6.1200J/18.062J *Mathematics for Computer Science* Thursday 14th March, 2024
Massachusetts Institute of Technology, Spring 2024
Z. Abel, B. Chapman, E. Demaine revised Thursday 14th March, 2024

Lecture 10: Cryptography

1 Cryptography

You've seen a couple of lectures on basic number theory now. Why was it in 6.1200? Today we will see how GCDs and modular arithmetic are extremely important for computer security!

Cryptography has become a subject of enormous significance to our daily life over the last 20-30 years. It is firmly a computer science topic, and it uses all sorts of extraordinary math, stuff you would have never thought would have any application at all. Like... modular arithmetic!

What is cryptography? In a sentence: Cryptography: Art/Science of Protecting Information.

Idea: Encrypt or "garble" a message that you want to send privately, in such a way that only certain parties can read it.

The intended recipient should be able to Decrypt or "ungarble" it to recover the original message.

Convention: Alicekazam is sending encrypted messages to Bobasaur, Bobasaur is decrypting them, and Eevee is an Eavesdropper who overhears everything being sent, but hopefully still can't understand it.

An Encryption method together with a Decryption method is known as a Cryptographic Scheme.

2 History (NOT EXAMINED)

2.1 Caesar Cipher

Early example: Caesar Cipher. Actually used by Julius Caesar for sensitive military messages. To encrypt a message m made of letters A to Z: Alicekazam moves each letter forward 3 letters in the alphabet, wrapping around. So CRYPTO becomes FUBSWR. Said differently: think of A–Z as numbers 1–26, and add 3 mod 26. To decrypt, Bobasaur just subtracts 3 mod 26.

"Security by Obscurity": this only works if Eevee doesn't know you're using it. Otherwise, it is very easy to break.

How to overcome security by obscurity? Answer: rely on some secret key, or parameter.

2.2 Caesar Shift

Caesar shift: Alicekazam and Bobasaur agree beforehand on a secret shift value k, that Eevee doesn't know. To encrypt, Alicekazam adds k mod 26 to each letter. To decrypt, Bobasaur subtracts k mod 26 from each letter. Above example used $k = 3$. Rot13 uses $k = 13$, to move each letter halfway around the alphabet.

Is this better? Eevee doesn't know k, so are we safe? Susceptible to a ***Brute Force*** attack: Eevee can just try all 26 options. One will probably look more like English than the others. E.g., try looking at all 26 shifts of "ORJMJVYNYDQZMBZYDIVTZGGJRRJJY". This site makes that easy: `https://www.dcode.fr/caesar-cipher`.

2.3 Substitution Cipher

Need larger set of secret keys, known as the *key space*. One idea: Alicekazam and Bobasaur agree on a table mapping each letter to a different letter. E.g., $A \to X$, $B \to T$, $C \to W$, etc. There are 26! different keys, so Eevee isn't going to try them all!

Susceptible to ***Frequency Analysis***. E.g., if W is the most common letter in your message, W probably decrypts to the most common English letter, E. (Except in special cases, like *A Void*, a novel that contains no Es, which is translated from the french *La Disparition*, which also has no Es.)

2.4 German Enigma

Imagine changing the shift each time? The WWII German Enigma Machine used a complicated mechanical device (about the size of a lunchbox) to change the shift values depending on *the letters in the message itself*, as well as the secret initial configuration of the machine. Complicated arrangement of rotors and wires and reflectors, oh my! For a total of around $3 \cdot 10^{114}$ different possible settings. Can't possibly check them all.

"Security by Hubris": this might not work if Eevee is smarter than you are. The engineer who developed Enigma did not know how to break it, and assumed that nobody else could either. But still, Allies broke the code. Developed a room-sized machine (called "bombe" because of the ticking sound). Used, among other things, knowledge about the kinds and formats of messages often sent, e.g., weather reports that all started with the same phrase. This is known as a ***Known Plaintext Attack***. Lots of other clever ideas went into it too.

