CS 70 Discrete Mathematics and Probability Theory
Spring 2022 Koushik Sen and Satish Rao

Note 7

This note is partly based on Section 1.4 of "Algorithms," by S. Dasgupta, C. Papadimitriou and U. Vazirani, McGraw-Hill, 2007.

Public Key Cryptography

In this note, we discuss a very nice and important application of modular arithmetic: the *RSA public-key cryptosystem*, named after its inventors Ronald Rivest, Adi Shamir and Leonard Adleman.

Cryptography is an ancient subject that really blossomed into its modern form at the same time[1] as the other great revolutions in the general fields of information science/engineering. The basic setting for cryptography is typically described via a cast of three characters: Alice and Bob, who with to communicate confidentially over some (insecure) link, and Eve, an eavesdropper who is listening in and trying to discover what they are saying. Let's assume that Alice wants to transmit a message x (written in binary) to Bob. She will apply her *encryption function* E to x and send the encrypted message $E(x)$ over the link; Bob, upon receipt of $E(x)$, will then apply his *decryption function* D to it and thus recover the original message: i.e., $D(E(x)) = x$.

Since the link is insecure, Alice and Bob have to assume that Eve may get hold of $E(x)$. (Think of Eve as being a "sniffer" on the network.) Thus ideally we would like to know that the encryption function E is chosen so that just knowing $E(x)$ (without knowing the decryption function D) doesn't allow one to discover anything about the original message x.

For centuries cryptography was based on what are now called *private-key* protocols. In such a scheme, Alice and Bob meet beforehand and together choose a secret codebook, with which they encrypt all future correspondence between them. (This codebook plays the role of the functions E and D above.) Eve's only hope then is to collect some encrypted messages and use them to at least partially figure out the codebook.

Public-key schemes such as RSA, first invented in the 1970s, are significantly more subtle and tricky: they allow Alice to send Bob a message without ever having met him before! This almost sounds impossible, because in this scenario there is a symmetry between Bob and Eve: why should Bob have any advantage over Eve in terms of being able to understand Alice's message? The central idea behind the RSA cryptosystem is that Bob is able to implement a *digital lock*, to which only he has the key. Now by making this digital lock public, he gives Alice (or, indeed, anybody else) a way to send him a secure message which only he can open.

Here is how the digital lock is implemented in the RSA scheme. Each person has a *public key* known to the whole world, and a *private key* known only to him- or herself. When Alice wants to send a message x to Bob, she encodes it using Bob's public key. Bob then decrypts it using his private key, thus retrieving x. Eve is welcome to see as many encrypted messages for Bob as she likes, but she will not be able to decode them (under certain basic assumptions explained later in this Note).

[1] And in reality, involving some of the same cast of characters. Both Alan Turing and Claude Shannon were active in codebreaking during WW2 and Shannon had a classic paper that is considered the birth of the information-theoretic understanding of secrecy and dovetails with his other more famous paper that gave rise to the modern information-theoretic view of communication. Both cryptography and communication weave together information, computation, and randomness in surprising ways. In 70, you just get a tiny taste of these spectacularly beautiful fields.

When it comes to something like RSA, there are different ways that help different people understand the scheme. In this note, we will first just walk through the scheme and its analysis. Along the way, we'll discuss what happens when one raises numbers to powers in modulo arithmetic. However, at the end of this note, we will talk about how anyone could have come up with the RSA scheme. After all, the understanding of exponentiating in modulo arithmetic predates RSA historically. It turns out that with the background understanding and perspective provided by the "vector view" of the Chinese Remainder Theorem and Fermat's Little Theorem's understanding of exponentiation, the RSA scheme is quite natural.

The RSA scheme is based heavily on modular arithmetic. Let p and q be two large primes (typically having, say, 512 bits each), and let $N = pq$. We will think of messages to Bob as numbers modulo N, excluding the trivial values 0 and 1. (Larger messages can always be broken into smaller pieces and sent separately.)

Also, let e be any number that is relatively prime to $(p-1)(q-1)$. (Typically e is chosen to be a small value such as 3.) Then Bob's *public key* is the pair of numbers (N,e). This pair is published to the whole world. (Note, however, that the numbers p and q are *not* public; this point is crucial and we will return to it below.)

What is Bob's private key? This will be the number d, which is the *inverse* of e mod $(p-1)(q-1)$. (This inverse is guaranteed to exist because e and $(p-1)(q-1)$ are coprime.)

We are now in a position to describe the encryption and decryption functions:

- **[Encryption]:** When Alice wants to send a message x (assumed to be an integer mod N) to Bob, she computes the value $E(x) \equiv x^e \pmod{N}$ and sends this to Bob.
- **[Decryption]:** Upon receiving the value $y = E(x)$, Bob computes $D(y) \equiv y^d \pmod{N}$; this will be equal to the original message x.

