PRNGs and Diffie-Hellman Key Exchange

Last Time: Hashes

- Map arbitrary-length input to fixed-length output
- Output is deterministic and unpredictable
- Security properties
  - One way: Given an output y, it is infeasible to find any input x such that $H(x) = y$.
  - Collision resistant: It is infeasible to find another any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.
- Some hashes are vulnerable to length extension attacks
- Hashes don't provide integrity (unless you can publish the hash securely)

Last Time: MACs

- Inputs: a secret key and a message
- Output: a tag on the message
- A secure MAC is unforgeable: Even if Mallory can trick Alice into creating MACs for messages that Mallory chooses, Mallory cannot create a valid MAC on a message that she hasn't seen before
- Example: HMAC(K, M) = $H((K' \oplus opad) \,||\, H((K' \oplus ipad) \,||\, M))$
- MACs do not provide confidentiality

Last Time: Authenticated Encryption

- Authenticated encryption: A scheme that simultaneously guarantees confidentiality and integrity (and authenticity) on a message
- First approach: Combine schemes that provide confidentiality with schemes that provide integrity and authenticity
  - MAC-then-encrypt: $\mathrm{Enc}(K_1, M \,||\, \mathrm{MAC}(K_2, M))$
  - Encrypt-then-MAC: $\mathrm{Enc}(K_1, M) \,||\, \mathrm{MAC}(K_2, \mathrm{Enc}(K_1, M))$
  - Always use Encrypt-then-MAC because it's more robust to mistakes
- Second approach: Use AEAD encryption modes designed to provide confidentiality, integrity, and authenticity
  - Drawback: Incorrectly using AEAD modes leads to losing *both* confidentiality and integrity/authentication

Today: PRNGs and Diffie-Hellman Key Exchange

- Symmetric-key encryption schemes need randomness. How do we securely generate random numbers?

- When discussing symmetric-key schemes, we assumed Alice and Bob managed to share a secret key. How can Alice and Bob share a symmetric key over an insecure channel?

Pseudorandom Number Generators (PRNGs)

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)
- Key management (certificates)
- Password management

Randomness

- Randomness is essential for symmetric-key encryption
  - A random key
  - A random IV/nonce
  - Universally unique identifiers (we'll see this shortly)
  - We'll see more applications later
- If an attacker can predict a random number, things can catastrophically fail
- How do we securely generate random numbers?

Entropy

ITIS 6200 / 8200

- In cryptography, “random” usually means “random and unpredictable”
- Scenario
  - You want to generate a secret bitstring that the attacker can't guess
  - You generate random bits by tossing a fair (50-50) coin
  - The outcomes of the fair coin are harder for the attacker to guess
- **Entropy**: A measure of uncertainty
  - In other words, a measure of how unpredictable the outcomes are
  - High entropy = unpredictable outcomes = desirable in cryptography
  - The uniform distribution has the highest entropy (every outcome equally likely, e.g. fair coin toss)
  - Usually measured in bits (so 3 bits of entropy = uniform, random distribution over 8 values)

True Randomness

ITIS 6200 / 8200

- To generate truly random numbers, we need a physical source of entropy
  - Lava lamps
  - Earthquake strength or interval
  - An unpredictable circuit on a CPU
  - Human activity measured at very fine time scales (e.g. the microsecond you pressed a key)
- Unbiased entropy usually requires combining multiple entropy sources
  - Goal: Total number of bits of entropy is the sum of all the input numbers of bits of entropy
    - Many poor sources + 1 good source = good entropy
- Issues with true randomness
  - It's expensive and slow to generate
  - Physical entropy sources are often biased

Exotic entropy source: Cloudflare has a wall of lava lamps that are recorded by an HD video camera that views the lamps through a rotating prism

Pseudorandom Number Generators (PRNGs)

- True randomness is expensive and biased
- **Pseudorandom number generator** (**PRNGs**): An algorithm that uses a little bit of true randomness to generate a lot of random-looking output
  - Also called **deterministic random bit generators** (**DRBGs**)
- Usage
  - Generate some expensive true randomness (e.g. noisy circuit on your CPU)
  - Use the true randomness as input to the PRNG
  - Generate random-looking numbers quickly and cheaply with the PRNG
- PRNGs are deterministic: Output is generated according to a set algorithm
  - However, for an attacker who can't see the **internal state**, the output is *computationally indistinguishable* from true randomness

PRNG: Definition

ITIS 6200 / 8200

- A PRNG has three functions:
  - PRNG.Seed(randomness): Initializes the internal state using the entropy
    - Input: Some truly random bits
  - PRNG.Reseed(randomness): Updates the internal state using the existing state and the entropy
    - Input: More truly random bits
  - PRNG.Generate(*n*): Generate *n* pseudorandom bits
    - Input: A number *n*
    - Output: *n* pseudorandom bits
    - Updates the internal state as needed
- Properties
  - **Correctness**: Deterministic
  - **Efficiency**: Efficient to generate pseudorandom bits
  - **Security**: Indistinguishability from random
  - **Additional security**: Rollback resistance

PRNG: Seeding and Reseeding

- Recall: Number of bits of entropy should be the sum of the number of bits in all sources

- A PRNG should be seeded with all available sources of entropy
  - Combining many low-entropy sources should result in high-entropy output
  - If one source has 0 entropy, it should not reduce the entropy of the output

- Reseeding is used to add even more entropy as it becomes available
  - Reseeding with 0 entropy should not reduce the entropy of the internal state or output