2.5 One-Time Pad

Different idea: Caesar Shift, but with bigger numbers. Instead of encoding a single character at a time, group letters into much larger numbers. An example (bad, but that's ok): HELLOWORLD, using 1–26, becomes 08051212152315181204, just a 20-digit number. Let's work with numbers with hundreds of digits! No problem. Or do it in ASCII and read the binary bits. The encoding doesn't matter. What matters is that now, messages are *numbers*.

Pick a large n (imagine 1000 bits), and now a message m is just a number in the range

$0, 1, 2, \ldots, n-1$. If Alice and Bob agree on a number $0 \le k \le n-1$, then we can Caesar Shift: Alice sends $enc(m,k) = \text{rem}(m+k, n)$, and Bob computes $m = \text{rem}(enc(m,k) - k, n)$.

Advantages: k is arbitrary, so without knowing k, any message can have any encoding! No information is revealed. If k is chosen uniformly randomly, then $m+k$ is also uniformly random. Will learn more about randomness later.

Disadvantage 1: Message has to be smaller than n. What if Alicekazam needs to send a longer message? Break it into chunks of size $< n$ and send each with key k? No, bad things happen when reusing k.

First bad thing: ***Known Plaintext Attack***: If Eevee ever gets hold of the encrypted *and* unencrypted message, or has some information on it (like the German weather reports), then she can subtract to learn k! On future messages, she can subtract k and read Alicekazam's messages.

Second bad thing: Say $enc(m_1, k) = m_1'$ and same for $enc(m_2, k) = m_2'$. Then $m_1' - m_2' \equiv_n m_1 - m_2$. So info is leaking about how two messages relate to each other. It mushes the two messages together, but still a lot of info is revealed.

We can't re-use k. That's why this scheme (and ones like it) is called a ***One-Time Pad***.

So for longer messages, Alicekazam will just send the new k value before each one... Oh, huh. Well, maybe she'll *encrypt* the new k value first... But that needs us to already have an earlier agreed-upon key... Hm. If she had a secure way to send a new one-time pad, then she'd just use that method instead! Drat.

2.6 Diffie-Hellman

A clever way for Alicekazam and Bobasaur to agree on a secret value k, over public channels, without Eevee being able to discover it. Used every time you connect to an https website (the "s" is for "secure").

Informal idea: suppose we have a "nice" *One-Way Function* f (a function that is easy to compute but hard to invert). Alicekazam can choose some random a and send $f(a)$ to Bobasaur. Bobasaur can choose some random b and send $f(b)$ to Alicekazam. Eevee only knows $f(a)$ and $f(b)$. Alicekazam and Bobasuar also each know one of a, b. If we can come up with a function g that somehow combines a and b and applies f only once, we could hope that $g(a,b)$ could be computed from $(a, f(b))$ or from $(f(a), b)$, but not from $(f(a), f(b))$.

Now, formally: f and g will be modular exponentiation. Choose n to be a large prime number, 100s of digits. Alicekazam chooses a random number $1 < c < n-1$ and sends it to Bobasaur. Then, Alicekazam chooses a random number a, while Bobasaur chooses a random number b. Alicekazam computes $f(a) := \text{rem}(c^a, n)$, and Bobasaur computes $f(b) = \text{rem}(c^b, n)$, and they send these numbers to each other. Finally, Alicekazam computes $x := \text{rem}(f(b)^a, n)$, and Bobasaur computes $y := \text{rem}(f(a)^b, n)$. Claim: $x = y$, so this will be their shared secret key. They can use this number as a one-time pad.

Note that $x \equiv_n f(b)^a \equiv_n (c^b)^a \equiv_n c^{ab}$, and the same is true of y, so they're equal.

Another note: how do Alicekazam and Bobasaur compute things like $\text{rem}(c^a, n)$? Both a and c have hundreds of digits! c^a is huge! (That's one reason we reduce mod n, so everything stays reasonable.) And we don't have time for a different multiplies! (That's what repeated squaring is for: can compute $\text{rem}(c^a, n)$ with just $O(\log a)$ arithmetic operations mod n.)

