Course Curriculum: Cryptography
Sharing Secret Messages

Talley Amir

Yale University

Summer 2020

Citation and Disclaimer

I acknowledge here that the theorems and algorithms presented below are not my own work, and the compilation of this material is intended for educational purposes only.

Module 1: Introduction to Cryptography

July 15, 2019

Cryptography is defined in various ways. Some definitions will say that it is the study of writing and solving codes, but more recently the study has come to encompass user authentication, differential privacy, and secure multiparty computation. Today we are going to look at some of the earliest developments in cryptography, and see how these codes have been broken, redesigned, and broken time and time again.

Codes

General Codes

- Consist of a codebook which maps words to codewords.
- Examples:
  - Encode each letter of the alphabet as its corresponding letter: A=1, B=2, ..., Z=26.
  - If we want to encode the word "CRYPTO," we get "3-18-25-16-20-15."

Morse Code

- First used in 1844.
- Can communicate each letter in the alphabet as a series of dots and dashes.
- Example:

$$Y = -.-- \qquad A = .- \qquad L = .-.. \qquad E = .$$

- It's easy to use; you can send messages with any medium that has digital signaling (switching a light on and off or sounding a horn).

Exercise 1:

Pick a friend from across the room and take turns picking a word and blinking Morse code to them. See if you can get the word right!

- Problems with Morse code:
  - Can only transmit one message per wire at a time.
  - Not everyone knew Morse code at the time it was introduced - it was like having to learn a new language.
  - NOT prefix free! (AE = U = dot dot dash). This can lead to confusion and many miscoded messages.

Prefix-Free Codes

- A word w_1 is a prefix of another word w_2 if w_2 begins with w_1.
- Prefix-free codes: No codeword contains another codeword as its prefix.
- This is helpful because it means we can decode a message as each symbol is received. (Why?)

Ciphers

- Require a *key* and an efficient algorithm for encryption and decryption.

Caesar Cipher

- Enumerate the alphabet: A = 1, B = 2, ..., Z = 26.
- Pick a key K from 1 to 25 and shift each of the letters in the alphabet by K.
- To decrypt, subtract K.

Exercise 2:

Write a note to a friend, keep your key a secret! See if the other person can guess your message from the code without the key. Does anyone get the answer? Why is this not so difficult to do?
Exchange keys – did you guess the message correctly?

- Why this is insecure: There are only 25 possible keys (26 if you can send the message "in the clear") which is few enough to try them all. You can also learn something about the message based on which letters repeat themselves and where the spaces are, as well as how far apart we know certain letters to be. For instance, if you see a letter by itself, it is likely `'A'` or `'I'` – once this letter is uncovered, so are the rest.

Substitution (Monoalphabetic) Cipher

- Again, enumerate the alphabet. This time, our key is a *permutation* of the alphabet (take all the letters and switch them around). For example, the key may be:

  `D X S F Z E H C V I T P G A Q L K J R U O W M Y B N`,

  indicating that A maps to D, B maps to X, and so on...
- To encrypt, look up each letter in the permutation and switch it to that letter.

- To decrypt, reverse the process of encryption using the given key.
- This is better than Caesar's cipher because even if you break one letter, you cannot infer the rest of the letters as easily. In addition, there are *many* more possible keys (26*25*24*...*2*1).
- This is still weak because we can use a letter frequency attack to decrypt (either look for a specific word or phrase in the encryption, or look at the frequencies of each letter – we know that `'E'` and `'T'` occur the most frequently).
- It also requires a much larger key which may be harder to communicate to the recipient of the message.

One-Time Pad

- Enumerate the alphabet and select a message. Now, generate a key as long as the message where each component of the key is draw *uniformly at random* from 1 to 25. Shift each letter of the message by the corresponding shift in the key.
- Must securely communicate the key for each message.
- Must hide the key / destroy it properly after sending and decrypting.

> *Exercise 3:*
>
> Now write a note to a friend using the one-time pad scheme, again keeping your key a secret. See if the other person can guess your message from the code without the key.
> Does anyone get the answer?
> Exchange keys – did you guess the message correctly?
> Why is this scheme more secure?

Considerations

At this point, we have seen some very different strategies for enciphering, each with their own pros and cons. Some elements we should consider when choosing a code:

- Strength of cipher: If we want strong "security," we do not want to use Caesar's cipher as this is easy to attack.
- Key size: Caesar's cipher has the smallest key size, one-time pad has the longest (which means higher cost to securely transmit the key each time because the key must be as long as the message!).

