay attacks, so we do not try to capture this in our definition, and we leave the issue to the application layer. (Some suggestions of how to do that appear below.) The two properties defining a successful attack in Step 3 are meant to capture these two facts. On the one hand, we do not consider this adversary successful if all he manages to do is bring back $(m, t)$, where $m \in Q$ was one of the messages he was given a tag on in step 2. This is just a replay attack, and not something we can hope to prevent, so we do not consider this a successful attack. However, if he manages to find a forgery on *any* other message, $m \notin Q$, regardless of its format, we consider this a successful attack, and we must throw away the proposed MAC scheme.

There are several natural ways to handle replay attacks at the application layer. If the two parties are able to stay synchronized and maintain state, then they could keep a counter, and insist that the MACd messages have the correct counter value prepended. Any message that does not have the correct counter prepended should then be rejected, even if the $\mathsf{Vrfy}$ would output 1. Another option is to prepend the time before computing the MAC. The users would have to agree on how much clock skew they are willing to accept: if a message has a timestamp that is either a little old or a little in the future, this is not necessarily a replay attack, while if it the timestamp is from last week, it is unlikely this is due to clock skew or network delay. The amount of skew that should be tolerated is application specific, which is why we do not build such a defense into our security definition.

# 2 Constructing MACs from block ciphers

## 2.1 Secure schemes for fixed-length messages

If the users know that their messages will always be exactly a single block length in size, then there is a simple construction of a MAC. We can simply run the message through the block cipher, and output the result as the tag.

$\mathsf{Gen}(1^n)$: Output a random $k \leftarrow \{0,1\}^n$.
$\mathsf{MAC}(k, m)$: Compute $F(k, m) = t$. Output $t$.
$\mathsf{Vrfy}(k, (m, t))$: If $|m| \neq n$, output 0. Otherwise, compute $F(k, m) = t'$. Output 1 iff $t == t'$.

It can be proven that as long as $F$ is a secure pseudorandom function (i.e. a secure block cipher), then this MAC scheme satisfies the definition provided in the previous section. Note that the MAC algorithm in this scheme is deterministic, but, unlike in the case of encryption, this is not a problem. Also, as mentioned above, because a deterministic MAC algorithm ensures that the same tag is always output for a fixed message, we can use canonical verification, as is done here. That is, we simply recompute the tag, and compare to the tag that was provided.

Obviously, the limitation on message size is a drawback with the construction above. We next show how to support arbitrary, but still fixed, message sizes. By that we mean that the users can fix some message size of their choice at the time that they share their key, and then agree to never accept any messages that are not of that size. Below we assume that the fixed size they agreed upon was $n \cdot \ell$. The construction looks a lot like CBC mode encryption, but with two important distinctions, described below.

$\mathsf{Gen}(1^n)$: Output a random $k \leftarrow \{0,1\}^n$.
$\mathsf{MAC}(k, m)$: Parse the message $m = m_1 || \cdots || m_\ell$.
Compute $t_1 = F(k, m_1)$, and,
for $1 < i \leq \ell$, compute $t_i = F(k, t_{i-1} \oplus m_i)$.
Output $t_\ell$.
$\mathsf{Vrfy}(k, m, t)$: If ($|m| == n \cdot \ell$), and $\mathsf{MAC}(k, m) == t$, output 1. Otherwise, output 0.

There are two important distinctions between this and CBC-mode encryption. The first is that we have (implicitly) used an IV that is $0^n$. (Note that $t_1 = F(k, m_1) = F(k, m_1 \oplus 0^n)$.) The second is that we only output $t_\ell$, rather than $t_1, \cdots, t_\ell$. Note that, because we can send the message to the receiver, they can compute all of the $t_i$ values on their own, using the key $k$ and the message blocks $m_1, \cdots, m_\ell$. In contrast, in CBC-mode decryption, the receiver's task was different: they were not given $m_1, \cdots, m_\ell$, and they had to use $t_1, \cdots, t_\ell$ to recover those values (though, there we called these values $c_1, \cdots, c_\ell$).

These particular distinctions are crucial. We show now that had we modified CBC-MAC to remove either of these 2 changes to CBC-mode encryption, we would have an insecure MAC scheme. That is, if we modified our scheme in these ways, we can demonstrate an adversary that succeeds in the security game Mac-forge described above.

