RSA Signatures

6.1600 Course Staff

Fall 2023

In this chapter, we will discuss the RSA digital-signature scheme. The RSA paper[1] was tremendously influential because it gave the first constructions of digital signatures and public-key encryption. (We will talk about public-key encryption in detail later on.)

The RSA cryptosystem is going out of style for a few reasons: generating RSA keys is relatively expensive and the keys are relatively large (4096 bits for RSA versus 256 bits for more modern elliptic-curve-based cryptosystems). In addition, a large-scale quantum computer could—in theory, at least—break RSA-style cryptosystems.

The RSA cryptosystem is worth studying for a few reasons:

- RSA's security is related to the problem of factoring large integers, which is (arguably) the most natural "hard" computational problem out there.
- RSA gives the only known instantiation of a *trapdoor one-way permutation*, which we will define shortly.
- RSA has a number of esoteric properties that are useful for advanced cryptographic constructions. For example, it gives a "group of unknown order." See Boneh-Shoup, Chapter 10.9 for details.
- RSA signatures are used on the vast majority of public-key certificates today.[2]

  The most commonly used type of RSA signatures ("PKCS #1 v1.5") is more complicated—and no more secure—than the construction we describe here, but that construction is still used for historical reasons.

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.

[1] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[2] As of today, around 94% of certificates in the Certificate Transparency logs use RSA signatures: `https://ct.cloudflare.com/`.

1 *Trapdoor one-way permutations*

RSA implements a *trapdoor one-way function*. Informally, a trapdoor one-way function is a function that is easy to compute in the forward direction but that is hard to invert *except* to someone knowing a secret key. So it is like a one-way function with a "trapdoor" that allows efficient inversion.

RSA actually implements a trapdoor one-way *permutation*—that is, it maps an input space onto itself with no collisions.

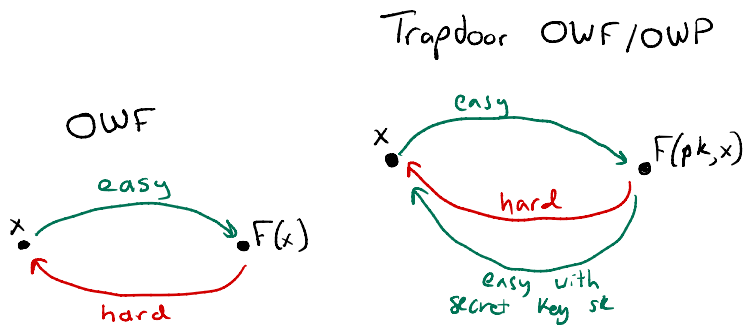

Figure 1: A one-way function (at left) is hard to invert on random inputs. A trapdoor one-way function/permutation (at right) is a function keyed by a public key. The function is easy to invert given the secret key and is hard to invert otherwise.

1.1 *Definition*

Formally, a trapdoor one-way permutation over input space $\mathcal{X}$ is a triple of efficient algorithms:

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$. The key-generation algorithm takes as input the security parameter $\lambda \in \mathbb{N}$, expressed as a unary string, and outputs a secret key and a public key.
- $F(\mathsf{pk}, x) \to y$. The evaluation algorithm F takes as input the public key pk and an input $x \in \mathcal{X}$, and outputs a value $y \in \mathcal{X}$.
- $I(\mathsf{sk}, y) \to x'$. The inversion algorithm I takes as input the secret key sk and a point $y \in \mathcal{X}$, and outputs its inverse $x \in \mathcal{X}$.

If we wanted to be completely formal, the input space would be parameterized by the security parameter λ. So we would have a family of input spaces $\{\mathcal{X}_\lambda\}_{\lambda \in \mathbb{N}}$—one for each choice of λ. This way the input space can grow with λ.

In the RSA construction, the input space $\mathcal{X}$ depends on the public key, but we elide that technical detail here.

Correctness. Informally, we want that for keypairs $(\mathsf{sk}, \mathsf{pk})$ output by Gen, we have that $F(\mathsf{pk}, \cdot)$ and $I(\mathsf{sk}, \cdot)$ are inverses of each other. More formally, for all $\lambda \in \mathbb{N}$, $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$, and $x \in \mathcal{X}$, we require:

$$I(\mathsf{sk}, F(\mathsf{pk}, x)) = x.$$

Security. Security requires that $F(\mathsf{pk}, \cdot)$ is hard to invert (in the sense of a one-way function) on a randomly sampled input in the input space $\mathcal{X}$, even when the adversary is given the public key pk. That is, for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathrm{negl}(\cdot)$ such that

$$\Pr\left[\mathcal{A}(\mathsf{pk}, F(\mathsf{pk}, x)) = x : \begin{array}{c}(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda) \\ x \xleftarrow{R} \mathcal{X}\end{array}\right] \leq \mathrm{negl}(\lambda).$$

When we use the RSA cryptosystem, we make the assumption that the RSA function is hard to invert given only the public key:

Definition 1.1 (RSA Assumption)**.** The RSA function (Gen, F, I) is a trapdoor one-way permutation.

