The RSA Cryptosystem

Notes by Henry Corrigan-Gibbs

MIT - 6.5610
Lecture 13 (March 20, 2023)

> ***Warning:*** This document is a rough draft, so it may contain bugs. Please feel free to email me with corrections.

Outline

- Historical notes
- Trapdoor one-way permutations
  - Definition
  - Applications
- The RSA construction
  - Forward direction
  - Comments on security
  - Inverse direction

Background: RSA

1974: Ralph Merkle introduced public key exchange in an 1974 undergraduate project report at Berkeley [4]. He gave a key-exchange protocol based on one-way functions in which the honest parties run in time n and the best attack runs in time $\Omega(n^2)$.

1976: Diffie and Hellman, in their *New Directions* paper [1], defined public key exchange, public-key encryption, and digital signatures. They constructed a key-exchange scheme from discrete log with conjectured security against all poly-time adversaries: honest parties run in time n, attacker runs in superpolynomial time.

1977: Rivest, Shamir, and Adleman (RSA) [2, 5] give the *first* construction of public-key encryption and digital signatures from a problem related to the hardness of factoring integers.

Later results from Lamport, Merkle, Naor and Yung, and others showed that it is possible to build digital-signature schemes from one-way functions alone—i.e., just from standard hash functions. Today, we still do not know how to construct public-key encryption or key exchange from one-way functions.

2011: Google's HTTPS servers stop using RSA-based key exchange by default. Instead, they use RSA-based key exchange only for backwards compatibility with old clients. (Most HTTPS servers today still use RSA for digital signatures to authenticate the messages in a Diffie-Hellman key exchange.)

Why study RSA? The RSA cryptosystem is interesting 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.[1]

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

Trapdoor one-way permutations

Definition

RSA implements a *trapdoor one-way permutation* ("trapdoor OWP"), which we will now define.

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

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

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.

Applications of trapdoor OWPs

Before describing how to construct a trapdoor one-way permutation, we give some applications of the primitive. Constructions of public-key encryption and digital signatures are almost immediate from trapdoor one-way functions.

Unauthenticated key exchange

We construct a key-exchange protocol that is secure against passive attacks. In particular, we use:

- a trapdoor OWP (Gen, F, I), and
- a hash function $H\colon \mathcal{X} \to \{0,1\}^n$, which we model as a random oracle.
- Alice generates a keypair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}()$ and sends pk to Bob.
- Bob samples $k \xleftarrow{R} \mathcal{X}$ and sends $y \leftarrow F(\mathsf{pk}, k)$ to Alice. Bob uses $H(k)$ as his shared secret with Alice.
- Alice computes $k' \leftarrow I(\mathsf{sk}, y)$ and uses $H(k')$ as her shared secret with Bob.

Correctness. We have $k' = I(\mathsf{sk}, y) = I(\mathsf{sk}, F(\mathsf{pk}, k)) = k$, so $H(k) = H(k')$ and Alice and Bob end up with the same shared secret key.

Security. Showing weak security—that a passive eavesdropper cannot guess the shared secret—is almost immediate. For simplicity here, let's remove the hash function in the above construction. Then the passive attacker sees (pk, y) and her job is to produce x such that $F(\mathsf{pk}, x) = y$. This is exactly the task of inverting the trapdoor one-way permutation F! Therefore if F is secure, no efficient adversary can break passive security of this key-exchange mechanism.

To get strong security against passive attacks—i.e., to show that no passive attacker can even distinguish the shared secret from random—we just hash the shared key that both parties hold. In the

See Boneh-Shoup, Chapter 10.2.1 for the full security analysis.

Digital signatures

This construction is called "full-domain hash," and it follows the "hash-and-sign" paradigm that Yael discussed in her lecture on hash functions.

We use:

- a trapdoor OWP (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 OWP.
- $\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$. Hash the message down to an element h of the input space $\mathcal{X}$ of the trapdoor OWP using the hash function H. Then invert the trapdoor OWP 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.

In fact, inverting F at attacker-chosen points *is* easy when F is the RSA function.

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.