Example: Let $p = 5$, $q = 11$, and $N = pq = 55$. (In practice, p and q would be much larger.) Then we can choose $e = 3$, which is relatively prime to $(p-1)(q-1) = 40$. Thus Bob's public key is $(55,3)$. His private key is $d \equiv 3^{-1} \pmod{40} \equiv 27$. For any message x that Alice (or anybody else) wishes to send to Bob, the encryption of x is $y \equiv x^3 \pmod{55}$, and the decryption of y is $x \equiv y^{27} \pmod{55}$. So, for example, if the message is $x = 13$, then the encryption is $y = 13^3 \equiv 52 \pmod{55}$, and this is decrypted as $52^{27} \equiv 13 \pmod{55}$.

How do we know that this scheme works? We need to check that Bob really does recover the original message x. The following theorem establishes this fact.

Theorem 7.1:

Under the above definitions of the encryption and decryption functions E and D, we have $D(E(x)) \equiv x \pmod{N}$ for every possible message $x \in \{0,1,\ldots,N-1\}$.

The proof of this theorem makes use of a standard fact from number theory known as *Fermat's Little Theorem*, which tells us that exponentiation is periodic when done modulo a prime, and that period is one less than the prime in question. Precisely, it says:

Theorem 7.2: **[Fermat's Little Theorem]** For any prime p and any $a \in \{1,2,\ldots,p-1\}$, we have $a^{p-1} \equiv 1 \pmod{p}$.

Proof. Let S denote the set of non-zero integers mod p, i.e., $S = \{1,2,\ldots,p-1\}$. Consider the sequence of numbers $a, 2a, 3a, \ldots, (p-1)a \bmod p$. We already saw in the previous Lecture Note that, whenever $\gcd(p,a) = 1$ (i.e., p,a are coprime, which certainly holds here since p is prime) these numbers are all distinct. Therefore, since none of them is zero, and there are $p-1$ of them, they must include each element

of S exactly once. Therefore, the set of numbers

$$S' = \{a \pmod p, 2a \pmod p, \ldots, (p-1)a \pmod p\}$$

is exactly the same as S (just in a different order)!

Now suppose we take the product of all numbers in S, mod p. Clearly, this product is

$$1 \times 2 \times \cdots \times (p-1) \equiv (p-1)! \pmod p. \tag{1}$$

On the other hand, what if we take the product of all the numbers in S'? Clearly this is

$$a \times 2a \times \cdots \times (p-1)a \equiv a^{p-1}(p-1)! \pmod p. \tag{2}$$

But from our observation in the previous paragraph that the sets of numbers in S and in S' are exactly the same (mod p), the products in (1) and (2) must in fact be equal mod p. Hence we have

$$(p-1)! \equiv a^{p-1}(p-1)! \pmod p. \tag{3}$$

Finally, since p is prime, we know that every non-zero integer has an inverse mod p, and therefore $(p-1)!$ has an inverse mod p. Hence we can multiply both sides of (3) by the inverse of $(p-1)!$ to get $a^{p-1} \equiv 1 \pmod p$, as required. □

Armed with Fermat's Little Theorem, we can now go back and prove the correctness of RSA.

Proof of Theorem 7.1. To prove the statement, we have to show that

$$(x^e)^d \equiv x \pmod N \qquad \text{for every } x \in \{0, 1, \ldots, N-1\}. \tag{4}$$

Let's consider the exponent, which is ed. By definition of d, we know that $ed \equiv 1 \pmod{(p-1)(q-1)}$; hence we can write $ed = 1 + k(p-1)(q-1)$ for some integer k, and therefore

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x = x(x^{k(p-1)(q-1)} - 1). \tag{5}$$

Looking back at equation (4), our goal is to show that this last expression in equation (5) is equal to 0 mod N for every x.

Now we claim that the expression $x(x^{k(p-1)(q-1)} - 1)$ in (5) is divisible by p. To see this, we consider two cases:

Case 1: *x is not a multiple of p.* In this case, since $x \not\equiv 0 \pmod p$, we can use Fermat's Little Theorem to deduce that $x^{p-1} \equiv 1 \pmod p$, and hence $x^{k(p-1)(q-1)} - 1 \equiv 0 \pmod p$, as required.

Case 2: *x is a multiple of p.* In this case the expression in (5), which has x as a factor, is clearly divisible by p.

By an entirely symmetrical argument, $x(x^{k(p-1)(q-1)} - 1)$ is also divisible by q. Therefore, it is divisible by both p and q, and since p and q are primes it must be divisible by their product, $pq = N$. But this implies that the expression is equal to 0 mod N, which is exactly what we wanted to prove. □

CRT Proof of Theorem 7.1. A closely related proof can be obtained using the Chinese remainder theorem. That is,

$$x^{ed} = x^{k(p-1)(q-1)+1} = \left(x^{k(q-1)}\right)^{(p-1)} x \equiv x \pmod{p}$$

for both $x \not\equiv 0 \pmod{p}$ by Fermat's Theorem and for $x = 0 \pmod{p}$ by inspection. Similarly, $x^{ed} \equiv x \pmod{q}$. By the uniqueness property established in Chinese remainder theorem, only $x \pmod{pq}$ satisfies these two equations. □

So we have seen that the RSA protocol is *correct*, in the sense that Alice can encrypt messages in such a way that Bob can reliably decrypt them again. But how do we know that it is *secure*, i.e., that Eve cannot get any useful information by observing the encrypted messages? The security of RSA hinges upon the following basic assumption:

Given N, e and $y \equiv x^e \pmod{N}$, there is no efficient algorithm for determining x.