IMPORTANT: Just as a one-way function is only hard to invert on a *randomly sampled input*, a trapdoor one-way function is only hard to invert on a randomly sampled input. Many of the cryptographic failures of RSA come from assuming that the RSA one-way function is hard to invert on non-random inputs.

1.2 *Digital signatures from trapdoor one-way permutations*

This construction is called "full-domain hash."[3] The construction makes use of a hash function H and resulting signature scheme is secure, provided that we model the hash function H as a "random oracle."

[3] Mihir Bellare and Phillip Rogaway. "Random oracles are practical: A paradigm for designing efficient protocols". In: *ACM Conference on Computer and Communications Security*. 1993.

In other words, to argue security, it is not sufficient to show that H is, for example, collision resistant. Instead, we can only prove security provided that we pretend that H is a truly random function—i.e., in the random-oracle model. When we instantiate the hash function H with some concrete cryptographic hash function, such as SHA256, we hope that the resulting signature scheme is still secure. In practice, this approach works quite well.

One way to think about it is that if a signature scheme is secure in the random-oracle model, then the concrete signature scheme is in some sense secure against attacks that do not exploit the peculiarities of the hash function.

In the construction, we use:

- a trapdoor one-way permutation (Gen, F, I), and
- a hash function $H\colon \{0,1\}^* \to \mathcal{X}$, which we model as a random oracle in the security analysis.

Construction. We construct a digital-signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ as follows:

- Gen – Just run the key-generation algorithm for the trapdoor one-way permutation.
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$. Hash the message down to an element h of the input space $\mathcal{X}$ of the trapdoor one-way permutation using the hash function H. Then invert the trapdoor one-way permutation at that point:
  - Compute $h \leftarrow H(m)$.
  - Output $\sigma \leftarrow I(\mathsf{sk}, h)$.
- $\mathsf{Ver}(\mathsf{pk}, m, \sigma) \to \{0,1\}$.
  - Compute $h' \leftarrow H(m)$.

– Accept if $F(\mathsf{pk}, \sigma) = h'$.

Notice that the use of a hash function here is **critical** to security, since (in the random oracle) it means that forging a signature is as hard as inverting F on a random point in its co-domain. Without the hash function, forging a signature is only as hard as inverting F on an attacker-chosen point in its co-domain, which could be easy.

Correctness. For all $\lambda \in \mathbb{N}$, $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$, and $m \in \{0,1\}^*$, we have:

$$\begin{aligned}\mathsf{Ver}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) &= 1\{F(\mathsf{pk}, I(\mathsf{sk}, H(m))) = H(m)\} \\ &= 1\{I(\mathsf{sk}, F(\mathsf{pk}, I(\mathsf{sk}, H(m)))) = I(\mathsf{sk}, H(m))\}\end{aligned}$$

and by correctness of the trapdoor one-way permutation:

$$= 1\{I(\mathsf{sk}, H(m)) = I(\mathsf{sk}, H(m))\} = 1.$$

Security. The intuition here is that if the adversary cannot invert F, it cannot find the preimage of $H(m)$ under F for any message on which it has not seen a signature. See Boneh-Shoup Chapter 13.3 for the full security analysis.

2 *The RSA construction: Forward direction*

The algorithms for key-generation and for evaluating the RSA permutation in the forward direction are not too complicated.

In what follows, we present RSA with public exponent $e = 5$. The same construction works with many other choices of e, just by replacing all of the "5"s below with some other small prime: 3, 7, 13, etc. A popular choice of the public exponent e in practice is $e = 2^{16} + 1$. The complexity of computing the RSA function in the forward direction scales with the size of e, so we prefer small choices of e.

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$.
  - Sample two random λ-bit primes p and q such that $p \equiv q \equiv 4 \pmod 5$.
  - Set $N \leftarrow p \cdot q$.
  - Output $\mathsf{sk} \leftarrow (p, q)$, and $\mathsf{pk} = N$.
- $F(\mathsf{pk} = N, x) \to y$.
  - The input space for the RSA function is $\mathcal{X} = \mathbb{Z}_N^*$—the set of elements in $\{0, 1, 2, \ldots, N-1\}$ relatively prime to N.