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 $e = 3$. The same construction works with many other choices of e, just by replacing all of the "3"s below with some other small prime: 7, 13, etc. A popular choice of the public exponent d 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 $p \equiv q \equiv 5 \pmod 6$.
  - 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 $\mathcal{X} = \mathbb{Z}_N = \{0, 1, 2, 3, \ldots, N-1\}$.
  - Output $y \leftarrow x^3 \bmod N$.

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

Standard RSA implementations require the weaker condition that $p \equiv q \equiv 2 \bmod 3$. Using the stronger condition here simplifies the inversion algorithm.

Remark. The key-generation algorithm relies on us being able to sample large random primes. 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 5 mod 6, but fortunately there are.

Notice that computing the RSA function in the forward direction is relatively fast: it just requires two multiplications modulo a 2048-bit number N. In contrast, executing a Diffie-Hellman key exchange in $\mathbb{Z}_p^*$ requires hundreds of multiplications modulo a 2048-bit number, depending on the order of the group.

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

To be completely precise, we should write that the input space of the RSA function is $\mathbb{Z}_N^*$, which is the set of numbers in $\mathbb{Z}_N$ that are relatively prime to the modulus N. Since we only ever sample random numbers from $\mathcal{X}$, the probability that a random sample from $\mathbb{Z}_N$ is not also in $\mathbb{Z}_N^*$ is

$$\begin{aligned} 1 - \frac{|\mathbb{Z}_N^*|}{|\mathbb{Z}_N|} &= 1 - \frac{(p-1)(q-1)}{N} \\ &= 1 - \frac{N - p - q + 1}{N} \\ &\leq \frac{p+q}{N} \\ &\approx 2/\sqrt{N} \\ &\approx 2^{-\lambda} \end{aligned}$$

which is negligible in the security parameter λ. In other words, you are as likely to hit one of these "bad" elements as you are to guess a prime factor of N.

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 a value x such that $x^3 = y \bmod N$. Or, put another way, the attacker's task is essentially the following:

- **Given:** A polynomial $p(X) := X^3 - y \in \mathbb{Z}_N[X]$, 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.
- **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 cube, and thus that there is an integral root to $X^3 - y$, is $\sqrt[3]{N}/N \approx 2^{-4\lambda/3}$, 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 $y \xleftarrow{R} \mathbb{Z}_N$ *is* as hard as factoring the modulus N. But for RSA-type polynomials, the answer is unclear.

The RSA construction: Inverse direction

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

Number-theoretic preliminaries

Theorem 1 (Fermat's Little Theorem)**.** *For every prime p and integer $a \in \mathbb{Z}_p^*$, $a^{p-1} = 1 \pmod{p}$.*

Proof. Consider the sets $\mathbb{Z}_p^*$ and $\{ax \bmod p \mid x \in \mathbb{Z}_p^*\}$. These sets are equal, so the product of the elements in the two sets is equal:

$$(p-1)! = a^{p-1}(p-1)! \bmod p \quad \Rightarrow \quad 1 = a^{p-1} \bmod p.$$

□

Lemma 2. *If p is a prime congruent to 5 modulo 6, then for all $a \in \mathbb{Z}_p^*$, at least one element r of $\{a^{\frac{p+1}{6}}, -a^{\frac{p+1}{6}}\}$ is such that $r^3 = a \bmod p$.*

Proof. First, observe that since $p = 5 \bmod 6$, $(p+1)/6$ is an integer, so exponentiation by $(p+1)/6$ is well defined.

Let $r = a^{\frac{p+1}{6}} \bmod p$. Then

$$r^3 = (a^{\frac{p+1}{6}})^3 = a^{\frac{p+1}{2}} = a^{\frac{p-1}{2}} a.$$

Since $(a^{\frac{p-1}{2}})^2 = 1$ (by Fermat's Little Theorem), $a^{\frac{p-1}{2}} \in \{-1, 1\}$, so $r^3 \bmod p \in \{-a, a\}$. If $r^3 = a \bmod p$, we are done. Otherwise, $(-r)^3 = -r^3 = a \bmod p$, and we are done. □

The following theorem dates back to the second century B.C.E. [3], and was used in the context of constructing calendars:

Keith Conrad has nice lecture notes on the Chinese Remainder Theorem, and we draw our treatment of the theorem from there.