Finally: Eevee knows c, c^a, and c^b. Why can't she recover a from this, and then compute $(c^b)^a = x$? If these were actual integers, she could just take a log! But we're working mod n, so recovering a from c and c^a mod n is known as the ***Discrete Log Problem***. Nobody (not even Eevee) knows a computationally tractable method to solve this.

3 RSA

Downside of Diffie-Hellman: need to actively agree on key before sending a message. If Alicekazam wants to send Bobasaur an encrypted email, she doesn't want that back-and-forth first!

One more protocol: RSA = Rivest, Shamir, Adleman, the three inventors of the cryptographic scheme. It's really a workhorse of modern internet security. And they did it here at MIT! And won a Turing award for it, in 2002!

Public-Key cryptosystem. You can tell everybody the encryption key, publicly! But it only lets you encrypt, not decrypt. You have the accompanying secret key that only you know, that you can use to decrypt. Boggling that this is possible!

Start with two large primes, p and q. Keep these primes secret, but publish $n := pq$. Then, choose a large number e that is coprime to $(p-1)(q-1)$. Your public key is $k_p := (n, e)$.

Next, compute d which is a multiplicative inverse of e modulo $(p-1)(q-1)$, which we can do with the Pulverizer, because e is coprime to $(p-1)(q-1)$. Keep p, q, d secret. Your secret key is $k_s := (n, d)$.

To encrypt message m, compute $E(m, k_p) := \text{rem}(m^e, n)$. To decrypt an encrypted message c, compute $D(c, k_s) := \text{rem}(c^d, n)$.

After encrypting and then decrypting, you're computing $(m^e)^d \equiv_n m^{ed}$. Claim: $m^{ed} \equiv_n m$.

Why is this true?! Fermat's Little Theorem!

We know $ed = 1 + t \cdot (p-1)(q-1)$ for some integer $t \geq 0$. Let's assume m is relatively prime to pq. (In practice, these primes are much too big to stumble on "by chance", so this assumption doesn't matter in practice. We'll see how to remove it in Recitation.)

Working mod p, we find

$$m^{ed} \equiv_p m^{1+t(p-1)(q-1)} \equiv_p m \cdot (m^{p-1})^{t(q-1)} \equiv_p m \cdot 1^{t(q-1)} \equiv_p m.$$

The same is true mod q. So p and q both divide $m^{ed} - m$, so pq divides $m^{ed} - m$. In other words, $m^{ed} \equiv_{pq} m$. † † †

3.1 Security

Public: $n = pq$ and e. Private: p, q, d. Relies on ***Factoring*** being hard! No efficient method known to factor n into its two primes. (Not true with Quantum Computers! Which we can't actually build yet. But soon? Maybe?)

Important point: we are *assuming* factoring is hard; we don't have a proof. No algorithm to efficiently factor large numbers is currently known publicly, but that doesn't mean it doesn't exist! It might! It might even be known right now, by someone or some intelligence agency that just hasn't shared it around. No way to know for sure. In fact, we are assuming even more. If we assume the Extended Riemann Hypothesis, then computing the secret key (n, d) from the public key (n, e) is (essentially) the same problem as factoring. However, without ERH, we don't actually know that breaking RSA will always factor the modulus n. Further, it is in principle possible that we could decrypt single messages without computing the secret key at all. This is why, in addition to assuming factoring is hard, we also assume ERH and/or the *RSA Assumption*, which says that computing e^{th} roots modulo pq is also hard.

Even though we don't have a full, unconditional proof of security for RSA, we are still *much* more confident in its security than in something like Enigma. The above assumptions have been well-studied for many years by many experts, plus there are large cash prizes for very incremental progress towards breaking RSA, so it is safer to assume that nobody yet knows how to break it.