In practice, we usually take the bitlength of primes to be $\lambda = 1024$ or $\lambda = 2048$.

Standard RSA implementations require the weaker condition that the public exponent e shares no prime factors with $p - 1$ and $q - 1$. Using the stronger condition here simplifies the inversion algorithm.

– Output $y \leftarrow x^5 \bmod N$.

Remark 2.1. The key-generation algorithm relies on us being able to sample large random primes. One perhaps surprising fact is that there are many many large primes. In particular, if you pick a random λ-bit number, the probability that it is prime is roughly $1/\lambda$.

For more on this, look up the Prime Number Theorem.

We can sample a random λ-bit prime by just picking random integers in the range $[2^\lambda, 2^{\lambda+1})$ until we find a prime. We can test for primality in $\approx \lambda^4$ time using the Miller-Rabin primality test. We also need that there are infinitely many primes congruent to 4 mod 5, but fortunately there are. Generating RSA keys is expensive—it can take a few seconds even on a modern machine.

Notice that computing the RSA function in the forward direction is relatively fast: it just requires three multiplications modulo a 2048-bit number N. That is, to compute $x^5 \bmod N$, we compute:

$$(x^2)^2 \cdot x = x^5 \mod N.$$

Before describing the RSA inversion algorithm, we discuss why the RSA trapdoor one-way permutation should be hard to invert without the secret key.

2.1 *Why should the RSA function be hard to invert?*

To invert the RSA function, the attacker's is effectively given a value $y \xleftarrow{R} \mathbb{Z}_N$ and must find a value x such that $x^5 = y \bmod N$. Or, put another way, the attacker's task is essentially the following:

- **Given:** A polynomial $p(X) := X^5 - y \in \mathbb{Z}_N[X]$, for $y \xleftarrow{R} \mathbb{Z}_N$.
- **Find:** A value $x \in \mathbb{Z}_N$ such that $p(x) = 0 \in \mathbb{Z}_N$.

So the attacker must find the root of a polynomial modulo a composite integer N.

The premise of RSA-style cryptosystems is that we only know of essentially two ways to find roots of polynomials modulo N:

- **Factor N into primes** and find a root modulo each of the primes. (We will say more on this in a moment.) Since the best algorithms for factoring run in time roughly $2^{\sqrt[3]{\log N}} = 2^{\sqrt[3]{\lambda}}$, this approach is infeasible at present without knowing the factorization of N.

In **??** we present a factoring algorithm that runs in sub-exponential time $2^{\sqrt{\log N \log\log N}}$.

- **Find a root over the integers** and reduce it modulo N. For example, it is easy to find a root of polynomials such as:

Actually, it suffices to find a root over the rational numbers, but the distinction isn't important here.

$$\begin{aligned} X + 4 &= 3 && \bmod N, \\ X + 2Y &= 5 && \bmod N, \\ X^2 &= 9 && \bmod N, \text{ and} \\ X^2 - 3x + 2 = (X-2)(X-1) &= 3 && \bmod N. \end{aligned}$$

When $y \xleftarrow{R} \mathbb{Z}_N$, the probability that y is a perfect 5-th power, and thus that there is an integral root to $X^5 - y$, is $\sqrt[5]{N}/N \approx 2^{-4\lambda/5}$, which is negligible in the security parameter λ. So solving this equation over the integers is a dead end.

There are many clever attacks for solving polynomial equations modulo composites that work in certain special cases, but for most purposes these are the two known attacks.

Is inverting the RSA function as hard as factoring the modulus? No one knows—the question has been open since the invention of RSA. We do know that finding roots of certain polynomial equations, such as $p(X) := X^2 - y \bmod N$ for random $y \xleftarrow{R} \mathbb{Z}_N$ *is* as hard as factoring the modulus N. But for RSA-type polynomials, the answer is unclear.

3 *The RSA construction: Inverse direction*

To understand how the inversion algorithm works, we will need some number-theoretic tools.

3.1 *Tools from number theory*

For a natural number N, let $\phi(N)$ denote the number of integers in $\mathbb{Z}_N = \{1, 2, 3, \ldots, N\}$ that are relatively prime to N. When p is prime $\phi(p) = p - 1$. The function $\phi(\cdot)$ is called *Euler's totient function*.