Theorem 3 (Chinese Remainder Theorem (CRT))**.** *Let p and q be distinct primes. For all integers a and b, the pair of congruences*

$$x = a \pmod{p} \qquad \text{and} \qquad x = b \pmod{q}$$

has a unique and efficiently computable solution modulo pq.

The full theorem is more general—it handles the case of more than two moduli, and works when the moduli are relatively prime (rather than primes).

One consequence of the CRT is that we can always represent an element $x \in \mathbb{Z}_N$ for $N = pq$, with p and q distinct primes, as pair a pair $(x_p, x_q) \in \mathbb{Z}_p \times \mathbb{Z}_q$. Adding and multiplying elements in this "CRT representation" corresponds to adding and multiplying elements modulo $\mathbb{Z}_N$.

So one implication of the CRT is that whenever we want to solve a polynomial equation modulo $N = pq$, we can solve the problem modulo p and modulo q, and then we can use the CRT to reconstruct a solution modulo N. This idea shows up all over cryptography, so it's an important one to understand.

Proof idea. One approach to proving the theorem is to define

$$p_1 = p^{-1} \pmod{q} \qquad \text{and} \qquad q_1 = q^{-1} \pmod{p}.$$

Since p and q are distinct primes, such integers must exist. Then the solution is:

$$x = aq_1q + bp_1p \pmod{pq}.$$

We can compute p_1 efficiently as $p_1 = p^{q-2} \bmod q$. By Fermat's Little Theorem, $p_1 \cdot p = p^{q-1} = 1 \bmod q$, so p_1 is the multiplicative inverse of p modulo q as desired.

Notice that $x = a \bmod p$ and $x = b \mod q$.

The last part is to show uniqueness. One way to show that is to argue that for any two solutions x, x', their difference must be $x - x' = 0 \bmod pq$, which implies that they are congruent modulo pq. □

Inverting the RSA function

With all of those preliminaries 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 cube root of y mod N. Our plan will be to compute a cube root of y modulo the prime factors p and q of N individually, and then reconstruct the solution modulo N.

- $I(\mathsf{sk}, y) \to x$.
  - Recall that the secret key sk consists of the factors of $N = pq$. We want to solve:

    $$x^3 = y \pmod{p} \quad \text{and} \quad x^3 = y \pmod{q}.$$

  - Use Lemma 2 to find the cube roots (x_p, x_q) of y modulo (p, q).
  - Then we need to solve:

    $$x = x_p \pmod{p} \quad \text{and} \quad x = x_q \pmod{q}.$$

    Use the Chinese Remainder Theorem (Theorem 3) to find an $x \in \mathbb{Z}_N$ (where $N = pq$) that satisfies these congruences.
  - Return x.

The standard way to describe RSA inversion uses the extended Euclidean algorithm. Since that algorithm requires some time to describe, I went with this simpler version.

To argue that $x^3 = y \bmod N$: By construction, we have that $x^3 = y \bmod p$ and $x^3 = y \bmod q$. Since, by the Chinese Remainder Theorem, there is a unique $y \in \mathbb{Z}_N$ that satisfies these congruences, we must have computed the cube root of the "right y" and we are done.

Inversion is hard without knowing the factorization of N. The first step of the inversion algorithm is to reduce the equation $x^3 = y \bmod N$ modulo each of the prime factors of N. Without knowing the factors

of N, it is not possible to execute this step. It could be that there exists some other algorithm for computing cube roots modulo N without having to compute it modulo each of the primes, but we know of no such algorithm.

References

[1] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 1976.

[2] Martin Gardner. A new kind of cipher that would take millions of years to break. *Scientific American*, 237(8):120–124, 1977.

[3] Shen Kangsheng. Historical development of the Chinese Remainder Theorem. *Archive for History of Exact Sciences*, pages 285–305, 1988.

[4] Ralph C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978. See original project report at `https://www.ralphmerkle.com/1974/`.

[5] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.