This assumption is quite plausible. How might Eve try to guess x? She could experiment with all possible values of x, each time checking whether $x^e \equiv y \pmod{N}$; but she would have to try on the order of N values of x, which is completely unrealistic if N is a number with (say) 512 bits. Alternatively, she could try to factor N to retrieve p and q, and then figure out d by computing the inverse of e mod $(p-1)(q-1)$; but this approach requires Eve to be able to *factor* N into its prime factors, a problem which is believed to be impossible to solve efficiently for large values of N. We should point out that the security of RSA has not been formally proved: it rests on the assumptions that breaking RSA is essentially tantamount to factoring N, and that factoring is hard.

We close this note with a brief discussion of implementation issues for RSA. Since we have argued that breaking RSA is impossible because *factoring* would take a very long time, we should check that the computations that Alice and Bob themselves have to perform are much simpler, and can be done efficiently.

There are really only two non-trivial things that Alice and Bob have to do:

1. Bob has to find prime numbers p and q, each having many (say, 512) bits.
2. Both Alice and Bob have to compute exponentials mod N. (Alice has to compute $x^e \pmod{N}$, and Bob has to compute $y^d \pmod{N}$.)

We briefly discuss the implementation of each of these tasks in turn.

To find large prime numbers, we use the fact that, given a positive integer n, there is an efficient algorithm that determines whether or not n is prime. (Here "efficient" means a running time of $O((\log n)^k)$ for some small k, i.e., a low-degree power of the *number of bits in* n. Notice the dramatic contrast here with factoring: we can tell efficiently whether or not n is prime, but in the case that it is not prime we cannot efficiently find its factors. The success of RSA hinges crucially on this distinction.) Given that we can test for primes, Bob just needs to generate some random integers n with the right number of bits, and test them until he finds two primes p and q. This works because of the following basic fact from number theory (which we will definitely not prove[2]), which says that a reasonably large fraction of positive integers are prime:

[2]Why don't we prove this? The reason is that standard proofs of this fact rely on complex analysis (i.e. calculus involving complex functions), and need to lean on the study of more advanced relatives of the transfer functions and s-impedances that you saw in EECS16B. Is it surprising that you need this kind of machinery to study prime numbers? This is simply another aspect of the interconnected beauty of mathematics. We need complex numbers to study the behavior of differential equations and electrical circuits — why should it be any surprise that they are needed to properly understand the prime numbers?

Theorem 7.3: **[Prime Number Theorem]** Let $\pi(n)$ denote the number of primes that are less than or equal to n. Then for all $n \geq 17$, we have $\pi(n) \geq \frac{n}{\ln n}$. (And in fact, $\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$.)

Setting $n = 2^{512}$, for example, the Prime Number Theorem says that roughly one in every 355 of all 512-bit numbers are prime. Therefore, if we keep picking random 512-bit numbers and testing them, we would expect to have to try only about 355 numbers until we find a prime.

We turn now to the second task above: modular exponentiation. This is actually something we have already discussed in the previous Lecture Note. Recall from that Note that we can compute an exponential expression $x^y \pmod N$ by repeated squaring, using a number of multiplications that is only $O(n)$, where n is the number of *bits* in y (i.e., the length of the binary representation of y). Since in the RSA application the exponents in the expressions that Alice and Bob need to compute (e, d, respectively) are both less than N, the number of multiplications needed is $O(\log N)$, since the number of bits in N is $O(\log N)$. Furthermore, since all multiplication is being done mod N, all multiplications involve numbers with at most $O(\log N)$ bits; and, as is well known from elementary school long multiplication, multiplying m-bit numbers takes $O(m^2)$ operations (and can actually done even faster: see CS170). Hence the total cost of Alice's and Bob's exponential computations is only $O((\log N)^3)$. Thus it is possible to work with very large numbers (having hundreds of bits) in practical implementations of RSA.

To summarize, then, in the RSA protocol Bob need only perform simple calculations such as multiplication, exponentiation and primality testing to implement his digital lock. Similarly, Alice and Bob need only perform simple calculations to lock and unlock the the message respectively—operations that any pocket computing device could handle. By contrast, to unlock the message without the key, Eve would have to perform operations like factoring large numbers, which (at least according to widely accepted belief) requires more computational power than all the world's most sophisticated computers combined! This compelling guarantee of security without the need for private keys explains why the RSA cryptosystem is such a revolutionary development in cryptography.