CS 798
Privacy in Computation and Communication

Module 2
Background

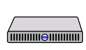
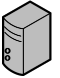
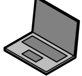
Spring 2024

Module outline

1. How the Internet works (briefly)
2. Cryptography
   - Symmetric-key encryption, hash functions, and MACs
   - Public-key encryption and signatures
3. Secret sharing
   - Additive/XOR secret sharing
   - Shamir secret sharing
   - Replicated secret sharing
4. Threat models
5. Zero-knowledge proofs

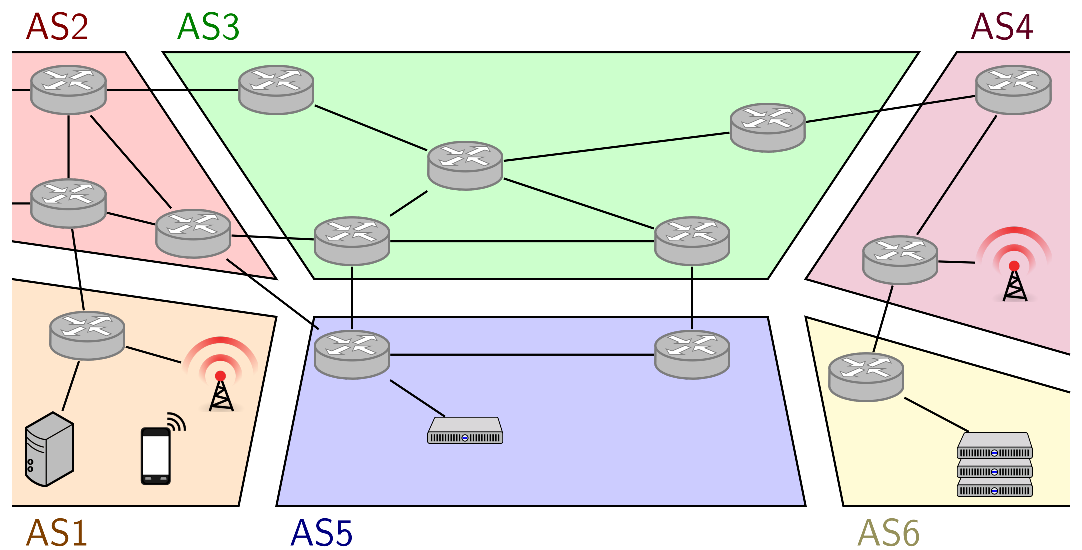
Components of the Internet

- Routers
- Links
  - Electrical, optical, wireless
- Endpoints
  - Servers, desktops, laptops, mobile, IoT, etc.

images from openclipart.org

The Internet



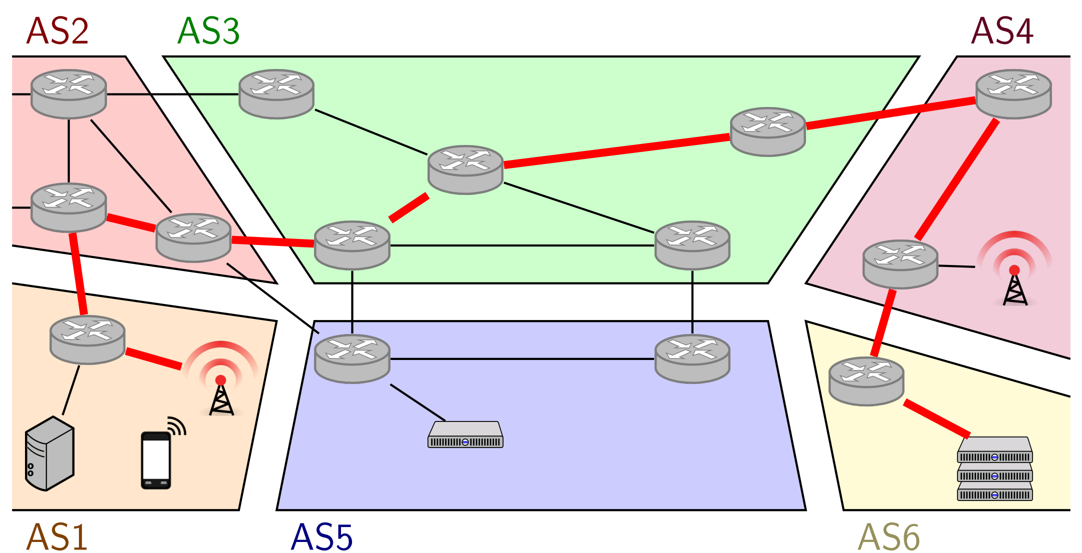
Naming

- Endpoints (particularly servers) have *names*
  - e.g., crysp.uwaterloo.ca, www.cs.ubc.ca
- But all Internet traffic works with *Internet Protocol (IP) addresses*
  - e.g., 129.97.167.96, 142.103.6.5
- Clients convert names to IP addresses using *DNS*

