CS 70 Discrete Mathematics and Probability Theory
Summer 2019 Course Notes

Note 7

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

1 Introduction to Cryptography

In this note, we discuss the notion of *cryptography*, which is the study of how to securely send messages over potentially insecure channels. The basic setting for cryptography is typically described via a cast of three characters: Alice and Bob, who with to communicate confidentially, 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 m (written in binary) to Bob. She will apply her *encryption function* E to m and send the encrypted message $E(m)$ over the link; Bob, upon receipt of $E(m)$, will then apply his *decryption function* D to it and thus recover the original message: i.e., $D(E(m)) = m$.

Since the link is insecure, Alice and Bob have to assume that Eve may get hold of $E(m)$. (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 $E(m)$ (without knowing the decryption function D) doesn't allow one to discover anything about the original message m.

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, 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 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.

2 Private Key Cryptography

2.1 The XOR Operation

Our first example of a secure cryptography system will be a private-key protocol based off the XOR operation. This operation takes as inputs two bits, and outputs the bit 1 if and only if exactly one of its two inputs is a 1. This can also be expressed in the following truth table:

b_1	b_2	$b_1 \oplus b_2$
0	0	0
0	1	1
1	0	1
1	1	0

Given two equal length bit strings x and y, we can consider the *bit-wise XOR* of the two strings, $x \oplus y$, the ith bit of which is just the XOR of the ith bit of x with the ith bit of y. One particular property of the XOR operation is that it is its own inverse. Formally, we have

Lemma 7.1. *Let b_1 and b_2 be any bits. Then $(b_1 \oplus b_2) \oplus b_2 = b_1$.*

Sanity check! Write out the truth table for $(b_1 \oplus b_2) \oplus b_2$ and verify that it equals b_1.

This result generalizes to bit strings:

Corollary 7.1. *Let x and y be length n bit strings. Then $(x \oplus y) \oplus y = x$.*

Proof. Let x_i be the ith bit of x and y_i be the ith bit of y. The ith bit of $(x \oplus y) \oplus y$ is $(x_i \oplus y_i) \oplus y_i$; by Lemma 7.1, this is just x_i. Hence each bit of $(x \oplus y) \oplus y$ matches x, so we have $(x \oplus y) \oplus y = x$. □

2.2 One-Time Pad

From the properties in the previous section, we can define our first private-key protocol, known as the one-time pad protocol, or OTP for short. This and other protocols can be viewed as happening in three stages: setup (where keys are decided and disseminated), encryption, and decryption. We describe how OTP functions in these three stages below, assuming that Alice wants to securely send Bob a message encoded in a length n bit string.

Setup: Alice and Bob agree on a length n bit string p, known as the pad, and keep it secret.

Encryption: To encrypt a message m into a ciphertext c, Alice evaluates $E_p(m) = m \oplus p$.

Decryption: To decrypt a ciphertext c into a message m, Bob evaluates $D_p(c) = c \oplus p$.

A natural question to ask now is how do we evaluate how good of a protocol this is? What exactly do we want out of a cryptosystem? For this note, we will be focusing on two properties: *correctness*, meaning that Bob will in fact receive Alice's message after decryption, and *security*, meaning that Eve will not be able to determine what message Alice sent.

Sanity check! Why is a system without correctness not interesting? What about a system without security?

Theorem 7.1. *The OTP protocol is correct. That is, for any pad p and message m, $D_p(E_p(m)) = m$.*

Proof. This is an immediate consequence of Corollary 7.1. Indeed, since $E_p(m) = m \oplus p$ and $D_p(c) = c \oplus p$, we have $D_p(E_p(m)) = (m \oplus p) \oplus p$, which is just m by Corollary 7.1. □

There are many ways one might define security. For example, suppose that Eve could determine the first bit of m from $E(m)$ but nothing else. One could reasonably argue that this is secure or that it is not secure depending on the use case. Or alternatively, suppose that Eve could determine m from $E(m)$, but it would take her approximately 10^{20} years to do so. One could reasonably define this to be secure or not secure.[1]

[1] While 10^{20} years is a long time, it is possible that advances in computing can cut down on the required time significantly. Indeed, many systems considered secure around the advent of modern cryptography have needed to add additional security to keep up with the speed of potential adversaries.