## 2.2 Insecure variants of CBC-MAC

Consider the first insecure MAC scheme:

Insecure-CBC-MAC-1:
$\mathsf{Gen}(1^n)$: Output a random $k \leftarrow \{0,1\}^n$.
$\mathsf{MAC}(k,m)$: Parse the message $m = m_1||\cdots||m_\ell$.
Choose random $IV \leftarrow \{0,1\}^n$. Compute $t_1 = F(k, m_1 \oplus IV)$, and,
for $1 < i \leq \ell$, compute $t_i = F(k, t_{i-1} \oplus m_i)$.
Output $(IV, t_\ell)$.
$\mathsf{Vrfy}(k,m,(IV,t_\ell))$: If $|m| \neq n \cdot \ell$, output 0. Otherwise, parse the message as $m = m_1||\cdots||m_\ell$.
Compute $t'_1 = F(k, m_1 \oplus IV)$, and,
for $1 < i \leq \ell$, compute $t'_i = F(k, t'_{i-1} \oplus m_i)$.
Output 1 iff $t'_\ell == t_\ell$.

If $(|m| == n \cdot \ell)$, and $\mathsf{MAC}(k,m) == t$, output 1. Otherwise, output 0.

To show that this is insecure, consider the following adversary that plays Mac-forge with a challenger, both using this scheme. Let's assume that the agreed message length is $2n$ (that is, $\ell = 2$). The adversary chooses any message of his choice, $m^{(1)} = m_1^{(1)}||m_2^{(1)}$ and submits it to the MAC oracle in step 2 of the security game. Let $(IV, t_2^{(1)})$ denote the tag he receives on message $m^{(1)}$ in response to his query. Recall this is computed as $t_1^{(1)} = F(k, IV \oplus m_1^{(1)})$, and $t_2^{(1)} = F(k, t_1^{(1)} \oplus m_2^{(1)})$ (and then discarding $t_1^{(1)}$). The adversary outputs the following forgery. He fixes any $\tilde{m}_1$ of his choice, and defines his forgery message as $m = \tilde{m}_1||m_2^{(1)}$. He computes $IV' = IV \oplus m_1^{(1)} \oplus \tilde{m}_1$, and outputs $(IV', t_2^{(1)})$.

To show that this adversary succeeds in the Mac-forge game, we have to show that his message tag pair satisfy the 2 stated constraints in Step 3 of the game. a) $\mathsf{Vrfy}(k,(m,(IV',t_2^{(1)}))) = 1$. Recall, Vrfy simply computes $\mathsf{MAC}(k,m)$ and compares the result with $(IV', t_2^{(1)})$. $\mathsf{MAC}(k,m)$ is computed as $\tilde{t}_1 = F(k, IV' \oplus \tilde{m}_1)$ and $\tilde{t}_2 = F(k, \tilde{t}_1 \oplus m_2^{(1)})$. However, note that, because of the way $IV'$ was chosen

$$\begin{aligned} \tilde{t}_1 &= F(k, IV' \oplus \tilde{m}_1) \\ &= F(k, (IV \oplus m_1^{(1)} \oplus \tilde{m}_1) \oplus \tilde{m}_1) \\ &= F(k, (IV \oplus m_1^{(1)})) \\ &= t_1^{(1)} \end{aligned}$$

Therefore, $\tilde{t}_2 = F(k, t_1^{(1)} \oplus m_2^{(1)}) = t_2^{(1)}$, and the tag matches the one provided by the adversary. Intuitively, what happened here is that, because we allow arbitrary IV values, the adversary was able to manipulate IV', specifically ensuring that the first input to the block cipher matched the first input to the block cipher for the message / tag pair he was given in Step 2. From there, since his own forged message maintained the same 2nd block as the message from his query, the 2nd tag block also remains the same, and he has found a good forgery.

The 2nd constraint for success in Mac-forge is that the message the adversary forges be different from all of the messages he queried in Step 2. Note that the 2nd block of this message is identical

to the 2nd block of the message that he submitted as a query in Step 2. But, because the 1st block has been modified, $m \neq m^{(1)}$, this still satisfies the requirements. As a practical matter, you could imagine that $m_1^{(1)}$ contains important header information, which the adversary just managed to manipulate arbitrarily. But, as per the prior conversation in Section 1.2, regardless of what real world attack we can think of that might correspond to this attack on the security game, we should dismiss this MAC scheme immediately, since we have no idea what application might use it, and how such attacks might impact that application.