3.2 Finding p and q

How do we find large primes p and q? Two key insights.

1. Numbers can be easily *tested* for primality. This is the Miller-Rabin algorithm from Pset. So, naive idea: pick random 300-digit numbers and test for primality, until one of them is prime! Are there enough primes for that?! Don't they get more and more sparse as numbers get big! Wouldn't this take forever?

2. This is actually fast, because there are *lots* of primes. Prime Number Theorem: $\pi(k) \sim k/\ln k$. Around $k = 10^{300}$, the density of primes is about $1/\ln(10^{300}) \approx 1/700$, so in expectation, 700 tries is enough! This is fast.

3.3 Warning

Don't implement RSA yourself now, expecting it to be secure! This is a simplified version, and lots of attacks are known without further precautions.

4 Chinese Remainder Theorem

In the RSA proof above († † †), we could have simplified our proof by invoking a powerful theorem called the Chinese Remainder Theorem (CRT). Roughly speaking, CRT says that if p and q are coprime, then computing modulo pq is the same as computing modulo p and modulo q separately.

As a simple example, consider the following: Suppose I have an integer $0 \le x < 55$ where $x \equiv_5 4$ and $x \equiv_{11} 7$. That's enough information to uniquely identify x. What is x in this case? We can proceed by trial and error. The numbers in this range that are congruent to 7 modulo 11 are $\{7, 18, 29, 40, 51\}$. Of these, only $x = 29$ is also congruent to 4 modulo 5.

Theorem 1 (CRT)**.** *Suppose p and q are coprime, and $a, b \in \mathbb{Z}$. Then, modulo pq, there exists a unique solution x to the following system of modular congruences:*

$$x \equiv_p a \tag{1}$$
$$x \equiv_q b \tag{2}$$

Proof of existence. Let's start by thinking about the special case where $a = 0$. Define p^{-1} to be a modular inverse of p modulo q, and define $e_q := p^{-1}p$. Notice that $e_q \equiv_p 0$ because it has a factor of p. By construction, $e_q \equiv_q 1$, so $be_q \equiv_q b$. This means that $x = be_q$ is our desired solution.

For the more general a and b, we can also define q^{-1} to be a modular inverse of q modulo p, $e_p := q^{-1}q$, and $x := ae_p + be_q$. Now, working modulo p:

$$\begin{aligned} x &\equiv_p ae_p + be_q \\ &\equiv_p aq^{-1}q + bp^{-1}p \\ &\equiv_p a1 + bp^{-1}0 \\ &\equiv_p a \end{aligned}$$

Also, working modulo q:

$$\begin{aligned} x &\equiv_q ae_p + be_q \\ &\equiv_q aq^{-1}q + bp^{-1}p \\ &\equiv_q aq^{-1}0 + b1 \\ &\equiv_q b \end{aligned}$$

So x satisfies Congruences 1 and 2. □

Proof of uniqueness. Suppose x and x' both satisfy Congruences 1 and 2. We want to prove that $x \equiv_{pq} x'$. To do this, we show that $y := x - x'$ is a multiple of pq.

We know that $x \equiv_p a$, $x' \equiv_p a$, $x \equiv_q b$, and $x' \equiv_q b$. Putting these together, we have $x \equiv_p x'$ and $x \equiv_q x'$. Unpacking definitions, this says that $p|y$ and $q|y$, so we may take integers k_p, k_q such that $pk_p = qk_q = y$. Now $\gcd(p, q) = 1$, so there exist s, t such that:

$$\begin{aligned} 1 &= ps + qt \\ k_p &= psk_p + qtk_p \\ &= ys + qtk_p \\ &= qk_qs + qtk_p \\ &= q(k_qs + tk_p) \end{aligned}$$

We have $q \mid k_p$, so $pq \mid pk_p = y$, as desired. $\square$

Note: the interactive demos from lecture (including the one that didn't project) are on Canvas under Files/Uploaded Media. They should run with a standard Python distribution that includes tkinter.