Module 2: Introduction to Programming in Java

July 16, 2019

Today everyone will learn basic programming which they will use to break two of the ciphers we discussed yesterday. Everyone will register an account with HackerRank and sign up for the contest at

https://www.hackerrank.com/eduexplora-cryptography

00:00 - 00:15	Make HackerRank accounts and join contest
00:15 - 00:30	Introduce Python 3 and important syntax, see notes below
00:30 - 00:45	Explain how HackerRank works, code `compute-maximum` as a class
00:45 - 01:00	Everyone codes `compute-maximum` and submits through HackerRank
01:00 - 01:15	Break into pairs and code `compute-average`
01:15 - 01:30	Swap pairs and code `break-shift-cipher` to reveal hidden message!
01:30 - 01:45	Swap pairs and code `break-monoalphabetic-cipher`
01:45 - 02:00	

Module 3: Number Theory

July 17, 2019

Today everyone will learn the number theory background needed to understand and prove the security of RSA.

An integer a is *divisible* by a positive integer b if there exists an integer c such that $a = bc$. b and c are both *divisors* of a.

A *prime* is an integer that is only divisible by 1 and itself. Likewise, a *composite* is an integer that has more divisors than only 1 and itself (example: $6 = 3 \cdot 2$).

The *greatest common divisor* of two numbers a and b is the maximum integer that is a divisor of both a and b. This is written as $\gcd(a, b)$.

If $\gcd(a, b) = 1$, then a and b are said to be *coprime*, or *relatively prime*.

Exercise 1:

- What is the gcd of 9 and 17?
- What is the gcd of 30 and 75?

You may have figured out the answer by writing out all of the prime factors of each number and taking the maximum overlapping set of primes and multiplying them together. However, in practice this is hard to do because factoring is considered to be a "hard problem." Say I give you the number 1789 and ask you to find its prime factors? How would you do this? You might have to go through all the numbers between 1 and 1789 to find each of its divisors (though in fact this number is prime)! For very large numbers, this becomes unfeasible.

There is a fast algorithm that computes the gcd of two numbers. This is called the *Euclidean algorithm*, and it is defined as follows:

To compute $\gcd(a, b)$:

$$\begin{aligned}
\mathbf{a} &= q_0\mathbf{b} + r_0 \\
\mathbf{b} &= q_1\mathbf{r_0} + r_1 \\
\mathbf{r_0} &= q_2\mathbf{r_1} + r_2 \\
\mathbf{r_1} &= q_3\mathbf{r_2} + r_3 \\
&\ldots
\end{aligned}$$

If a is smaller than b, the first step of the algorithm swaps the numbers.

Walk through example: $\gcd(8, 30)$.

Proof of Correctness. Loop invariant: At each iteration, the gcd of the two bold values is the same as in the previous iteration. Thus at the end, we get the gcd of the first two bolded values (a and b).

Thus it suffices to show that $\gcd(r_i, r_{i+1}) = \gcd(r_{i+1}, r_{i+2})$. *Prove this on board.* □

Exercise 2:

- Compute $\gcd(9, 17)$ and $\gcd(30, 75)$ by hand and verify your answers with what you computed before.
- Log into HackerRank and code `compute-gcd`.

We can use the *extended Euclidean algorithm* to write $\gcd(a, b)$ as a *linear combination* of a and b. Specifically, we can find integers u, v such that:

$$au + bv = \gcd(a, b)$$

Define the relation R_m to be a relation over integers such that $(a, b) \in R_m$ if and only if m divides $a - b$. We write that a and b are *congruent* under the modulus m as follows:

$$a \equiv b \pmod{m}$$

Each number will be equivalent to its *remainder* when divided by m. Another way to think about this relation: If $(a, b) \in R_m$, then there exists an integer k such that $a = b + km$.

Let $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$. Given this definition, some properties of the "mod" relation:

$$\begin{aligned} a + c &\equiv b + d \pmod{m} \\ ac &\equiv bd \pmod{m} \\ a^n &\equiv b^n \pmod{m} \text{ for } n \in Z \\ a + k &\equiv b + k \pmod{m} \\ ak &\equiv bk \pmod{m} \end{aligned}$$

Notice that *division* does not always work here. (Example: $18 \equiv 6 \pmod{12}$ but $9 \not\equiv 3 \pmod{12}$).