Two natural numbers are *relatively prime* if they share no prime factors.

When $N = pq$ is the product of two distinct primes, $\phi(N) = (p-1)(q-1)$. That is so because all numbers in $\mathbb{Z}_N$ are relatively prime to N except N and the multiples of p and q:

$$p, 2p, 3p, \ldots, (q-1)p, \quad q, 2q, 3q, \ldots, (p-1)q.$$

So there are $N - (q-1) - (p-1) - 1 = (p-1)(q-1)$ numbers in $\mathbb{Z}_N$ relatively prime to N.

Theorem 3.1 (Euler's Theorem)**.** *Let N be a natural number. Then for all $a \in \mathbb{Z}_N^*$,*

$$a^{\phi(N)} = 1 \mod N.$$

Proof. Consider the sets $\mathbb{Z}_N^*$ and $\{ax \bmod N \mid x \in \mathbb{Z}_N^*\}$. These sets are equal, so the product of the elements in the two sets is equal. Let $X \leftarrow \prod_{x \in \mathbb{Z}_N^*} x \bmod N$. Then

$$X = a^{\phi(N)} X \bmod N \quad \Rightarrow \quad 1 = a^{\phi(N)} \bmod N.$$

□

Lemma 3.2. *Let p and q be distinct primes congruent to 4 modulo 5. Define the integer $d = \frac{\phi(N)-4}{5} + 1$. Then $5d \equiv 1 \bmod \phi(N)$.*

Proof. Observe that

$$p \equiv 4 \bmod 5 \quad \Rightarrow \quad \phi(N) - 4 \equiv 0 \bmod 5,$$

so $\frac{\phi(N)-4}{5}$ is an integer and thus d is well defined. Then $5d = \phi(N) - 4 + 5 = 1 \bmod \phi(N)$. □

3.2 *Inverting the RSA function*

With the number theory out of the way, we can now describe how to invert the RSA function. All we have to do is to show how to compute a fifth root of y mod N.

- $I(\mathsf{sk}, y) \to x$.
  - The secret key sk consists of the prime factors p, q of N. Recall that $\phi(N) = (p-1)(q-1)$.
  - Compute the integer $d \leftarrow \frac{\phi(N)-4}{5} + 1$, as in Lemma 3.2.
  - Return y^d mod N.

We sometimes call d the *private exponent* in RSA.

It is not obvious why the inversion algorithm is correct. Say that $y = x^5$ mod N. Then:

$$\begin{aligned}
y^d &= (x^5)^d && \bmod N && \\
&= x^{5d} && \bmod N && \\
&= x^{k\cdot\phi(N)+1} && \bmod N, && \text{for some } k \in \mathbb{Z}, \text{ by Lemma 3.2} \\
&= x \cdot (x^{\phi(N)})^k && \bmod N && \\
&= x && \bmod N, && \text{by Theorem 3.1.}
\end{aligned}$$

We could write $5d = k\phi(N) + 1$ because from Lemma 3.2, we know that $5d \equiv 1 \bmod \phi(N)$.

Using other public exponents. For our RSA-inversion algorithm to work, we need only to compute the multiplicative inverse e modulo $\phi(N)$. That is, we need to compute an integers d such that $ed \equiv 1 \bmod \phi(N)$. Such an inverse always exists when e and $\phi(N) = (p-1)(q-1)$ are relatively prime. RSA implementations typically use the extended Euclidean algorithm to compute the multiplicative inverse of e modulo $\phi(N)$. That algorithm is more general, but the one we used in Lemma 3.2 is simpler and is self-contained.

Inverting RSA is easy on a negligible fraction of points. Recall the RSA is If the preimage under the RSA function of a point y is very very small, then If $x < N^{1/5}$, then computing x given $y = x^5$ mod N is *easy*.

Is inverting RSA as hard as factoring the modulus N? The inversion algorithm we showed here requires knowing the prime factors of the modulus N. Inverting RSA is thus *no harder than* factoring N.

Is inverting RSA *as hard as* factoring N? In particular, if we have an efficient algorithm $\mathcal{A}$ that inverts RSA, can we use $\mathcal{A}$ to factor the modulus N? No one knows!

Most cryptographers, I would guess, believe that inverting the RSA function is as hard as factoring. But for all we know, it could be that computing fifth roots modulo N is *easier* than factoring the modulus.

References

Bellare, Mihir and Phillip Rogaway. "Random oracles are practical: A paradigm for designing efficient protocols". In: *ACM Conference on Computer and Communications Security*. 1993.

Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.