Insecure-CBC-MAC-2:
$\mathsf{Gen}(1^n)$: Output a random $k \leftarrow \{0,1\}^n$.
$\mathsf{MAC}(k, m)$: Parse the message $m = m_1||\cdots||m_\ell$.
Compute $t_1 = F(k, m_1)$, and,
for $1 < i \leq \ell$, compute $t_i = F(k, t_{i-1} \oplus m_i)$.
Output $(t_1, \cdots, t_\ell)$.
$\mathsf{Vrfy}(k, m, (t_1, \cdots, t_\ell))$: If $|m| \neq n \cdot \ell$, output 0. Otherwise, parse the message as $m = m_1||\cdots||m_\ell$.
Compute $t'_1 = F(k, m_1)$, and,
for $1 < i \leq \ell$, compute $t'_i = F(k, t'_{i-1} \oplus m_i)$.
Output 1 iff for $1 \leq i \leq \ell, t'_i == t_i$.

We show how an adversary can forge any arbitrary message of his choice. Let that message be $m = m_1||\cdots||m_\ell$. He will make $\ell$ queries to the oracle in Step 2 of the security game, and each query will enable him to determine one of the $\ell$ blocks that make up a forgery on his message. To learn $t_1$, he queries $m^{(1)} = m_1||\underbrace{0||\cdots||0}_{\ell-1 \text{ times}}$. Letting $(t_1^{(1)}, \cdots, t_\ell^{(1)})$ denote the response that he gets from the challenger, note that $t_1^{(1)} = F(k, m_1)$, which is exactly the value $t_1$ that should appear as the first element in a valid MAC on $m$. To learn $t_2$, he submits $m^{(2)} = t_1^{(1)} \oplus m_2||\underbrace{0||\cdots||0}_{\ell-1 \text{ times}}$. Letting $(t_1^{(2)}, \cdots, t_\ell^{(2)})$ denote the MAC that he receives in response, note that $t^{(2)} = F(k, t^{(1)} \oplus m_2)$ which is precisely the value that should appear as the 2nd element in a valid MAC on $m$. He continues in this way, letting $m^{(i)} = t^{(i-1)} \oplus m_i||\underbrace{0||\cdots||0}_{\ell-1 \text{ times}}$, and using the first block of the challenger's response to define $t_i$. Finally, he outputs $(m, t_1, \cdots, t_\ell)$ as his valid message and forgery. The reader can verify that $\mathsf{Vrfy}(k, m, (t_1, \cdots, t_\ell))$ would output 1. To see that the 2nd requirement of Step 3 in the challenge game is satisfied, note that for any $i \in \{1, \cdots, \ell\}$, $m \neq m^{(i)}$.[1]

## 2.3 Secure MACs from block ciphers, arbitrary length messages

Recall that in both of the secure MACs we saw above, we required the users to agree on the message size before sending any messages, and to reject any message that wasn't of the appropriate size. What goes wrong with CBC-MAC if users don't reject messages of the wrong size? Suppose we modified the verification algorithm to ignore the size check, but that it otherwise remained the same. In this case, CBC-MAC uses a fixed IV of 0, and only outputs the final block of $t_\ell$,

[1] Well, implicitly we're assuming that $m$ does not itself has $\ell - 1$ 0 blocks. But if the message that the adversary wished to forge had that property, we can use any other garbage to fill the last $\ell - 1$ blocks in each of his queries.

so we might assume it is not subject to either of the attacks we described in the previous subsection. Unfortunately, even though the MAC algorithm only outputs the last tag block, it is easy to trick the challenger into giving all $\ell$ blocks on any message. Suppose the adversary would like to learn $t_1, \cdots, t_\ell$ corresponding to message $m_1||0|| \cdots ||0$, as he did in the last attack that we saw. Currently, the MAC algorithm will only reply with $t_\ell$, and what he really needs for his attack is $t_1$. But by shortening the message to the single block $m_1$, the MAC algorithm, again outputting only the last tag block, will now output exactly $t_1$, which is what the adversary needs for his forgery. The reader should convince themselves that they can carry out the 2nd attack described above, and that the resulting forgery is "valid", in the sense that the forged message was never queried to the challenger in Step 2.

There are two easy fixes to CBC-MAC that would allow us to use it for arbitrary length messages. The first is to pre-pend $|m|$ as a length $n$ string. Verification should check this value and output 0 if it doesn't match the actual size of the message being verified. How does prevent the attack that was just described?