The *multiplicative inverse* of an integer a modulo m is an integer $b = a^{-1}$ such that $ab \equiv 1 \pmod{m}$. For example, the inverse of 3 modulo 17 is 6 because $3 \cdot 6 \equiv 18 \equiv 1 \pmod{17}$. *Not all numbers have an inverse for a given modulus.* The rule is that an integer a has an inverse modulo m if and only if $\gcd(a, m) = 1$.

Now, we can use the extended Euclidean algorithm to find the inverse! Why? Because if we can find u, v such that $au + mv = 1$, then u is the multiplicative inverse of a modulo m.

Exercise 3:

- Use the extended Euclidean algorithm by hand to express 4 as a linear combination of 12 and 32 (Answer: $4 = 12 \cdot 3 + 32 \cdot (-1)$).
- Use the extended Euclidean algorithm by hand to find the inverse of 21 modulo 92 (Answer: 57).
- Log into HackerRank and code `compute-inverse`.

Euler's totient function computes the number of integers between 1 and N that are relatively prime to N. This is often denoted $\varphi(N)$. Notice that for a prime number p, $\varphi(p) = p - 1$.

We also have a general form for $\varphi(N)$ when N is the product of *exactly two primes* p and q:

$$\varphi(N) = (p-1)(q-1)$$

However, this does not extend to three or more primes.

Exercise 4:

Log into HackerRank and code `compute-phi-N`.
Hints: How will you test to see if a number is coprime with N? How many numbers must you check this coprimality?

Fermat's little theorem (FℓP): For prime p and any integer a not divisible by p,

$$a^{p-1} \equiv 1 \pmod{p}$$

Euler's theorem generalizes Fermat's little theorem for composite numbers: For any integer N and any integer a such that $\gcd(a, N) = 1$:

$$a^{\varphi(N)} \equiv 1 \pmod{N}$$

Module 4: RSA Encryption

July 18, 2019

Today we will learn how to encrypt and decrypt messages using RSA.

Last time we learned about the mod relation R_m. Essentially this relation groups together all integers with the same remainder when divided by m. Just like how we know that factoring is hard for integers, we know certain problems are hard within the mod relation.

The *RSA problem* is the problem of finding g such that $g^a \equiv h \pmod{m}$ for some $h \in \{0, 1, ..., m-1\}$. This problem is "hard," that is, we have no efficient algorithm for computing g. Finding the value of g is as hard as guessing the value at random. We can use this fact to build an encryption scheme!

RSA is a public key encryption scheme that was developed by Ron Rivest, Adi Shamir, and Leonard Adleman (where the name RSA comes from) in 1977. It is "secure" because we assume that factoring is hard and the *RSA problem* is hard.

Public key encryption schemes consist of three kinds of algorithms:

- KeyGen: This is a key generation algorithm. It produces pairs of keys (a public key and a private key) that are used to encrypt messages to one another. An individual runs the algorithm and then publishes their public key, but keeps their private key a secret.
- Enc: This is the encryption algorithm. If Alice wants to send a message to Bob (who has published his public key PK_{Bob}, Alice can encrypt a message to Bob using his public key.
- Dec: This is the decryption algorithm. If Alice sends a ciphertext C to Bob that was encrypted under Bob's public key PK_{Bob}, Bob can use his secret key SK_{Bob} to decrypt the message. Recall that the pair of keys $(PK_{\text{Bob}}, SK_{\text{Bob}})$ was obtained from the KeyGen algorithm.

The RSA encryption scheme is defined as follows:

- $\mathsf{KeyGen}(1^\lambda)$:
  1. Pick two large primes p and q and compute $N = pq$.
  2. Pick e such that $\gcd(e, \varphi(N)) = 1$, and compute $d = e^{-1} \pmod{\varphi(N)}$.
  3. Publish public key $PK = (N, e)$. Keep private secret key $SK = (p, q, d)$.
- $\mathsf{Enc}(m, PK)$:
  1. m must be an integer in $\{1, ..., N-1\}$. If not, define a reversible mapping from the message space to this set of integers.

2. Compute and output $c = m^e \pmod{N}$.

- $\mathsf{Dec}(c, SK)$:

1. Compute and output $m' = c^d \pmod{N}$.

Claim: The decryption m' is the originally encrypted message.

