Message Authentication Codes and Digital Signatures

Updated 27Mar11: added minor clarification regarding Fact 3 in Section 6.

We've now looked at symmetric- and asymmetric-key encryption. Encryption is used to protect the *confidentiality* of communications over an insecure channel. Now we'll look at cryptographic schemes that provide *integrity* and *authentication*. In particular, the threat we're concerned about is adversaries who send spoofed messages (pretending to be from a legitimate participant) or who modify the contents of a message from a legitimate participant. To address these threats, we will introduce cryptographic schemes that enable the recipient to detect spoofing and tampering.

We'll look at schemes in both the symmetric-key and asymmetric-key models. If Alice and Bob share a secret key K, they can use a *Message Authentication Code* (also called a MAC, for short) to detect tampering with their messages. If they don't have a shared key, but Bob knows Alice's public key, Alice can sign her messages with her private key, using a *digital signature* scheme (also known as a public-key signature scheme). In tabular form, the big four types of cryptographic primitives are:

	Symmetric-key	Asymmetric-key
Confidentiality	Symmetric-key encryption (e.g., AES-CBC)	Public-key encryption (e.g., El Gamal, RSA encryption)
Integrity and authentication	MACs (e.g., AES-CBC-MAC)	Digital signatures (e.g., RSA signatures)

1 Message Authentication Codes (MACs)

Suppose Alice and Bob share a secret key K, and Alice wants to send a message to Bob over an insecure channel. The message isn't secret, but she wants to prevent attackers from modifying the contents of the message. The idea of a Message Authentication Code (MAC) is to send a keyed checksum of the message along with the message, chosen so that any change to the message will render the checksum invalid.

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. Typically, we might use a 128-bit key K and 128-bit tags. Alice will send the pair of values M, T to Bob, where she computed the tag $T = F(K, M)$ using the MAC. When Bob receives M, T, Bob will compute $F(K, M)$ and check that it matches

the provided tag T. 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.

The algorithm F is chosen so that if the attacker replaces M by some other message M', then the tag will almost certainly[1] no longer be valid: in particular, $F(K, M) \neq F(K, M')$. 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 a message M' that Alice has never sent.

Modern MACs are designed to be secure against known-plaintext attack. For instance, suppose Georgia the Forger eavesdrops on Alice's communications and observes a number of messages and their corresponding tags: $(M_1, T_1), (M_2, T_2), \ldots, (M_n, T_n)$, where $T_i = F(K, M_i)$. Then Georgia 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 Georgia was able to choose the M_i's.

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.

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: $K = (K_1, K_2)$. The message M is decomposed into a sequence of 128-bit blocks: $M = M_1 \cdot M_2 \cdots M_n$. We set $S_0 = 0$ and compute

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

[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.

Finally we compute $T = \text{AES}_{K_2}(S_n)$; T is the tag for message M. This scheme can be proven secure, assuming AES is a secure block cipher.

What does it mean to say that a MAC algorithm is secure? Here is a formal definition. 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 (M_i, T_i) 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 (M_n, T_n) 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.

Something to note: there's no general promise that a MAC algorithm doesn't leak any information about the message M to which it is applied. For some MAC algorithms, it will be clear that they don't leak information because the algorithm directly applies a block cipher (which, if any good, indeed has the property that it does not leak information about the plaintext). But for others, it could be that an eavesdropper Eve who observes the value $F(K, M)$ can infer some information about M. So in situations where we want not only *integrity* for M but also *confidentiality*, we need to consider confidentiality for $F(K, M)$, too. One approach is to compute $F(K, E(M))$ instead; that is, compute the MAC for the ciphertext rather than the plaintext. This value might then leak information about the *ciphertext*, but that's fine; we already assume that Eve can directly view the complete ciphertext.

2 Cryptographic Hash Functions

Cryptographic hash functions are another useful primitive. A cryptographic hash function is a deterministic and *unkeyed* function H; $H(M)$ is called the *hash* of the message M. Typically, the output of a hash function is a fixed size: for instance, the SHA256 hash

algorithm can be used to hash a message of any size, and produces a 256-bit hash value.

A cryptographic hash function can be used to generate a "fingerprint" of a message. Any change to the message, no matter how small, is likely to change many of the bits of the hash value, and there are no detectable patterns to how the output changes. In a secure hash function, the output of the hash function looks like a random string, chosen differently and independently for each message—except that, of course, a hash function is a deterministic procedure.

Cryptographic hash functions have many nice properties, usually including the following:

- **One-way:** The hash function can be computed efficiently: Given x, it is easy to compute $H(x)$. However, given a hash y, it is infeasible to find any input x such that $y = H(x)$. (This property is also known as "preimage resistant.")
- **Second preimage resistant:** Given a message x, it is infeasible to find another message x' such that $x' \neq x$ but $H(x) = H(x')$.
- **Collision resistant:** It is infeasible to find *any* pair of messages x, x' such that $x' \neq x$ but $H(x) = H(x')$.

By "infeasible", we mean that there is no known way to do it with any realistic amount of computing power.