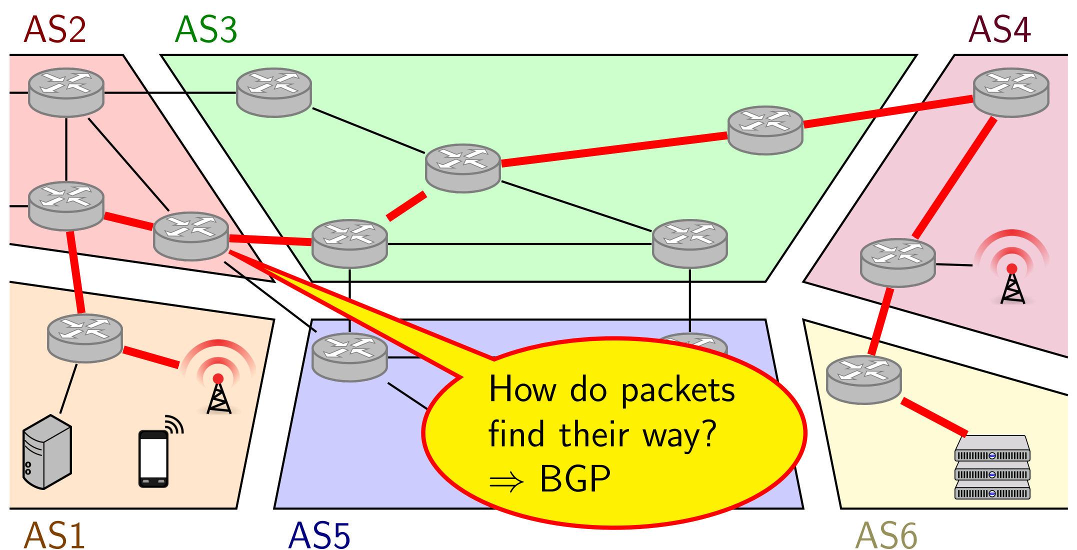
Packet routes

```
Host
 1. dc-csfw1-1-csresearch1.uwaterloo.ca
 2. po356-1500-10-dist-rt.ns.uwaterloo.ca
 3. po130-ftd-dist-sa-mc-campus-trust.ns.uwaterloo.ca
 4. v500-dist-rt-untrust.nsx.uwaterloo.ca
 5. po40-cn-rt-rac.ns.uwaterloo.ca
 6. gi0-0-0-ext-rt-rac.ns.uwaterloo.ca
 7. 72.15.57.69
 8. (waiting for reply)
 9. be320.dr01.151FrontStW01.YYZ.beanfield.com
10. (waiting for reply)
11. beanfield.ip4.torontointernetxchange.net
12. shaw.ip4.torontointernetxchange.net
13. rc1fs-ge11-0-0.mt.shawcable.net
14. rc2nr-be25.wp.shawcable.net
15. rc3no-be110-1.cg.shawcable.net
16. rc1st-be11-1.vc.shawcable.net
17. rd3bb-tge0-11-0-0.vc.shawcable.net
18. 208.98.209.22
19. cr2-100g-bb3928ae1.vncv1.bc.net.bc.net
20. 134.87.30.149
21. 137.82.88.122
22. a1-a0.net.ubc.ca
23. 137.82.73.13
24. www.cs.ubc.ca
```