Proof.

$$\begin{aligned} m' &\equiv c^d \pmod{N} \\ &\equiv (m^e)^d \pmod{N} \\ &\equiv m^{ed} \pmod{N} \\ &\equiv m^{1+k\cdot\varphi(N)} \pmod{N} \\ &\equiv m \cdot (m^k)^{\varphi(N)} \pmod{N} \\ &\equiv m \cdot 1 \pmod{N} \\ &\equiv m \pmod{N} \end{aligned}$$

□

Let's try it! Split up into pairs and do the following activity:

Exercise 1:

- Let $A = 1$, $B = 2$, ..., $Z = 26$.
- Pick your own two primes (they should be *large*) and compute your public and private keys. Write your name and your public key on the board.
- Write a message to a friend! Pick someone on the board and encrypt a message to them using their public key. You should write the person's name on the message (unencrypted) for the next step.
- Give the message to your friend. Now they should decrypt their message, but still keep it private.

Why is RSA *secure*? Given a ciphertext c, it is hard to find the decryption m as long as p and q are unknown. What happens if p and q are known?

If p and q are known, then we can easily compute $\varphi(N)$ and use the extended Euclidean algorithm to find the inverse of e (a public parameter) modulo $\varphi(N)$. This inverse is exactly equal to the decryption exponent, which we can then use to decrypt any message we see!

Exercise 2:

- Steal someone else's message (a message that you did not write and that was not written to you).
- Everyone writes their prime numbers on the board.
- Use the now public private values to compute the private key and decrypt the message you stole!

Module 5: Symmetric Key Cryptography

July 19, 2019

Today we will learn about symmetric key cryptography and how to securely exchange keys in order to perform this kind of encryption.

Symmetric Key Cryptography

Symmetric key cryptography is an encryption scheme (consisting of algorithms KeyGen, Enc, and Dec) such that encryption and decryption are performed under the same key.

- $K \leftarrow \mathsf{KeyGen}(1^\lambda)$
- $c \leftarrow \mathsf{Enc}(m, K)$
- $m \leftarrow \mathsf{Dec}(c, K)$

This means that if two people want to communicate secretly using symmetric key cryptography, they need to have the same key. How do two people obtain the same key?

If I send the key to my friend, everyone will see it “in the clear,” so we need to do something more clever.

Diffie-Hellman Key Exchange

Diffie-Hellman key exchange is an algorithm for obtaining the same key by two participants over a public channel. It leverages the hardness of the *discrete logarithm problem.*

The *discrete logarithm problem* is the problem of finding a such that $g^a \equiv h \pmod{m}$ for some $h \in \{0, 1, ..., m-1\}$. This problem is “hard,” that is, we have no efficient algorithm for computing a. Finding the value of a is as hard as guessing the value at random.

The Diffie-Hellman key exchange algorithm works as follows:

- Pick a large prime modulus p.
- Pick a *primitve root* (generator) g of the modulus. This is an integer between 1 and $p-1$ such that g^a spans all of the integers for a from 0 to $p-1$. The number of primitive roots modulo p is $\varphi(p-1)$.
- Publish public parameters p and g.
- Alice and Bob want to share a common secret symmetric key S. Each one selects an integer a and b uniformly at random.

- Alice and Bob each compute:

  $$A = g^a \pmod{p}$$

  $$B = g^b \pmod{p}$$

  Alice sends A to Bob and Bob sends B to Alice. Notice that the two messages sent publicly reveal nothing about the secret values a and b because we assume that solving for the discrete logarithm is hard.

- Alice receives B and computes:

  $$S_a = B^a \pmod{p}$$

  Similarly, Bob receives A and computes:

  $$S_b = A^b \pmod{p}$$

- Now, Alice has a secret key S_a and Bob has a secret key S_b, and low and behold, they're equal!

Proof of Correctness.

$$\begin{aligned} S_a &\equiv S_b \pmod{p} \\ (g^a)^b &\equiv (g^b)^a \pmod{p} \\ g^{ab} &\equiv g^{ab} \pmod{p} \end{aligned}$$

□

Uses

Why use symmetric key cryptography if we have *public key* cryptography (like RSA encryption)? It's faster! However, we can combine the two to get secure key exchange and fast encryption.

How? Use public key cryptography to encrypt the symmetric key. After exchanging the key, use it to encrypt the rest of the conversation using symmetric key cryptography.

Module 6: Hash Functions

July 22, 2019

Today we learn about hash functions, their properties, various security notions regarding hash functions, and some applications.