Hash functions can be used to verify message integrity. For instance, suppose Alice downloads a copy of the installation disk for the latest version of the Ubuntu distribution, but before she installs it onto her computer, she would like to verify that she has a valid copy of the Ubuntu software and not something that was modified in transit by an attacker. One approach is for the Ubuntu developers to compute the SHA256 hash of the intended contents of the installation disk, and distribute this 256-bit hash value over many channels (e.g., print it in the newspaper, include it on their business cards). Then Alice could compute the SHA256 hash of the contents of the disk image she has downloaded, and compare it to the hash publicized by Ubuntu developers. If they match, then by the collision-resistance property, it would be reasonable for Alice to conclude that she received a good copy of the legitimate Ubuntu software.

3 Digital Signatures

A *digital signature* is the public-key version of a MAC. Suppose Alice wants to send messages to Bob over an insecure channel. In a digital signature scheme, Alice has a public key (also known as a *verification* key) and a private key (also known as a *signing* key) that she

generated in advance. Bob needs to know Alice's public key, but Alice will keep her private key absolutely secret from everyone.

Mathematically, a digital signature scheme specifies three algorithms:

- **Key generation:** There is a randomized algorithm KeyGen that outputs a matching private key and public key: $(K, U) = \text{KeyGen}()$. Each invocation of KeyGen produces a new keypair.
- **Signing:** There is a signing algorithm Sign: $S = \text{Sign}_K(M)$ is the signature on the message M (with private key K).
- **Verification:** There is a verification algorithm Verify, where $\text{Verify}_U(M, S)$ returns true if S is a valid signature on M (with public key U) or false if not.

If K, U are a matching pair of private and public keys (i.e., they were output by some call to KeyGen), and if $S = \text{Sign}_K(M)$, then $\text{Verify}_U(M, S) = \text{true}$.

To develop a working digital signature scheme in more concrete terms, it now helps to introduce a new concept: trapdoor one-way function.

4 Trapdoor One-way Functions

A *trapdoor one-way function* is a function F that is one-way, but also has a special backdoor that enables someone who knows the backdoor to invert the function.

A trapdoor one-way function is associated with a public key U and a private key K. Given the public key U, it is computationally easy to compute F, but hard to compute F^{-1}. In other words, given x and U, it is easy to compute $y = F(x)$, but given y and U, it is hard to find x such that $y = F(x)$, i.e., it is hard to compute $F^{-1}(y)$.

The private key unlocks the trapdoor. Given the private key K, it becomes easy to compute F^{-1}, and of course it remains easy to compute F. Put another way, given y and K, it becomes computationally easy to find x such that $y = F(x)$, i.e., it is easy to compute $F^{-1}(y)$.

The RSA scheme specifies a particular method for building a trapdoor one-way function, but let's defer the details of how to do that until later.

5 RSA Signatures: High-level Outline

At a high level, the RSA signature scheme works like this. It specifies a trapdoor one-way function F. The public key of the signature scheme is the public key U of the trapdoor function, and the private key of the signature scheme is the private key K of the trapdoor function. We also need a one-way function H, with no trapdoor; we typically let H be some cryptographic hash function. The function H is standardized and described in some public specification, so we can assume that everyone knows how to compute H, but no one knows how to invert it.

A signature on a message M is defined to be a value S that satisfies the following equation:

$$H(M) = F_U(S).$$

Note that given a message M, an alleged signature S, and a public key U, we can verify whether it satisfies the above equation. This makes it possible to verify the validity of signatures.

How does the signer sign messages? It turns out that the trapdoor to F, i.e., the private key K, lets us find solutions to the above equation. Given a message M and the private key K, the signer can first compute $y = H(M)$, then find a value S such that $F_U(S) = y$. In other words, the signer computes $S = F^{-1}(H(M))$; that's the signature on M. This is easy to do for someone who knows the private key K, because K lets us invert the function F, but it is hard to do for anyone who does not know K. Consequently, anyone who has the private key can sign messages.

For someone who does not know the private key K, there is no easy way to find a message M and a valid signature S on it. For instance, an attacker could pick a message M, compute $H(M)$, but then the attacker would be unable to compute $F^{-1}(H(M))$, because the attacker does not know the trapdoor for the one-way function F. Similarly, an attacker could pick a signature S and compute $y = F(S)$, but then the attacker would be unable to find a message M satisfying $H(M) = y$, since H is one-way.

This is the general idea underpinning the RSA signature scheme. Now let's look at how to build a trapdoor one-way function, which is the key idea needed to make this all work.

6 Number Theory Background

Here are some basic facts from number theory, which will be useful in deriving RSA signatures. In the following, $\varphi(n)$ denotes Euler's totient function of n: the number of positive integers less than n that share no common factor with n.

Fact 1 *If* $\gcd(x, n) = 1$*, then* $x^{\varphi(n)} = 1 \pmod{n}$.

Fact 2 *If* p *and* q *are two different odd primes, then* $\varphi(pq) = (p-1)(q-1)$.