As it turns out, the OTP protocol has the best kind of security you could hope for: if Eve knows nothing about the pad p, she gains no information about m from $E_p(m)$! Formally, we have the following theorem

Theorem 7.2. *Let m be any message and c be any ciphertext. Then there exists a pad p such that $E_p(m) = c$.*

Proof. Take $p = c \oplus m$. We then have that $E_p(m) = (c \oplus m) \oplus m$, which by Corollary 7.1 is just c. □

Why does Theorem 7.2 guarantee us security? Suppose that Eve has intercepted some ciphertext c, but does not know what the pad p used was. Given any message m, Theorem 7.2 tells us that there is a pad p that would make m encrypt to c, so every possible message is consistent with what Eve has seen! Thus, Eve has no reason to believe the message sent was one possibility over any other; in other words, she gains no information about m.

3 Public Key Cryptography

While the previous section gave us a system which was correct and perfectly secure, there are some issues with private key cryptography. By far the biggest is that Alice and Bob need some way to agree on their secret key such that no one else finds out about it. But if they have some secure way of sharing this key, they could just use that to share their messages in the first place!

This problem is even worse in the case of the one-time pad protocol. As suggested by the name, in order to maintain security, Alice and Bob can only use a pad one time. If they try using the same pad multiple times, it may leak some information about their messages to Eve. Thus, Alice and Bob need to invoke their secure key-sharing protocol every time they want to send messages to one another.

In order to avoid these issues, we will attempt to create a *public-key* protocol. In these sorts of protocols, we will replace the shared private key with two keys: one private one known only to Bob, and one public one publicized to everyone. When Alice wishes to send a message to Bob, she will encrypt it using his public key; Bob can then decrypt the message using his private key. Thus, anyone in the world can send Bob a message, but since only Bob knows the private key, he is the only one who can decrypt those ciphertexts.

3.1 RSA

As an example of a public-key protocol, we describe the RSA cryptosystem, named after its inventors Ronald Rivest, Adi Shamir, and Leonard Adleman. As with the one-time pad procedure, we consider RSA in three stages:

Setup: Bob generates two distinct primes p and q, as well as a number e relatively prime to $(p-1)(q-1)$. Bob calculates $d = e^{-1} \pmod{(p-1)(q-1)}$ and $N = pq$. Bob publicizes (N, e), and keeps d secret.

Encryption: Given Bob's public key (N, e), Alice encrypts m as $E_{N,e}(m) = m^e \pmod{N}$.

Decryption: Since Bob knows his private key d, he can decrypt c as $D_{N,d}(c) = c^d \pmod{N}$.

How big should p and q be? Since our encryption and decryption are done modulo N, we certainly need to make sure N is bigger than our message in order to have any hope at unique decryption. Thus, if we wish to send an n bit message, we should ensure that N is at least 2^n. Additionally, as we will see later, the security of this protocol gets better as p and q get larger. For current real-world applications, p and q are often chosen to be 512 bit numbers in order to have sufficient security.

3.2 RSA Correctness

As before, in order to show that RSA is an interesting protocol, we need to show that it is *correct*, meaning that Bob will always get the message Alice sent. The proof of this fact relies heavily on Fermat's Little Theorem, which we stated and proved in note 6.5. For reference, it is reproduced below:

Theorem 7.3. *Let p be a prime and $a \not\equiv 0 \pmod{p}$. Then $a^{p-1} \equiv 1 \pmod{p}$.*

Theorem 7.4. *The protocol described in Section 3.1 is correct. That is, for any choice of p, q, and e, and for any message m, we have that $D_{N,d}(E_{N,e}(m)) = m$.*

Proof. We first note that $D_{N,d}(E_{N,e}(m)) = (m^e)^d \pmod{N} = m^{ed} \pmod{N}$. In the setup phase we defined d to be $e^{-1} \pmod{(p-1)(q-1)}$, so $ed \equiv 1 \pmod{(p-1)(q-1)}$, and hence $ed = 1 + k(p-1)(q-1)$ for some integer k.