A *hash function* is a function that maps from a set of arbitrary size to a set of fixed size. Some examples:

- The parity function (maps from the set of all integers to the set $\{0, 1\}$):
$$f(x) = \begin{cases} 0 & \text{if } x \text{ is even} \\ 1 & \text{if } x \text{ is odd} \end{cases}$$
- Modular equivalence (maps from the set of all integers to the set of integers from 0 to $p - 1$):
$$f(x) = x \pmod{p}$$
- Direct mappings (e.g. maps from the set of all integers to the set $\{1, 2, 3, 4\}$):
$$f(x) = \begin{cases} 1 & \text{if } x = 1 \\ 2 & \text{if } x = 2 \\ 3 & \text{if } x = 3 \\ 4 & \text{otherwise} \end{cases}$$

Hash functions must be deterministic (i.e. the same input must give the same output every single time, it cannot use randomness to compute the output), and must be fast to compute.

Hash functions have important applications in cryptography. They can be used to detect small changes in data, and to identify unique objects.

Hash functions are characterized by three notions of security:

- *Pre-image resistance:* A hash function H is pre-image resistant if, given a hash h, it is "hard" to find input x such that $H(x) = h$.
- *Second pre-image resistance:* A hash function H is second pre-image resistant if, given an input x_1, it is "hard" to find another input x_2 such that $H(x_1) = H(x_2)$.
- *Collision resistance:* A hash function H is collision resistant if it is "hard" to find two inputs x_1 and x_2 such that $H(x_1) = H(x_2)$.

  A note about collision resistance: Since we are mapping from an *infinitely large set* to a *fixed size set*, we necessarily must have collisions. Collision resistance is not saying that there will not be collisions, but rather that they will be hard to find.

We typically find these things to be "hard" if there is a computational hardness assumption we use to implement the hash function. To demonstrate this, lets look at some examples:

The Discrete Logarithm Problem

Recall the discrete logarithm problem that we learned about for Diffie-Hellman key exchange. The assumption is that given g, h, and p it is "extremely difficult" to find a such that $g^a \equiv h \pmod{p}$. Here, "extremely difficult" just means "as difficult as if we were guessing values for a completely at random."

Let's define the following hash function:

1. Pick *large* prime modulus p and generator g. Select integer h uniformly at random from $\{0, ..., p-1\}$.
2. Define $H(x_1, x_2) = g^{x_1} h^{x_2}$.

Then this hash function is *collision resistant.* Why? Because if we can find a collision, then we can break the discrete logarithm problem. This means that the discrete logarithm problem *is not hard*; however, we assume that it *is* hard. Therefore, it is *as hard to find a collision for H as it is to solve the discrete logarithm problem.* As a consequence, we can say that this hash function is collision resistant.

Exercise 1:
Prove that H is collision resistant. Namely, if we can find a collision for H, show how we can use this to solve an instance of the discrete logarithm problem.

Bad Examples

There are many failed attempts at making secure hash functions. In practice, people try to break the security of these functions all the time! Let's try to break a few ourselves:

Exercise 2:

a. Let $H(x_1, x_2) = x_1 + x_2$. Is H pre-image resistant? Is it second pre-image resistant? Is it collision resistant?

b. Let $H(x) = g^x \pmod{p}$, where g and p are known. Is H pre-image resistant? Is it second pre-image resistant? Is it collision resistant?

c. Let's assume that H is a collision resistant hash function. Is $H(x + 10)$ collision resistant?

Module 7: ElGamal Encryption Scheme

July 23, 2019

Today we will cover the ElGamal encryption scheme which also uses the hardness of the discrete logarithm problem to perform public key encryption.

The ElGamal encryption scheme is a public key cryptosystem, just like RSA. It consists of three algorithms (KeyGen, Enc, and Dec), defined as follows:

- $\mathsf{KeyGen}(1^\lambda)$:
  1. Pick a prime modulus p and generator of the modulus g.
  2. Pick a random integer x from $\{1, ..., p-1\}$. Compute $h = g^x \pmod{p}$.
  3. Publish public key $PK = (p, g, h)$. Key secret private key $SK = x$.
- $\mathsf{Enc}(m, PK)$:
  1. Pick a random integer y from $\{1, ..., p-1\}$, we call this an *ephemeral key* because it is random and is not reused in the algorithm. Compute $s = h^y$, we call this the *shared secret*.
  2. Compute $c_1 = g^y$ and $c_2 = m \cdot s$.
  3. Output $c = (c_1, c_2)$.
- $\mathsf{Dec}(c, SK)$:
  1. Interpret c as (c_1, c_2) and compute $c_1^x = (g^y)^x = (g^x)^y = h^y = s$ (the shared secret).
  2. Compute $s^{-1} \pmod{p}$ (using the Euclidean algorithm).
  3. Compute $m' = c_2 s^{-1} \equiv m \cdot s \cdot s^{-1} \equiv m \pmod{p}$.
  4. Output m'.