The Internet



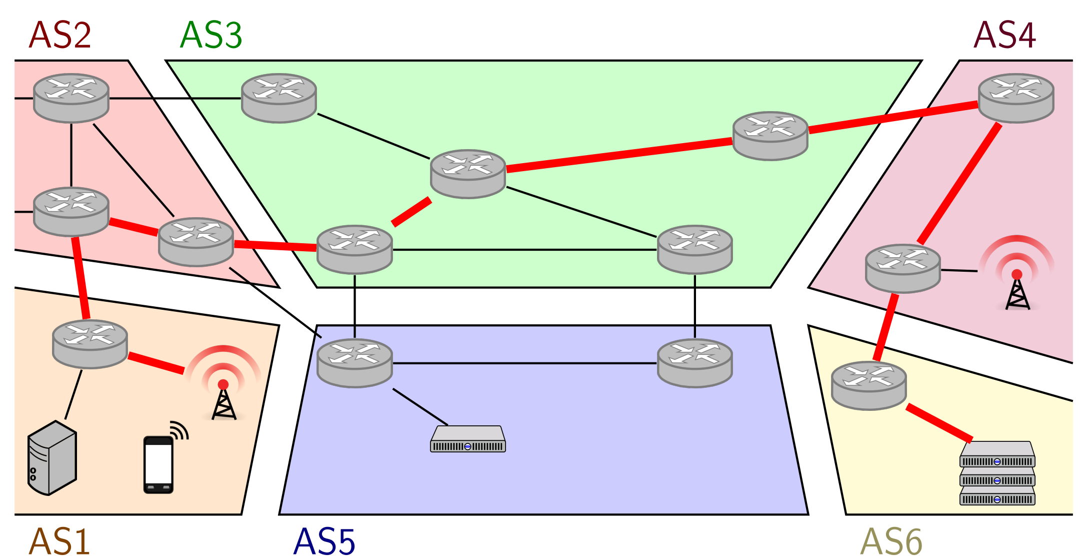
The Internet



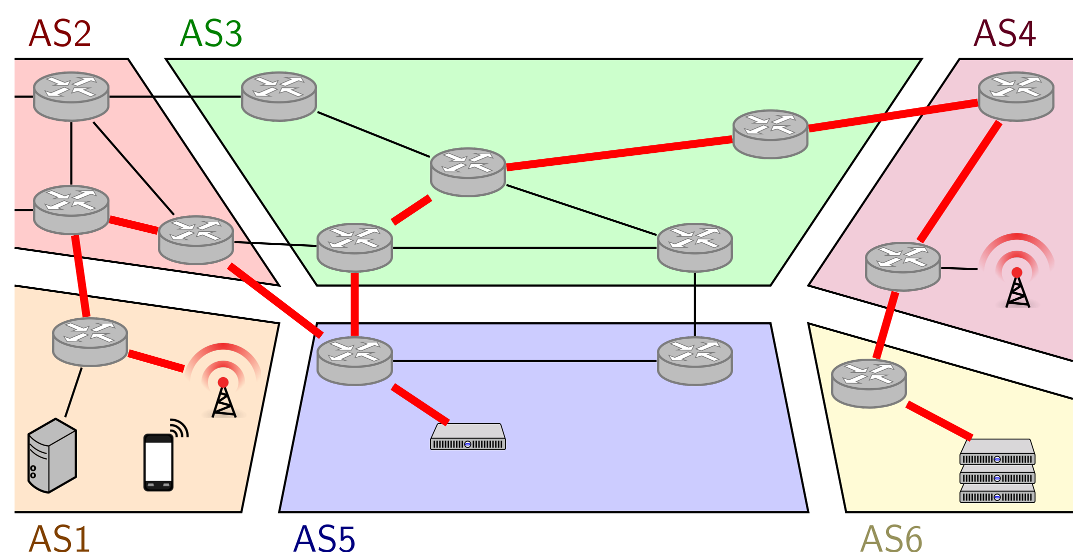
Packet contents

- What is visible to the routers along the way?
  - Packet headers: source and destination IP addresses, ports
  - Packet data: may or may not be encrypted
- Not only visible, but packets can be *modified*, *dropped*, or *forged*
  - National firewalls often do this, for example

BGP hijacking



BGP hijacking



KlaySwap crypto users lose funds after BGP hijack

Hackers have stolen roughly $1.9 million from South Korean cryptocurrency platform KLAYswap after they pulled off a rare and clever BGP hijack against the server infrastructure of one of the platform's providers.

The BGP hijack—which is the equivalent of hackers hijacking internet routes to bring users on malicious sites instead of legitimate ones—hit KakaoTalk, an instant messaging platform popular in South Korea.

https://therecord.media/klayswap-crypto-users-lose-funds-after-bgp-hijack

Packet contents

- What is visible to the routers along the way?
  - Packet headers: source and destination IP addresses, ports
  - Packet data: may or may not be encrypted
- Not only visible, but packets can be *modified*, *dropped*, or *forged*
  - National firewalls often do this, for example

Packet contents

- What is visible to the routers along the way?
  - Packet headers: source and destination IP addresses, ports
  - Packet data: may or may not be encrypted
- Not only visible, but packets can be *modified*, *dropped*, or *forged*
  - National firewalls often do this, for example

$\Rightarrow$ So not even just routers along the way, but potentially adversaries elsewhere!

When is packet data protected?

- No encryption
  - e.g., DNS, a small amount of web, some messaging
- Client-to-server encryption
  - e.g., most web (TLS), DNS-over-HTTPS (DoH), email, many messengers (e.g., WeChat)
- End-to-end encryption
  - e.g., iMessage, Signal, WhatsApp, sometimes Zoom and Teams

When are packet headers protected?

- Almost never
- Virtual Private Networks (VPNs) protect packet headers to some degree
  - But not from the VPN service itself
- Tor protects packet headers better
  - Even from Tor itself

$\Rightarrow$ Both still vulnerable to some extent to adversaries that have many vantage points on the Internet

Cryptography

- The main tool we use to protect the contents of communications online
  - Data, not metadata
- Can protect data in transit
  - TLS, Signal, and other Internet encryption
- Can protect data at rest
  - Encrypted storage on your laptop or phone
- Can protect data in use
  - Computing directly on encrypted data (more later)

Cryptography

- Two main kinds of cryptography:

- Symmetric-key cryptography
  - The two communicating parties (often called Alice and Bob) share a secret *key* that no one else knows
  - Anyone with the key can both encrypt and decrypt (and so read) the message, but no one else can (hopefully)

- Public-key cryptography
  - The key used to encrypt is different from the one to decrypt
  - So Bob can publish his *public key*, which can be used to encrypt messages to him
  - But he keeps his *private key* a secret, so that only he can decrypt those messages

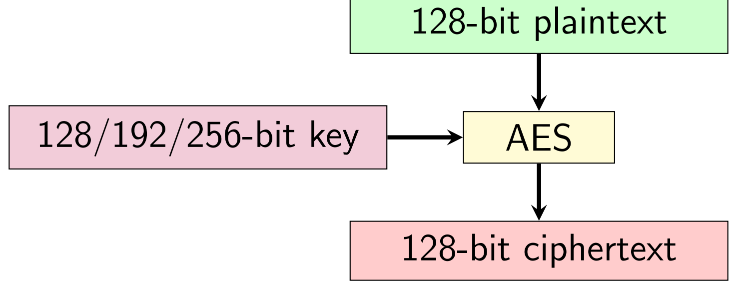
Confidentiality, Integrity, Authenticity

Cryptography can be used for three primary purposes:

- Confidentiality
  - If Alice sends a message to Bob, no third party can read it
- Integrity
  - If Bob receives a message from Alice, then that was the message Alice sent
- Authenticity
  - If Bob receives a message from Alice, it was really Alice who sent that message

Symmetric-key cryptography

- A common building block for symmetric-key cryptography is AES
- We won't talk about how AES works "on the inside"
- From the outside, it looks like this:



AES

- Primarily used for *confidentiality*, but other uses as well
- AES is very fast, because modern CPUs implement it *in hardware*
- It can encrypt at 1–2 GB/s (per core on a multi-core CPU)
- Try it yourself: `openssl speed aes`
- Look for "`aes`" in the "`flags`" line of `/proc/cpuinfo` to see if your CPU does AES in hardware