Fact 3 *If* $p = 2 \pmod 3$ *and* $q = 2 \pmod 3$*, then there exists a number* d *satisfying* $3d = 1 \pmod{\varphi(pq)}$*, and this number* d *can be efficiently computed given* $\varphi(pq)$.

Let's assume that p and q are two different odd primes, that $p = 2 \pmod 3$ and $q = 2 \pmod 3$, and that $n = pq$. (**UPDATE** 27Mar11: *Why do we pick those particular conditions on* p *and* q*? Because then* $\varphi(pq) = (p-1)(q-1)$ *will not be a multiple of 3, which is going to allow us to have unique cube roots.*) Let d be the positive integer promised to exist by Fact 3. As a consequence of Facts 2 and 3, d can be computed efficiently given knowledge of p and q.

Theorem 1 *With notation as above, define functions* F, G *by* $F(x) = x^3 \bmod n$ *and* $G(x) = x^d \bmod n$*. Then* $G(F(x)) = x$ *for every* x *satisfying* $\gcd(x, n) = 1$.

Proof: By Fact 3, $3d = 1 + k\varphi(n)$ for some integer k. Now applying Fact 1, we find

$$G(F(x)) = (x^3)^d = x^{3d} = x^{1+k\varphi(n)} = x^1 \cdot (x^{\varphi(n)})^k = x \cdot 1^k = x \pmod{n}.$$

The theorem follows.

If the primes p, q are chosen to be large enough—say, 1024-bit primes—then it is believed to be computationally infeasible to recover p and q from n. In other words, in these circumstances it is believed hard to factor the integer $n = pq$. It is also believed to be hard to recover d from n. And, given knowledge of only n (but not d or p, q), it is believed to be computationally infeasible to compute the function G. The security of RSA will rely upon this hardness assumption.

7 RSA Signatures

We're now ready to describe the RSA signature scheme. The idea is that the function F defined in Theorem 1 will be our trapdoor one-way function. The public key is the number n, and the private key is the number d. Given the public key n and a number x, anyone can compute $F(x) = x^3 \bmod n$. As mentioned before, F is (believed) one-way: given $y = x^3 \bmod n$, there is no known way to recover x in any reasonable amount of computing time. However, we can see that the private key d provides a trapdoor: given d and y, we can compute $x = G(y) = y^d \bmod n$. The intuition underlying this trapdoor function is simple: anyone can cube a number modulo n, but computing cube roots modulo n is believed to be hard if you don't know the factorization of n.

We then apply this trapdoor one-way function to the basic approach outlined earlier. Thus, a signature on message M is a value S satisfying

$$H(M) = S^3 \bmod n.$$

The RSA signature scheme is defined by the following three algorithms:

- **Key generation:** We can pick a pair of random 1024-bit primes p, q that are both 2 mod 3. Then the public key is $n = pq$, and the private key is the value of d given by Fact 3 (it can be computed efficiently using the extended Euclidean algorithm).
- **Signing:** The signing algorithm is given by

$$\textsc{Sign}_d(M) = H(M)^d \bmod n.$$

- **Verification:** The verification algorithm Verify is given by

$$\textsc{Verify}_n(M, S) = \begin{cases} \text{true} & \text{if } H(M) = S^3 \bmod n, \\ \text{false} & \text{otherwise.} \end{cases}$$

Theorem 1 ensures the correctness of the verification algorithm, i.e., that $\textsc{Verify}_n(M, \textsc{Sign}_d(M)) = \text{true}$.

A quick reminder: in these notes we're developing the conceptual basis underlying MAC and digital signature algorithms that are widely used in practice, but again don't try to implement them yourself based upon just this discussion! We've omitted some technical details that do not change the big picture, but that are essential for security in practice. Use a reputable crypto library.

8 Definition of Security for Digital Signatures

Finally, let's outline a formal definition of what we mean when we say that a digital signature scheme is secure. The approach is very similar to what we saw for MACs.

We imagine a game played between Georgia (the adversary) and Reginald (the referee). Initially, Reginald runs KeyGen to get a keypair K, U. Reginald sends the public key U to Georgia and keeps the private key K to himself. In each round of the game, Georgia may query Reginald with a message M_i; Reginald responds with $S_i = \textsc{Sign}_K(M_i)$. At any point, Georgia can yell "Bingo!" and output a pair (M, S). If this pair satisfies $\textsc{Verify}_U(M, S) = \text{true}$, and if Reginald has not been previously queried with the message M, then Georgia wins the game: she has forged a signature. Otherwise, Georgia loses.

If Georgia has any strategy to successfully forge a signature with non-negligible probability (say, with success probability at least $1/2^{40}$), given a generous amount of computation time (say, 2^{80} steps of computation) and any reasonable number of rounds of the game (say, 2^{40} rounds), then we declare the digital signature scheme insecure. Otherwise, we declare it secure.

This is a very stringent definition of security, because it declares the signature scheme broken if Georgia can successfully forge a signature on any message of her choice, even after tricking Alice into signing many messages of Georgia's choice. Nonetheless, modern digital signature algorithms—such as the RSA signature scheme—are believed to meet this definition of security.