Exercise 1:

- Split into pairs and try an example. Compute public key (p, g, h) and private key x. Publish your public key, but not your private key!
- Encrypt a message to the other person using their public key. Swap ciphertexts.
- Decrypt the message you received using your private key. Confirm that the answer is correct.
- Mathematically show why decryption works.

One notion of security considers how hard it is to decrypt a ciphertext. Another notion of security that we need to be concerned with is whether or not we can alter a ciphertext in transit. This is called *malleability.*

For example, let's say I am participating in an auction. I decide to bid d dollars. An adversary intercepts my message $c =\mathsf{Enc}(d, PK)$ and modifies it to $c' = f(c)$ in such a way that now the adversary can submit c' as their bid and win the auction. Even though the adversary cannot decrypt my message, she can still cause damage by knowing how to modify the underlying message to achieve her goal.

Exercise 2:

- Split into pairs and try an example. Compute public key (p, g, h) and private key x. Publish your public key, but not your private key.
- Encrypt a message to someone else using their public key.
- Steal someone else's ciphertext and see if you can create a *new* ciphertext for a different message (any one of your choosing). How did you do it?
- When does this attack work? When doesn't it work?
- Why use a new y for every message? What happens if they're the same?

Reconvene as a class and discuss how you can modify messages under the ElGamal encryption scheme.

Also discuss how we can prevent such an attack: Use padding!

We can prevent malleability attacks by enforcing a padding scheme that allows a receiver of a message to detect whether or not their message was tampered with in transit. A *padding scheme* dictates that a message has to have a certain form when decrypted. For example, we might say that a message must look like $m(\cdot)H(m)$, where $(\cdot)$ represents concatenation and H is a secure hash function.

Exercise 3:
Log into HackerRank and code `decrypt-padding-elgamal`, assuming that the padding scheme we use is $m(\cdot)H(m)$, where H compressed m into an integer between 0 and 99. In python, we can implement this by hashing m using `hash`(`str`(m)) and then reducing modulo 100. Perform concatenation by multiplying m by 100 and adding $H(m)$. (Provide examples in class).

Module 8: Zero-Knowledge Proofs

July 24, 2019

Today we will learn about zero-knowledge proofs and explore some examples.

Up until this point, we have been looking at a very small subset of cryptography that deals with secure communication. In the presence of an adversary, we want to be able to communicate with a recipient and guarantee not only that the adversary cannot eavesdrop on the communication, but also cannot tamper with it.

From this point on, we will talk about another aspect of cryptography involving *zero-knowledge proofs*. This is a type of proof that allows a Prover to prove a statement to a Verifier without revealing *how* the Prover knows that the statement is true (only that it is indeed true). This allows the Verifier to trust that the statement is true while maintaining the Prover's privacy.

For example, let's say that Alice wants to prove to Bob that she has found Waldo in the picture of the game "Where's Waldo?" but she does not want to ruin the game for Bob. Bob can see the "Where's Waldo?" picture, but does not know where Waldo is yet. Alice can prove to Bob that she knows where Waldo is without revealing where he is in the picture. How?

Alice can take a piece of paper (much larger than the picture) and cut out a small hole exactly the size and shape of Waldo in the picture. She then covers the picture, placing the hole exactly on top of Waldo, and shows Bob Waldo through the hole. Bob trusts that Alice has found Waldo on her own, but does not himself know where he is in the picture.

Properties of zero-knowledge proofs:

- Completeness: If the statement is correct, then the Prover will successfully convince the Verifier of this every time.
- Soundness: If the statement is false, then the Prover will successfully convince the Verifier that the statement is actually true almost never.
- Zero-Knowledge: The Verifier cannot infer any more information from interacting with the Prover than the truth (or falsehood) of the claim that the Prover aims to prove.

Exercise 1:

- This exercise uses colored candy that I will bring into class!
- Split into pairs and take turns being the Prover and the Verifier.
- Every pair of students gets two pieces of candy, either the same color or two different colors.
- The Prover gets to look at the candy and then the Verifier needs to hold one in each hand (without ever having looked at them) and hold them behind her back.
- The Verifier will show the candy to the Prover (without looking at the candy) and ask the Prover to identify which candy is a certain color.
- The Verifier will them shuffle the candies behind her back and ask again. The Verifier should do this as many times as she needs to before she is certain of her answer to the following question: Does the Prover actually know the difference between the two candies? Specifically, did the Prover choose two candies of the same color, or of different colors?
- How many times did the Verifier shuffle the candies before she made her guess?