Hash functions

- An important building block to achieve *integrity*
- A hash function takes an input of arbitrary length, and outputs a fixed-size *digest*.
  - H("The latest set of transactions were: ...") = 0x708a87a29a18b42d0b2ec47b18814d201c88c33e6e7642c047720812f30f082d
  - Common digest sizes are 256, 384, 512 bits
- Security properties of cryptographic hashes include:
  - Given a digest, you can't find an input that hashes to it
  - You can't find two different inputs with the same digest

SHA

- The most commonly used hash functions are the SHA2 family
  - SHA2-256, SHA2-384, SHA2-512
  - (often shortened as SHA256, SHA384, SHA512)
- Also implemented in hardware on modern CPUs, 1–2 GB/s
  - `openssl speed sha`
- Look for "`sha_ni`" in the "`flags`" line of `/proc/cpuinfo` to see if your CPU does SHA2-256 in hardware

Message authentication codes (MACs)

- Anyone can compute a hash function
- So if an adversary changes a message, they can also change the hash to match
- A MAC is like a "keyed hash": only someone with the (symmetric) key can compute the output (for MACs, called a *tag*)
  - Keys and tags are typically 128/192/256 bits
- MACs can be built from AES or SHA, so can also be fast
- A key component of symmetric-key *authenticity*

AEAD

- Putting the pieces together: encryption and MACs are usually combined into a single operation called "authenticated encryption with associated data" (AEAD).
- Inputs:
  - A key (128/192/256 bits): known only to Alice and Bob
  - A *nonce* (96/144/192 bits): this can be public, but *must be used only once*! (Security falls apart if you reuse a nonce with the same key!)
  - The plaintext to encrypt (arbitrary length, not just 128 bits)
  - Associated public data (e.g., the sequence number of the message)
- Output:
  - Ciphertext (usually the same length as the plaintext)
  - Tag (128/192/256 bits)

AEAD

- Alice sends the nonce, associated data, ciphertext, and tag to Bob
  - Sometimes Bob can figure out the nonce and/or associated data on his own, and then Alice doesn't need to send them
- Bob uses those four items (plus the key) to "open" (decrypt) the ciphertext and tag
- If any of the items have been modified, the opening will fail
- All of confidentiality, integrity, and authenticity are provided in a single operation

Pseudo-random generators (PRGs)

- Another useful building block that will show up in this course is a PRG
- A PRG takes an input key of fixed size (typically 128/192/256 bits) and produces an arbitrary-length (longer) output
- The main security property is that given any amount of the output (but not the input key), that gives you no help in determining the rest of the output, which will look random to you.
- Can be built from AES, hash functions, or MACs

Public-key cryptography

- A major shortcoming of symmetric-key cryptography is that Alice and Bob need to share a symmetric key in advance of communicating

- When you communicate online with Amazon.com's web server, did you first go to Amazon headquarters to arrange a symmetric key with them?

- Public-key cryptography allows Alice and Bob to create a symmetric key using only *public* (but *authenticated*) information
  - Alice needs to know Bob's public key, and be assured that it actually is Bob's key and not the key of someone else pretending to be Bob

Public-key cryptography

- Public-key cryptography is commonly based on *group theory*
  - There are other options as well, particularly when you want to defend against future quantum computers
  - We won't be using those in this course
- Again, we won't be going "inside" the math (much, but a little more than for AES), but we'll give a programmer-style "API"

The API of group theory

- In an API, you need to know:
  - What data types there are
  - What operations you can do on them
- The group theory API has two data types:
  - Points
  - Scalars
- A group is a set of points (with certain properties)
- The number of points in a group is called the *order* of the group
- In cryptography, we usually like to work in groups where the order is a prime number (often denoted q)

Scalars

- Scalars are just ordinary integers, taken modulo q
  - (often written $\mathbb{Z}_q$)
- So if you subtract the scalar 5 from the scalar 3, you get the scalar $q - 2$.
- You can do all the ordinary math operations on scalars:
  - Addition, subtraction, multiplication
  - Division is really multiplication by the *inverse* of b: $\frac{a}{b} = a \cdot b^{-1}$
  - So you can also compute inverses in order to do division (except there's no inverse of 0, since you can't divide by 0)

Points

- There are unfortunately *two* common notations for points in group theory: *additive* and *multiplicative*. Both are extremely commonly used. We will use additive notation in this course, but we'll show you both notations on the next couple of slides.
- Recall a group is a set of q points. Two of these points are special:
  - The *identity* point
  - The *generator* point (also called the *basepoint*)

	Additive	Multiplicative
Identity	$\mathcal{O}$	1
Basepoint / generator	B	g

Operations on points

- General points are typically denoted by uppercase letters in additive notation, and lowercase letters in multiplicative notation. (Scalars are typically lowercase in both notations.)

- There is one operation that is done on a single point (negation / inversion), one that is done on two points (addition / multiplication), and one that is done between a scalar and a point (scalar multiplication / exponentiation)

Operations on points

	Additive	Multiplicative	
Negation	$-X$	g^{-1}	Inversion
Addition	$X + Y$	gh (or $g \cdot h$)	Multiplication
Scalar multiplication	aX (or $a \cdot X$)	g^a	Exponentiation

- Recall that scalars are just ordinary integers (mod q)
- So $5X$ for example just means the same thing as $X + X + X + X + X$
  - Or in multiplicative notation, $g^5 = g \cdot g \cdot g \cdot g \cdot g$
- The particular group we'll be using in this course (e.g., in Assignment 1) is called Ristretto255.

