12. Digital Signatures

We can use the ideas from public-key encryption to build asymmetric cryptographic schemes that guarantee integrity and authentication too. In this section, we will define *digital signatures*, which are essentially the public-key version of MACs, and show how they can help guarantee integrity and authentication.

12.1. Digital signature properties

Recall that in public-key encryption, anyone could use Bob's public key to encrypt a message and send it to him, but only Bob could use his secret key to decrypt the message. However, the situation is different for digital signatures. It would not really make sense if everyone could generate a signature on a message and only Bob could verify it. If anyone could generate a signature with a public key, what's stopping an attacker from generating a malicious message with a valid signature?

Instead, we want the reverse: only Bob can generate a signature on a message, and everyone else can verify the signature to confirm that the message came from Bob and has not been tampered with.

In a digital signature scheme, Bob generates a public key (also known as a *verification* key) and a private key (also known as a *signing* key). Bob distributes his public verification key to everyone, but keeps his signing key secret. When Bob wants to send a message, he uses his secret signing key to generate a signature on the message. When Alice receives the message, she can use Bob's public verification key to check that the signature is valid and confirm that the message is untampered and actually from Bob.

Mathematically, a digital signature scheme consists of three algorithms:

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

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

12.2. 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.

We define a signature on a message M as 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.

12.3. Number Theory Background

Here are some basic facts from number theory, which will be useful in deriving RSA signatures. As previously discussed in lecture, we use $\varphi(n)$ to denote 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* $x^{\varphi(n)} = 1 \pmod{n}$*. ("Euler's theorem.")*

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 $n = pq$.[1] Let d be the positive integer promised to exist by Fact 3. As a consequence of Facts 2 and 3, we can efficiently compute d 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.

12.4. 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 $2 \bmod 3$. 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

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

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

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

Theorem 1 ensures the correctness of the verification algorithm, i.e., that

$$\mathrm{Verify}_n(M, \mathrm{Sign}_d(M)) = \mathrm{true}.$$

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. For your actual systems, use a reputable crypto library!

12.5. 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 $\langle K, U \rangle$. 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 $S_i = \mathrm{Sign}_K(M_i)$. At any point, Georgia can yell "Bingo!" and output a pair $\langle M, S \rangle$. If this pair satisfies $\mathrm{Verify}_U(M, S) = \mathrm{true}$, 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.

Note however that the security of signatures do rely on the underlying hash function. Signatures have been broken in the past by taking advantage of the ability to create hash collisions when the hash function, not the public key algorithm, is compromised.

1 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. ↩