Proof of Knowledge: RSA Encryption

Zero-knowledge proofs are often used to prove one's identity.

For example, say Alice posts her RSA public key $Alice : (N, e)$ to a bulletin. This way, if someone wants to communicate securely with Alice, they can encrypt a message using her public key and send the ciphertext to her email `alice@crypto.net`. Eve wants to prevent Alice from receiving her messages. She can post to the bulletin $Eve : (N, e)$ in an attempt to steal Alice's messages, even though Eve cannot decrypt them (because she does not have the private decryption exponent d).

Alice wants to prove that she is the sole owner of the public key (N, e). A Verifier can verify this by sending an encrypted *nonce*, which is simply a message that is generated uniformly at random over all possible messages, to Alice.

The Verifier encrypts the nonce n with Alice's public key and sends Alice $c = \mathsf{Enc}(n, (N, e))$. Alice can then decrypt the message and send it back to the Verifier. Since only the Verifier knows the nonce it sent, it can check to see whether or not Alice decrypted correctly. If she did, then she must be the owner of the public key.

Module 9: More Zero-Knowledge Proofs

July 25, 2019

Today is a continuation of the topic of zero-knowledge proofs.

Today we will learn about a specific application of zero-knowledge proofs: digital signatures.

Digital signatures are used to sign documents, just as you would in real life. A person's signature on a document means that the person read and understood the document and is agreeing to what the document says. Digital signatures aim to simulate this functionality cryptographically.

Digital signatures must have three properties:

- *Authentication*: The signature should be able to prove that the person who signed the document did in fact sign the document.
- *Integrity*: The signature should ensure that the document has not been modified since it was signed.
- *Non-repudiation*: If a person signed a document, they cannot later try to say that they did not sign the document, i.e. they cannot revoke their signature.

Exercise 1:
Split into pairs and answer the following questions:

- Alice signs a legal document at home with no witnesses with a red pen. She gives the document to Bob to give to Charley. How does Charley know that the signature on the document is Alice's?
- The legal document consists of 3 pages, with Alice's signature only on the last page. How does Charley know that the document was not modified since Alice signed it?
- Alice later decides that she does not want to have her signature on the document: The document says she will sell Charley her house, but she no longer wants to move. Can Charley prove that Alice really signed the document?

Clearly there are many issues with handwritten signatures. They do not actually do what we want them to do. Digital signatures try to achieve the functionality described above with a certain level of "security" by leveraging the hardness of problems, as we have done before to encrypt messages.

Digital signature schemes consist of three algorithms: KeyGen, Sign, and Verify:

- KeyGen: This is an algorithm that takes a security parameter and randomly generates a pair of keys: A signing key SK and a verification key VK. The signing key is kept secret and is known only to the signer and the verification key is made public so that anyone can verify the signature.
- Sign: This is an algorithm that is performed by a signer on a message, or document. The signer uses their private signing key to sign the message.
- Verify: This is an algorithm that takes as input a document, a signature, and a verification key. It outputs `Valid` if the signature is a valid signature on the given document for this verification key. Otherwise, it outputs `Invalid`.

A digital signature scheme is said to be secure if it is difficult to *forge* a signature. That is, it is difficult to create a signature on a document that is valid according to the Verify algorithm with a given verification key without knowing the associated signing key.

Miller-Rabin Primality Testing

Like RSA, large primes are often used as keys because there are certain properties in modular arithmetic that only hold for primes (sometimes primes that have specific characteristics, as we will learn tomorrow!). How do we find large primes?

Miller-Rabin primality test: Tests whether an input integer $n > 3$ is prime or composite (with high probability):

1. Write n as $2^r \cdot d + 1$ with r odd. If cannot do this, return `composite`.
2. Repeat the following loop k times, where k is tuned according to the desired level of accuracy for the algorithm:
    a. Pick $a \in \{2, ..., n-2\}$ at random and compute $x = a^d \pmod{n}$. If $x = \pm 1$, continue (n *may* be prime).
    b. Repeat the following steps up to $r - 1$ times:
        i. $x = x^2 \pmod{n}$
        ii. If $x = 1$, return `composite`.
        iii. If $x = n - 1$, go to line 2.
    c. Return `composite`.
3. Return `prime`.

Exercise 2:
Log into HackerRank and code `is-prime`.

Module 10: Digital Signature Algorithm

July 26, 2019

Today we will look at some of the practical applications of digital signatures.