Diffie-Hellman

- One of the most basic parts of public-key cryptography
- Private keys are scalars, public keys are points
- If the private key is x, the public key is $X = xB$
  - Recall B is the basepoint (generator)
  - It is common that the public key will be the uppercase letter corresponding to the private key (X, x in this example)

Diffie-Hellman

- Alice picks private key x, computes public key $X = xB$, sends X to Bob
- Bob picks private key y, computes public key $Y = yB$, sends Y to Alice
- Alice computes $P_A = xY$, Bob computes $P_B = yX$
- Note! $P_A = xY = x(yB) = (xy)B = (yx)B = y(xB) = yX = P_B$
- Alice and Bob each compute $\kappa = H(P)$ (where $P = P_A = P_B$)
- They now can use κ as a shared symmetric key

El Gamal

- The El Gamal public-key encryption system is basically:
  - Do Diffie-Hellman to get a shared secret key
  - Encrypt the message with that key
  - But typically Bob (the recipient of the message) publishes his key ahead of time
- (Ahead of time) Bob picks a private key y, publishes $Y = yB$
- Alice wants to send the message m to Bob. Alice picks a private key x, computes $X = xB$, $P = xY$, and $\kappa = H(P)$
- Alice encrypts m with the key κ using, for example, AEAD
- Alice sends to Bob X and the AEAD nonce, ciphertext, and tag
- Bob computes $P = yX$, $\kappa = H(P)$ and decrypts the AEAD message

Commitments

- A *commitment* is kind of a public-key version of a hash
- The input is a *random* scalar x, and the output is the point $X = xB$
  - A public key is just a commitment to a private key
- Similar properties to a hash:
  - Given the output X, it's hard to find the input x
  - You can't find two different scalars with the same commitment
- But different from a hash:
  - The inputs are fixed size, not arbitrary size
  - Commitments are *homomorphic*:
  - If $X = xB$ is a commitment to x and $Y = yB$ is a commitment to y, then $X + Y = xB + yB = (x + y)B$ is a commitment to $x + y$
  - This is extremely useful, and not something you can do with a hash function

Digital signatures

- To get integrity and authenticity in a public-key setting, we use *digital signatures*
- Again, Alice has a private key (typically a scalar a) and a public key (typically $A = aB$)
- Alice publishes A (ahead of time)

Digital signatures

- When Alice *sends* a message, she *signs* it using a to produce a signature σ, which she attaches to the message
- Anyone can *verify* that σ is a valid signature on the message under the public verification key A
- So only Alice could have produced that signature, and the message has not been modified since Alice did so

Digital signatures

- Two common digital signature schemes are ECDSA and Schnorr
  - Don't be confused! EdDSA is a digital signature scheme that's actually Schnorr (using a particular kind of group), even though its name is very similar to ECDSA
- Schnorr is better in many ways, but ECDSA was invented because Schnorr was patented, and people wanted to use something free of patents
- The patent has expired, and people are starting to move away from ECDSA (e.g., Bitcoin)

Secret sharing

- Private keys (or other secrets) can have high value
- Cryptocurrency wallets are an obvious example, where the private key directly allows you to send the value elsewhere
- Also for critical infrastructure securing TLS, DNS, etc.
- Having the key just sitting on a computer somewhere invites attackers, and is very fragile

Secret sharing

- Strategy: break the secret into a number of pieces, called *shares*, and give each share to a different party (or machine)
- But do so in a way that each share contains *no information* about the secret
  - As opposed to putting the first 20 bytes on one machine, the next 20 on the second machine, and so on
- So if one of the machines is attacked, the attacker gains no information about the secret

Additive/XOR secret sharing

- The simplest case is when you want to split the secret across n parties, and you require all n to come together in order to reconstruct the secret (in order to use it to sign a message, for example)
- If the secret s is a scalar in $\mathbb{Z}_q$, for example, pick $n-1$ random scalars $s_1, s_2, \ldots, s_{n-1}$, and let $s_n = s - (s_1 + s_2 + \ldots + s_{n-1})$.
- Give s_i to party i as their share
- Note that all the shares add up to s
- But even if an attacker gets $n-1$ of the shares, they learn *nothing* about s
- If the secret is a string instead of a number, use XOR ($\oplus$) instead of $+$ and $-$

Threshold secret sharing

- The downside of additive/XOR secret sharing is that of *availability*: if even one of the parties/machines is unavailable or crashes, the secret is unusable, or even *gone forever*
- A more flexible approach is *threshold secret sharing*: divide the secret s into n shares, but only require t (where $1 \leq t \leq n$) shares to have to come together in order to reconstruct s
  - And importantly, even if an attacker learns $t - 1$ of the shares, they gain *no information* about s

Threshold secret sharing

- Threshold secret sharing schemes can have some nice properties, including:
  - Improved resilience to unavailable shares
  - Reconstruction of lost shares
  - Proactive protection against attackers compromising shares one at a time
  - Being able to adjust n or t "on the fly" after the secret is shared, without having to recombine the shares to reconstruct s
- But: lower resilience to an attacker, since an attacker only has to compromise t shares in order to learn (and use) s, instead of all n

Threshold secret sharing

We will look at two kinds of threshold secret sharing:

- Shamir secret sharing
- Replicated secret sharing

Shamir secret sharing

- Suppose the secret s is a private key (a scalar in $\mathbb{Z}_q$)
- Similarly to additive secret sharing, pick some random scalars $a_1, a_2, \ldots, a_{t-1}$
    - Note that unlike additive secret sharing, there are only $t-1$ of them, not $n-1$
    - Also, these random scalars will *not* directly be any of the shares, but they will be used to construct (all n) shares