Let's now consider what happens if we take m^{ed} modulo p. Plugging in the above value for ed, we have $m^{ed} = m^{1+k(p-1)(q-1)} = m \cdot (m^{p-1})^{k(q-1)}$. If $m \equiv 0 \pmod{p}$, this will evaluate to zero mod p. Otherwise, Theorem 7.3 tells us that m^{p-1} is equivalent to 1 modulo p, and hence the whole thing will just simplify to m. In either case, we have that $m^{ed} \equiv m \pmod{p}$.

We can use exactly the same argument to show that $m^{ed} \equiv m \pmod{q}$. Thus, in order to determine the value of $m^{ed} \pmod{pq}$, we simply need to find a value which is equivalent to m mod p and mod q. Setting $m^{ed} \equiv m \pmod{pq}$ certainly satisfies both these equivalences — and by the Chinese Remainder Theorem from note 6.5, this is the *only* value modulo pq that satisfies both of them. Thus, we must in fact have that $m^{ed} \equiv m \pmod{N}$; since we chose p and q such that $N > m$, this means that $m^{ed} \bmod N = m$, and hence we indeed will correctly decrypt.[2]

□

3.3 RSA Efficiency

In addition to *correctness*, we should also make sure that our protocol is *efficient* for both Alice and Bob. For security reasons (as discussed in the next section), we will generally make p and q 512-bit numbers. Thus, in order to ensure that Alice and Bob can send their messages in a reasonable amount of time, we should make sure that all operations can be done in time related to the *number of bits* in p and q, rather than related to p and q themselves.

Looking at the protocol in Section 3.1, there are three non-trivial operations Alice and/or Bob have to perform: (1) Bob has to choose primes p and q, (2) Bob has to find the inverse of e mod $(p-1)(q-1)$, and (3) both Alice and Bob need to perform modular exponentiations.

For the first task, we will rely on the following theorem, though it is beyond our scope to prove it:

Theorem 7.5. [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, $\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1$.)*

What this tells us is that if we just choose a random 512-bit number, the chance of it being prime is approximately $\frac{1}{\ln(2^{512})} \approx \frac{1}{355}$, so we are likely to find a prime within a few hundred guesses. Note that for this, we also need a way of testing if a number is prime. Many algorithms exist for this task; you can search for "primality testing" if you are interested in reading more about them.

[2] One can also give a version of this proof that does not explicitly use CRT. Using similar methods to this proof, we can show that $m^{ed} - m$ is zero modulo both p and q, and hence is divisible by both. Since p and q are distinct primes, being divisible by both of them means you must be divisible by their product, so $m^{ed} - m \equiv 0 \pmod{N}$, and hence $m^{ed} \equiv m \pmod{N}$.

For the second point, we recall from Note 6 that the extended version of Euclid's GCD algorithm allows us to compute inverses, where the number of recursive calls made is within a constant factor of the number of bits in the number we wish to take the inverse of. Thus, this too can be done efficiently.

Finally, we consider the problem of modular exponentiation. We again go think back to Note 6 and recall the repeated squaring procedure, which allows us to calculate $x^y \pmod{m}$, where the number of recursive calls is the number of bits in y. Since our exponents will always be at most N (and indeed, at most $(p-1)(q-1)$), this will also be efficient even for very large values of p and q.

3.4 RSA Security

We now finally turn to the question of whether or not RSA is secure. At this point, we reach a surprising answer: we don't know. To date, no one has publicly found a proof that this system cannot be broken — but at the same time, many people have been working very hard to break it, and as far as we know, no one has succeeded! Formally, the security of RSA rests upon the following assumption:

Given N, e and $c = m^e \bmod N$, there is no efficient algorithm for determining m.

This assumption is quite plausible. How might Eve try to guess m? She could experiment with all possible values of m, each time checking whether $m^e = c \bmod N$; but she would have to try on the order of N values of m, 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 $e \bmod (p-1)(q-1)$; 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 this does not constitute a *proof* that RSA is secure (indeed, there may be a very clever way of breaking it that we didn't consider here!), but it does strongly suggest the scheme is secure. And indeed, enough people believe RSA to be secure that it forms the backbone of much of modern e-commerce and online security!

As a side note, it turns out that while no one has yet come forward with a way to break RSA using a *classical* computer, it is known how to factor numbers, and thus how to break RSA, using a *quantum* computer. The details of this are well beyond the scope of our course, but it does suggest that RSA's days are numbered. While current quantum computers are too small to realistically break 512-bit RSA, in the future that may no longer be the case, requiring us as computer scientists to devise new methods to ensure online security.