The other fix is to use two different keys: the first as was already done in CBC-MAC, and the 2nd to provide a fixed-message MAC on the result. Formally,

$\mathsf{Gen}(1^n)$: Choose a random $k_1 \leftarrow \{0,1\}^n$, and random $k_2 \leftarrow \{0,1\}^n$. Output $(k_1, k_2)$.
$\mathsf{MAC}(k, m)$: Parse the message $m = m_1|| \cdots ||m_\ell$.
Compute $t_1 = F(k_1, m_1)$, and,
for $1 < i \leq \ell$, compute $t_i = F(k_1, t_{i-1} \oplus m_i)$.
Output $F(k_2, t_\ell)$.
$\mathsf{Vrfy}((k_1, k_2), m, t)$: Perform canonical verification.

Of course, while it's useful to think about how these fix address the particular attacks we saw in the previous section, we can't know what other attacks might lie out there. To be sure that these fixes suffice, we should prove that NO polynomial time adversary can win in the Mac-forge game, as long as we make these modifications. We don't do that in this class, but a proof of this fact does exist.

# 3 Hash functions

A hash function is a fixed function that takes arbitrary length input and outputs a fixed length string (say, of size 256 bits). Because hash functions can take arbitrary length input, note that, on most inputs, such functions are *compressing*. In particular, the number of possible outputs from a hash function are much fewer than the infinite number of possible inputs! Nevertheless, the security property that we require is that it is hard to find collisions in this function. That is, it is hard to find two different inputs $x_1, x_2$, such that $x_1 \neq x_2$, but $H(x_1) = H(x_2)$. We call such a function *collision resistant*. Because the function is compressing, it is easy to see that such collisions must exist. Despite that, if certain computational problems are difficult, we can prove that collision-resistant hash functions do indeed exist.[2]

[2]Technically, we can't even define what what we mean when we say that it is hard to find collisions in some single fixed function. What if a polynomial time adversary is given two colliding points, and he hardcodes them into his "search" algorithm? So, from a theoretical standpoint, we should actually define a *family*, or collection, of hash functions, and claim that when we choose a function at random from this family, no fixed adversary can find collisions

In addition to collision resistance, it is also common to assume that the output of a hash function looks random. In fact, this is a bit problematic, and we can prove that no single, fixed function can be indistinguishable from a truly random function. It is best to avoid this assumption if it is possible. But, it is still sometimes a useful assumption, and we don't know of any real attacks on systems that have made use of this assumption.

### 3.1 Hash and MAC

Hash functions give us a very easy mechanism for creating secure MACs. Since the output of a hash function has fixed length, we can use a fixed-length MAC from the beginning of this lecture, and, rather than MACing our message directly, we can simply MAC the hash of the message. Since it is hard to find collisions under this hash function, it would be hard for the adversary to find a different message that has the same MAC! In following scheme, we let $H$ denote a collision-resistant hash function with output size $n$.

$\mathsf{Gen}(1^n)$: Output a random $k \leftarrow \{0,1\}^n$.
$\mathsf{MAC}(k, m)$: Compute $F(k, H(m)) = t$. Output $t$.
$\mathsf{Vrfy}(k, (m, t))$: Compute $F(k, H(m)) = t'$. Output 1 iff $t == t'$.

### 3.2 HMAC

The above construction is provably secure (assuming a secure block-cipher and a collision-resistant hash function), but for a long time, developers were not using this construction, and instead opted for a simpler MAC that simply pre-pended the key to the message and computed the hash: $\mathsf{MAC}(k, m) = H(k||m)$. This is actually insecure, and real attacks on such a construction exist! The problem has to do with the particular design of commonly used hash functions, and we don't discuss the issue here. However, we point out that the reason people preferred this MAC is because block-ciphers are a little bit slower to compute than hash functions, and this gave them some performance speedup. Cryptographers responded, and developed a new MAC that is provably secure, and relies only on collision-resistant hash functions. The result is called HMAC, and it is now the most common way of handling message authentication. The scheme is described below. $H$ is a hash function, and $\mathsf{ipad}$ and $\mathsf{opad}$ are fixed, constants values.

$\mathsf{Gen}(1^n)$: Output a random $k \leftarrow \{0,1\}^n$.
$\mathsf{MAC}(k, m)$: $H(k \oplus \mathsf{opad}||H(k \oplus \oplus ipad||m))$

Intuitively, we can view this as another example of "hash and mac": the inner hash brings the message down to a fixed length, and the outer application of the hash function provides the secure MAC. This construction is provably secure

in the chosen function. This is similar to what we did for pseudorandom functions, where we can view the key $k$ as "selecting" one of the functions in the family. In practice, nobody does this – instead, we all just use one fixed, unkeyed hash function; today, we usually use SHA-256. In practice, nobody has found any collisions in SHA-256, so we just let this slide and accept the disconnect between what we can prove and what we do in practice.