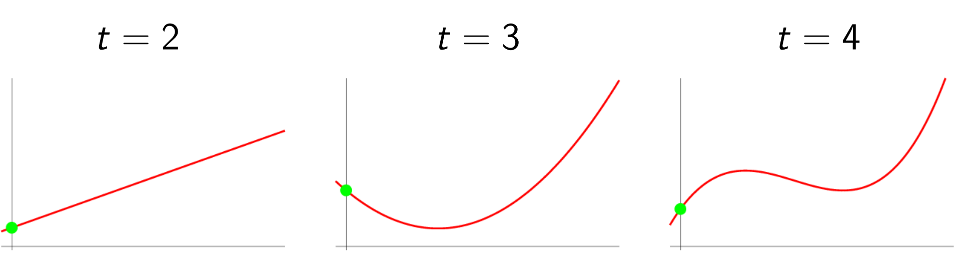
Shamir secret sharing

- Define the polynomial

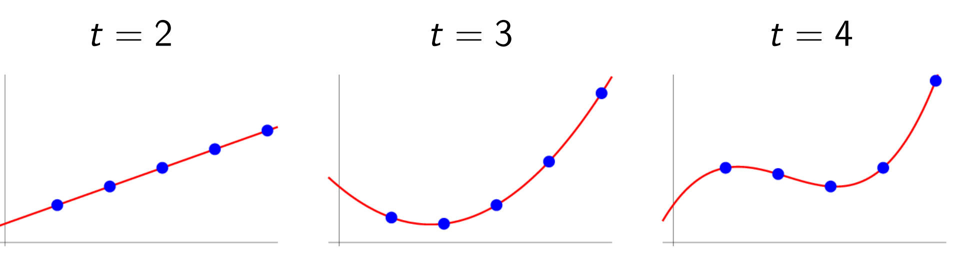
$$f(x) = s + a_1 x + a_2 x^2 + \cdots + a_{t-1} x^{t-1}$$

- Compute $s_1 = f(1)$, $s_2 = f(2)$, $\ldots$, $s_n = f(n)$
- Give s_i to party i as their share
- What is $f(0)$?

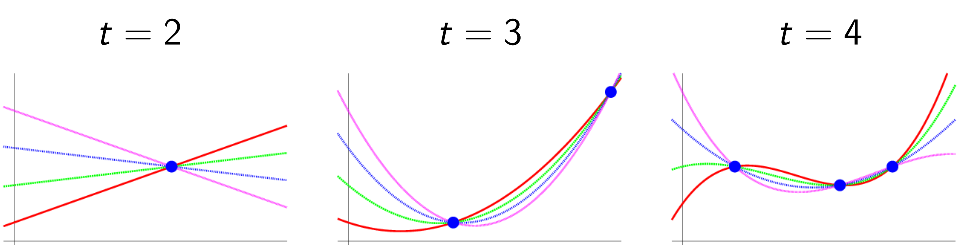
Why does this work?



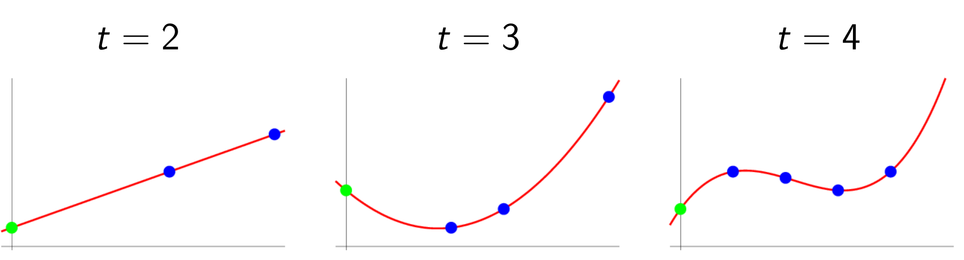
Why does this work?



Why does this work?



Why does this work?



Reconstructing the secret

- t of the parties come together and want to reconstruct the secret
- Suppose the parties are party numbers $x_1, x_2, \ldots, x_t$
  - For example, if $n = 7$ and $t = 3$, you might have parties 2, 5, and 7 coming together, so x_1 would be 2, $x_2 = 5$, and $x_3 = 7$.
- Given only the party numbers (and not yet the values of their shares), compute the t *Lagrange coefficients*:

$$\lambda_i = \prod_{j=1, j \neq i}^{t} \frac{x_j}{x_j - x_i} \quad \text{for } i = 1, \ldots, t$$

Reconstructing the secret

$$x_1 = 2, x_2 = 5, x_3 = 7$$

$$\lambda_i = \prod_{j=1, j \neq i}^{t} \frac{x_j}{x_j - x_i} \quad \text{for } i = 1, \ldots, t$$

- $\lambda_1 = \frac{5}{5-2} \cdot \frac{7}{7-2} = \frac{35}{15} \pmod{q}$
- $\lambda_2 = \frac{2}{2-5} \cdot \frac{7}{7-5} = \frac{14}{-6} \pmod{q}$
- $\lambda_3 = \frac{2}{2-7} \cdot \frac{5}{5-7} = \frac{10}{10} \pmod{q}$

Reconstructing the secret

- Once you have the Lagrange coefficients $\lambda_1, \ldots, \lambda_t$, gather the shares themselves; let y_i be the share of party x_i
- Then the secret s is just:

$$s = \lambda_1 \cdot y_1 + \lambda_2 \cdot y_2 + \cdots + \lambda_t \cdot y_t$$

- This procedure is called *Lagrange interpolation*

Replicated secret sharing

- One last type of secret sharing we will look at is *replicated secret sharing*
- It is used in some distributed trust private computation protocols
- Recall there are n parties, and we want t of them to be able to recover the secret s, but an attacker compromising $t-1$ of them learns *nothing* about s
- How many possible sets of $t-1$ parties are there that the attacker might compromise?