4 RSA: Beyond Textbook

What we discussed in the previous section is often referred to as *textbook RSA* — it distills the essence of what the protocol is doing in such a way that it is relatively easy to describe and prove properties about. However, uses of RSA in the real world tend to look somewhat different from what we have described here. In this section, we describe two real-world attacks that get around the security of RSA, along with an alternate application of the RSA protocol.

4.1 Replay Attacks

For our first attack against textbook RSA, consider the following situation: you wish to send your credit card information to Amazon (or some other online retailer) so as to make a purchase. You have Amazon's public key, so you can encrypt your credit card number m as $E_{N,e}(m)$ and send it along.

But now what happens if Eve is eavesdropping on your message to Amazon? She now has access to $E_{N,e}(m)$, so when Amazon asks for her credit card number, she can reply with that — effectively allowing her to use your credit card despite not explicitly knowing its number!

This type of attack is what is known as a *replay attack*, since an eavesdropper is replaying a message they saw before in the hopes of passing it off as real. In order to avoid these sorts of issues, we can add randomness to the end of our messages. Formally, instead of sending $E_{N,e}(m)$, we send $E_{N,e}(m||r)$ where r is a random string of predetermined length and $m||r$ denotes the concatenation of m and r. Now if Eve tries to replay a previously seen message, Amazon will notice that the random string at the end of her message is the same as that at the end our ours, and so can reject her attempted forgery.

4.2 Application: Digital Signatures

Before discussing the second type of attack, we first introduce a situation where it is likely to come up. Up to this point, we've been using Bob's private key to ensure that he is the only person who can decrypt a message sent to him. But it turns out that we can also use it as a *proof of identity*; that is, Bob can use his private key to prove that he was actually the person who sent a message.

One naïve way to do this would be to have Bob include his private key d as part of his message. However, this has the problem of revealing Bob's private key, which is always a bad idea. After all, this method may work once, but then Bob's private key is compromised, so he would have to generate a new one — and any encrypted messages Bob previously received can now be decrypted by anyone.

Being a little bit more clever, we can instead sign a message m using $D_{N,d}(m)$; that is, we pretend that m is actually a ciphertext and sign with whatever it would end up decrypting to. For Bob, this is easy to do, as he knows d. Additionally, it is easy for Bob's recipient to verify that the signature is valid: a signature s for a message m is valid if and only if $s^e \equiv m \pmod{N}$ by a proof similar to that of Theorem 7.4. However, someone who is not Bob cannot forge a signature for arbitrary messages, as that would be equivalent to being able to decrypt arbitrary ciphertexts, which is impossible under the assumption that RSA is secure.

4.3 Digital Signature Attack

One thing you may have noticed about our scheme for digital signatures is that while it may be hard to forge arbitrary signatures, some are quite easy to form. In particular, even without knowing d, one can sign the message $m = 0$ or $m = 1$, as $m^d = m$ for these values of m regardless of what d is. Hence, we might figure that if we want to be sure Bob is really Bob, we should choose what message he has to sign.

Unfortunately, this opens up a vulnerability in our system. In particular, suppose Eve previously intercepted some ciphertext c intended for Bob. She chooses a random number r and asks Bob to sign the message $r^e c \pmod{N}$. Bob doesn't know how she came up with that message, and sees nothing suspicious about it, so he happily signs it. But now his signatures is $(r^e c)^d \pmod{N}$, which is just $r^{ed}c^d \pmod{N}$. By Theorem 7.4, $r^{ed} \equiv r \pmod{N}$ and $c^d \equiv m \pmod{N}$ (where m is the original message that lead to the ciphertext c), so Eve now has access to $rm \pmod{N}$.

But now Eve knows r and N, so if she was clever enough to choose r coprime to N (which can be checked using Euclid's GCD algorithm), she can use the extended version of Euclid's algorithm to find $r^{-1} \pmod{N}$. Multiplying this by Bob's signature gives her the plaintext message m! This tells us that we have to be careful when designing our digital signature system to ensure that we don't accidentally break our own encryption in the process.