The *Digital Signature Algorithm* (DSA) is a federal standard for cryptographically signing documents. It was developed and accepted by the National Institute of Standards and Technology (NIST) in 1991.

- KeyGen:

  Key generation consists of two steps. First, the algorithm generates public parameters that are known to all users in the system. Then, a separate algorithm is used to generate keys for each individual user.

  PublicParamsGen(1^λ):

  1. Pick a cryptographically secure hash function H.
  2. Pick a large prime q and another large prime p such that $p \equiv 1 \pmod{q}$.
  3. Choose an integer h randomly from $\{2, ..., p-2\}$ and compute $g = h^{(p-1)/q} \pmod{p}$. If $g = 1$, pick a new h and try again.
  4. Output $PP = (p, q, g)$.

  UserKeyGen(PP):

  1. Choose an integer SK from $\{1, ..., q-1\}$ and compute $VK \equiv g^x \pmod{p}$.
  2. Output (SK, VK).

- Sign(D, SK):

  1. Pick a random $k \in \{1, ..., q-1\}$ and compute $r = (g^k \pmod{p}) \pmod{q}$.
  2. Compute $s = k^{-1}(H(m) + xr) \pmod{q}$.
  3. Output $\sigma = (r, s)$

- Verify(D, σ, VK):

  1. Compute $w = s^{-1} \pmod{q}$.
  2. Compute $u_1 = H(m) \cdot w \pmod{q}$.
  3. Compute $u_2 = r \cdot w \pmod{q}$.
  4. Compute $v = (g^{u_1} y^{u_2} \pmod{p}) \pmod{q}$.
  5. If $v = r$, output `Valid`. Otherwise, output `Invalid`.

Exercise 1:
Split into pairs and prove with your partner that the scheme is correct (i.e. the output of Verify is `Valid` *if and only if* the signature is actually valid. This requires you to prove two things:

- If the signature is valid, then the output of Verify is `Valid`.
- If the signature is invalid, then the output of Verify is `Valid` only with negligible probability.

Exercise 2:
Swap pairs and do the following activities on HackerRank:

- Code `DSA-Sign` to sign important documents.
- Code `DSA-Verify` to determine whether or not signed documents are valid under a person's public verification key.

Module 11: Secret Sharing

July 25, 2019

Today we will learn about secret sharing.

Secret sharing was invented independently by Adi Shamir and George Blakley in 1979.

Uses:

- Access control (make sure information can only be recovered when a certain number of collaborators contribute information).
- Key storage.
- Missile launch codes.
- Storage of any kind of information that is highly sensitive and needs to remain perfectly hidden, but is not completely lost (can still be recovered later on).

Secret sharing gives us *confidentiality* and *reliability* (i.e. instead of keeping multiple copies of the same information, we keep redundant information that allows us to reconstruct the secret without storing the secret at all).

A t-out-of-n *secret sharing scheme* is a tuple of algorithms (Share, $\mathsf{Reconstruct}$). The Share algorithm takes a secret and creates shares, which are then securely distributed to multiple parties. The $\mathsf{Reconstruct}$ algorithm takes a number of shares, and if it gets at least t shares, it outputs the secret.

A t-out-of-n secret sharing scheme is *secure* if t shares can be used to reconstruct he secret, but $t-1$ or fewer shares reveal as much about the secret as zero shares. (Give bad example and good example).

Binary Strings

- The secret is a binary string s (which represents some message that we have encoded).
- $\mathsf{Share}(s)$ randomly samples $n-1$ binary strings $u_1, u_2, ..., u_{n-1}$ of the same length as s and computes

  $$u_n = s \oplus u_1 \oplus u_2 \oplus ... \oplus u_{n-1}$$

  The n values are then distributed to n parties.
- In order to $\mathsf{Reconstruct}$, the n parties must come together and XOR all of the shares to get s.

Shamir Secret Sharing

- A polynomial of degree $t-1$ is uniquely defined by any t points on that polynomial.
- Let the secret s be an integer. Then define the point $(0, s)$ in the xy-plane.
- $\mathsf{Share}(s)$ randomly samples $t-1$ points (each with unique x values) and then interpolates the points to find a polynomial of degree n.
- Polynomial interpolation:

$$p(x) = \sum_{i=0}^{n} \left(\prod_{i \neq j} \frac{x - x_j}{x_i - x_j} \right) y_i$$

- The remaining shares are randomly samples points that are on $p(x)$, each with unique x values.