Replicated secret sharing

- One last type of secret sharing we will look at is *replicated secret sharing*
- It is used in some distributed trust private computation protocols
- Recall there are n parties, and we want t of them to be able to recover the secret s, but an attacker compromising $t-1$ of them learns *nothing* about s
- How many possible sets of $t-1$ parties are there that the attacker might compromise?
  - $\binom{n}{t-1}$

Replicated secret sharing

- How many possible sets of $t-1$ parties are there that the attacker might compromise?
  - $\binom{n}{t-1}$
- Idea: additively share s into $\binom{n}{t-1}$ pieces
- For each set of $t-1$ parties the attacker might compromise, give one of the pieces to *everyone else*
- That way, whichever set of $t-1$ parties the attacker compromises, there's one piece of s the attacker doesn't know
- And so the attacker has no information about s
- (But any t parties between them have all the pieces of s and so can reconstruct s)

Replicated secret sharing

Example: $n = 5$, $t = 3$

- $\binom{5}{3-1} = 10$, so additively split s into 10 pieces so that $s_1 + s_2 + \cdots + s_{10} = s$

- Party 1 gets: s_5, s_6, s_7, s_8, s_9, s_{10}
- Party 2 gets: s_2, s_3, s_4, s_8, s_9, s_{10}
- Party 3 gets: s_1, s_3, s_4, s_6, s_7, s_{10}
- Party 4 gets: s_1, s_2, s_4, s_5, s_7, s_9
- Party 5 gets: s_1, s_2, s_3, s_5, s_6, s_8

- Check: any 2 parties are missing one of the pieces, but any 3 parties have all the pieces

Replicated secret sharing

- Problem: each party has to hold on to $\binom{n-1}{t-1}$ pieces
- This could be very large! For $n = 30$ and $t = 10$, for example, this is 10,015,005 pieces per party, and for $n = 40$ and $t = 20$, it's over 68 *billion* pieces per party.
- But for small n and t, in particular $n = 3$ and $t = 2$, this method is quite efficient:
  - Party 1 gets s_2, s_3
  - Party 2 gets s_1, s_3
  - Party 3 gets s_1, s_2

Threat models

- So far, we've been considering an (external) attacker that compromises otherwise-honest parties
- But what if the parties might themselves be adversarial?
- Whenever we make a system to protect security or privacy, we need to say what our *threat model* is
  - These are the kinds of attackers we do, and do not, aim to protect against
  - And ideally, say what goes wrong with the system if there's an attacker you didn't plan to protect against

Threat models

- There are usually a number of different kinds of entities in a system:
  - Servers (possibly multiple different roles)
  - Clients
  - Data subjects
  - Third parties

- Any of these could be adversarial, trying to make the security or privacy system fail in some way

- For each entity (including third parties not directly participating in the system), we need to say what kinds of behaviours we *do* and *do not* aim to protect against

Threat models

- We commonly group behaviours into three categories:
  - Semi-honest (aka “honest-but-curious”): a semi-honest party will participate in the system correctly, but will try to learn things (that it’s not supposed to find out) from the messages it sees from the other parties
  - Malicious: a malicious party is allowed to do anything at all in an attempt to break the system, including sending garbage messages, stopping to participate entirely, or sending messages that cause other participants to misbehave
  - Covert: in between these two extremes are covert adversaries, who are allowed to change their behaviour in the system, but *only if they won’t be caught doing so*

Threat models

- All systems must minimally protect against semi-honest parties, or there's no actual security or privacy being provided
- Ideally, the system will protect against all parties acting maliciously
- This is sometimes hard or expensive to do, however
- So protecting against malicious clients and third parties, while protecting only against covert servers, might be a reasonable threat model
- Why would "covert" be a reasonable threat model for the servers participating in a security or privacy system?

Threat models

- One approach to designing security and privacy systems is to start with a design secure only against semi-honest clients and servers
  - These designs are usually much easier to come up with
- Then build on that design to protect against covert or malicious adversaries
  - This can sometimes be done by having the participants in the system *prove* that they are following the protocol correctly, without *revealing* any private data
- For simplicity, in this course we will look mostly at systems with a semi-honest threat model

Zero-knowledge proofs (ZKPs)

- Consider the situation where you're logging in to an online account: you need to prove that you know your password
- Typically, you enter your username *and your password* into the website.
- You haven't just proven that you *know* your password, you've *told* your password to the website.
- But what if the website wasn't the real one (a phishing site, for example)?

Zero-knowledge proofs (ZKPs)

- ZKPs are a technique that allows a prover (often called Peggy) to convince a verifier (often called Victor) that some statement is true, *without revealing any information except that the statement is true*

- So using a ZKP, Peggy could prove to a website that she *knows* her password, without *revealing* the password, for example

- ZKPs are an important tool for designing systems intended to be secure against covert or malicious attackers

Zero-knowledge proofs (ZKPs)

- There are two main kinds of ZKPs: *interactive* and *non-interactive*
- In an interactive ZKP, Peggy tries to convince Victor of a statement by sending him one or more initial *commitments*, then there are one or more rounds of Victor sending some *challenges*, to which Peggy produces the *responses*. If Peggy's responses are correct, Victor can conclude that the statement is true, *but learns nothing else*.
- In an interactive ZKP, *only Victor* is convinced of the truth of the statement. If Peggy wants someone else to believe the statement, she must do the interactive protocol with them separately.