# 4 Authenticated encryption

Message authentication and message encryption should be thought of as two distinct tasks. As we've seen, the security goals are completely different. But, certainly it is often important to have both message authentication and encryption, so what is the right way to achieve both properties? There are 3 obvious approaches, but only one of them is secure.

1. Authenticate and encrypt. That is, using some encryption scheme $(\mathsf{Gen}_1, \mathsf{Enc}, \mathsf{Dec})$ and some MAC $(\mathsf{Gen}_2, \mathsf{MAC}, \mathsf{Vrfy})$, generate $k_1 \leftarrow \mathsf{Gen}_1(1^n)$, and $k_2 \leftarrow \mathsf{Gen}_2(1^n)$. Then, on message $m$, compute $c = \mathsf{Enc}(k_1, m)$ and $t = \mathsf{MAC}(k_2, m)$. Output $(c, t)$.

2. Authenticate, then encrypt. Generate the keys as above, but then compute $t = \mathsf{MAC}(k_2, m)$, and $c = \mathsf{Enc}(k_1, m||t)$. Output $c$.

3. Encrypt, then authenticate. Generate the keys as above, but then compute $c = \mathsf{Enc}(k_1, m)$, $t = \mathsf{MAC}(k_2, c)$, and Output $(c, t)$.

The first approach is very clearly insecure, because there is no guarantee that a MAC preserves any privacy of the message at all. So, including $\mathsf{MAC}(k_2, m)$ could mean including $m$ in the clear, completely destroying any privacy we were hoping to gain from the encryption scheme. But even if we used a MAC scheme that does not appear to completely leak the message, note that the Mac-forge security game says nothing about the privacy of the message, so we can't possibly found such a construction on a security proof.

The second approach is also insecure, though the issue is less obvious. (In fact, this approach was used in early versions of IPSec, before people realized the issue.) If we authenticate and then encrypt the MACd message, we are still vulnerable to the same padding attack we saw in the previous class. Suppose, by way of example, that we're again using CBC mode encryption for the encryption portion. (The choice of MAC algorithm is irrelevant.) Note that the $\mathsf{Dec}$ algorithm will recover a pair of values, $m||t$, but before it can do anything to verify $t$, it has to determine how much padding was used in the process of encrypting $m||t$, so that the padding can be removed. Of course, if the padding is intact, but the value of $t$ is manipulated, which will happen at some point during the padding oracle attack, the server will likely still indicate an error. If the attacker cannot tell whether the error is caused due to the manipulated padding, or due to the manipulated MAC, then it might be hard to carry out the attack. But, as long as the attacker can distinguish these two different errors, then the attack can continue as before.

The final approach allows us to verify the tag before we even attempt to decrypt the ciphertext, so the verification process cannot possibly leak any information about the plaintext value. This approach is provably secure, regardless of what encryption scheme and MAC scheme are used. This is the way authenticated encryption should always be performed.

## 4.1 Timing attacks on MACs

All MACs that have canonical verification can become subject to the following clever attack. Recall that in canonical verification, a fresh tag is constructed from the message, using the shared key, and the result is compared against the original tag that was received with the message. In the final step, where the two tags are compared, the comparison algorithm likely compares the two tags one byte at a time, and likely aborts as soon as it finds a byte in which they differ. By timing how long

verification takes, it is sometimes possible to determine exactly how many bytes were compared before the comparison operation terminated. The attacker can therefore guess the tag, one byte at a time, as follows. Let's suppose that $t$ is a valid tag on message $m$ (for some key $k$ that the adversary doesn't know). Let's write $t = t_1||\cdots||t_\ell$, where each $t_i$ is one byte of $t$. Since a byte is 8 bits, there are exactly $2^8 = 256$ possible values for each $t_i$. The adversary will start trying to guess $t_1$, filling the remaining bytes of $t$ with arbitrary values. Until he guesses $t_1$ correctly, the verification algorithm will simply fail in the first byte comparison, but when $t_1$ is actually filled in correctly, it will fail in the 2nd byte comparison. Since one more byte comparison is performed before the abort, there is a small difference in the time it takes to abort, and this can be detected by a careful adversary. The adversary therefore learns when he has guessed $t_1$ correctly! He fixes this byte of $t$, and then continues to start guessing the 2nd byte. If the tag is 32 bytes long, he will have to make at most $256 \times 32$ guesses, totaling 8192. This is quite doable, and obviously far far fewer than it would take to guess all $2^{256}$ possible tag values. This attack was used to load un-authenticated games onto the Xbox 360.