Zero-knowledge proofs (ZKPs)

- In a non-interactive ZKP, Victor's challenges are replaced by the output of a hash function (which anyone can compute). So Peggy just outputs her initial commitments, and the responses to the computed challenges.

- Then, *anyone* can check that the statement is true

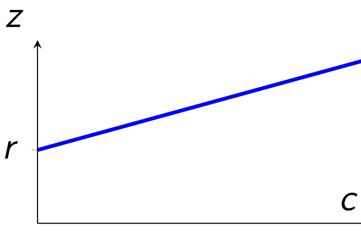
- Slight caveat: if the statement is, for example, "I know Peggy's password", you don't want Victor to then turn around and just show that proof (that he received from Peggy) to someone else.

A simple ZKP

- As an example, suppose Peggy has a private key a with corresponding public key $A = a \cdot B$
- Peggy wants to prove to Victor that she knows a (Victor knows A, which is public)
- The protocol:
  - Peggy picks a random scalar r and computes its commitment $R = r \cdot B$; Peggy sends R to Victor
  - Victor sends a random challenge scalar c to Peggy
  - Peggy replies with the response $z = r + c \cdot a$
  - Victor checks whether $z \cdot B \stackrel{?}{=} R + c \cdot A$

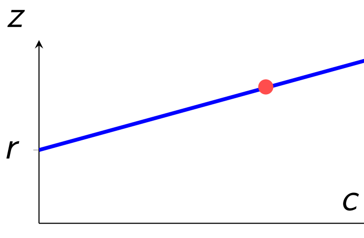
A simple ZKP

- Why does this convince Victor that Peggy knows a?
- Note that $z = r + c \cdot a$ is the equation of a line, where the x-coordinate is c, the y-coordinate is z, the slope is a, and the y-intercept is r:



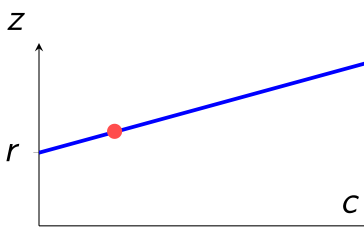
A simple ZKP

- Why does this convince Victor that Peggy knows a?
- Note that $z = r + c \cdot a$ is the equation of a line, where the x-coordinate is c, the y-coordinate is z, the slope is a, and the y-intercept is r:



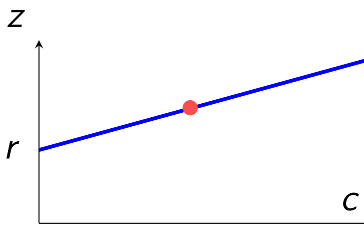
A simple ZKP

- Why does this convince Victor that Peggy knows a?
- Note that $z = r + c \cdot a$ is the equation of a line, where the x-coordinate is c, the y-coordinate is z, the slope is a, and the y-intercept is r:



A simple ZKP

- Why does this convince Victor that Peggy knows a?
- Note that $z = r + c \cdot a$ is the equation of a line, where the x-coordinate is c, the y-coordinate is z, the slope is a, and the y-intercept is r:



Making it non-interactive

- Instead of Victor sending c, compute c as a hash:
  - Peggy picks a random scalar r and computes its commitment $R = r \cdot B$; Peggy sends R to Victor
  - Victor sends a random challenge scalar c to Peggy
  - Peggy replies with the response $z = r + c \cdot a$
  - Victor checks whether $z \cdot B \stackrel{?}{=} R + c \cdot A$

Making it non-interactive

- Instead of Victor sending c, compute c as a hash:
  - Peggy picks a random scalar r and computes its commitment $R = r \cdot B$; Peggy sends R to Victor
  - Peggy computes $c = H_c(R, A, m)$
  - Peggy replies with the response $z = r + c \cdot a$
  - Victor checks whether $z \cdot B \stackrel{?}{=} R + c \cdot A$

Making it non-interactive

- Instead of Victor sending c, compute c as a hash:
  - Peggy picks a random scalar r and computes its commitment $R = r \cdot B$; Peggy sends R to Victor
  - Peggy computes $c = H_c(R, A, m)$
  - Peggy replies with the response $z = r + c \cdot a$
  - Victor computes $c = H_c(R, A, m)$
  - Victor checks whether $z \cdot B \stackrel{?}{=} R + c \cdot A$

Making it non-interactive

- Instead of Victor sending c, compute c as a hash:
  - Peggy picks a random scalar r and computes its commitment $R = r \cdot B$; Peggy sends R to Victor
  - Peggy computes $c = H_c(R, A, m)$
  - Peggy replies with the response $z = r + c \cdot a$
  - Victor computes $c = H_c(R, A, m)$
  - Victor checks whether $z \cdot B \stackrel{?}{=} R + c \cdot A$

- H_c is a hash function that outputs a *scalar* instead of a digest
  - Typically implemented as a normal hash function that outputs a digest, and then a function that can convert a digest into a scalar
  - m is an arbitrary *message*; including it in the hash allows Peggy to "bind" the ZKP to a particular message so that it can't be reused
  - This is one way to make digital signatures!