PRNG: Security

ITIS 6200 / 8200

- Can we design a PRNG that is truly random?
- A PRNG cannot be truly random
  - The output is deterministic given the initial seed
  - If the initial seed is *s* bits long, there are only 2^s possible output sequences
- A secure PRNG is computationally indistinguishable from random to an attacker
  - Game: Present an attacker with a truly random sequence and a sequence outputted from a secure PRNG
  - An attacker should not be able to determine which is which with probability > 1/2 + negligible
- Equivalent definition: An attacker cannot predict future output of the PRNG

Insecure PRNGs: OpenSSL PRNG bug

- What happens if we don't use enough entropy?
- Debian OpenSSL CVE-2008-0166
  - Debian: A Linux distribution
  - OpenSSL: A cryptographic library
  - In "cleaning up" OpenSSL (Debian "bug" #363516), the author "fixed" how OpenSSL seeds random numbers
  - The existing code caused Purify and Valgrind to complain about reading uninitialized memory
  - The cleanup caused the PRNG to only be seeded with the process ID
  - There are only 2^{15} (32,768) possible process IDs, so the PRNG only has 15 bits of entropy
- Easy to deduce private keys generated with the PRNG
  - Set the PRNG to every possible starting state and generate a few private/public key pairs
  - See if the matching public key is anywhere on the Internet

PRNG: Rollback Resistance

- **Rollback resistance**: If the attacker learns the internal PRNG state, they cannot learn anything about previous states or outputs
  - Game: An attacker knows the current internal state of the PRNG and is given a sequence of truly random bits and a sequence of previous output from the PRNG
  - The attacker cannot determine which is which with probability > 1/2
- Rollback resistance is not required in a secure PRNG, but it is a useful property
  - Consider:
    - Alice uses the same PRNG to generate her secret key and the IVs for encryption
    - Mallory compromises the internal state of the PRNG
    - If the PRNG is not rollback resistant, Mallory can derive previous PRNG output… such as the secret key

HMAC-DRBG

- Idea: HMAC output looks unpredictable. Let's use HMAC to build a PRNG!

- HMAC takes two arguments (key and message). Let's keep two values, K (key) and V (value) as internal state

HMAC-DRBG

Seed(s):

$K = 0$

$V = 0$

Initialize internal state

$K = \mathrm{HMAC}(K, V \,||\, \mathrm{0x00} \,||\, s)$

$V = \mathrm{HMAC}(K, V)$

$K = \mathrm{HMAC}(K, V \,||\, \mathrm{0x01} \,||\, s)$

$V = \mathrm{HMAC}(K, V)$

Update internal state with provided entropy

HMAC-DRBG

Reseed(s):

$$K = \mathrm{HMAC}(K,\ V \,||\, \mathrm{0x00} \,||\, s)$$
$$V = \mathrm{HMAC}(K,\ V)$$
$$K = \mathrm{HMAC}(K,\ V \,||\, \mathrm{0x01} \,||\, s)$$
$$V = \mathrm{HMAC}(K,\ V)$$

Update internal state with provided entropy

HMAC-DRBG

Generate(n):

output = ''
while len(output) < n **do**
  V = HMAC(K, V)
  output = output || V
end while

K = HMAC(K, V || 0x00)
V = HMAC(K, V)

return output[:n]

Call HMAC repeatedly to generate random-looking output

Update internal state with no extra entropy

HMAC-DRBG: Security

- Assuming HMAC is secure, HMAC-DRBG is a secure, rollback-resistant PRNG
  - Secure: If you can distinguish PRNG output from random, then you've distinguished HMAC from random
  - Rollback-resistant: If you can derive old output from the current state, then you've reversed the hash function or HMAC
  - The full proof is out of scope
  - In other words: if you break HMAC-DRBG, you've either broken HMAC or the underlying hash function

Insecure PRNGs: Rust Rand_Core

- A Rust library has an interface for “secure” random number generators… but it isn’t actually secure!
- Example: ChaCha8Rng
    - A stream cipher PRNG
    - No reseed function: no way of adding extra entropy after the initial seed
    - Seed only takes 32 bits: no way to combine entropy
    - No rollback resistance
- None of the “secure” RNGs are cryptographically secure
    - None have a reseed function to add extra entropy
    - None take arbitrarily long seeds
- **Takeaway**: Always make sure you use a secure PRNG
    - Consider human factors? Use fail-safe defaults?

Application: Universally Unique Identifiers (UUIDs)

- Scenario
  - You have a set of objects (e.g. files)
  - You need to assign a unique name to every object
  - Every name must be unique and unpredictable
- Solution: Choose a random value
  - If you use enough randomness, the probability of generating the same random value twice are astronomically small (basically 0)
- Universally Unique Identifiers (UUIDs)
  - 128-bit unique values
  - To generate a new UUID, seed a secure PRNG properly, and generate a random value
  - Often written in hexadecimal: **`00112233-4455-6677-8899-aabbccddeeff`**

PRNGs: Summary

- True randomness requires sampling a physical process
  - Slow, expensive, and biased (low entropy)
- PRNG: An algorithm that uses a little bit of true randomness to generate a lot of random-looking output
  - Seed(entropy): Initialize internal state
  - Reseed(entropy): Add additional entropy to the internal state
  - Generate(n): Generate n bits of pseudorandom output
  - Security: Computationally indistinguishable from truly random bits
- CTR-DRBG: Use a block cipher in CTR mode to generate pseudorandom bits
- HMAC-DRBG: Use repeated applications of HMAC to generate pseudorandom bits
- Application: UUIDs

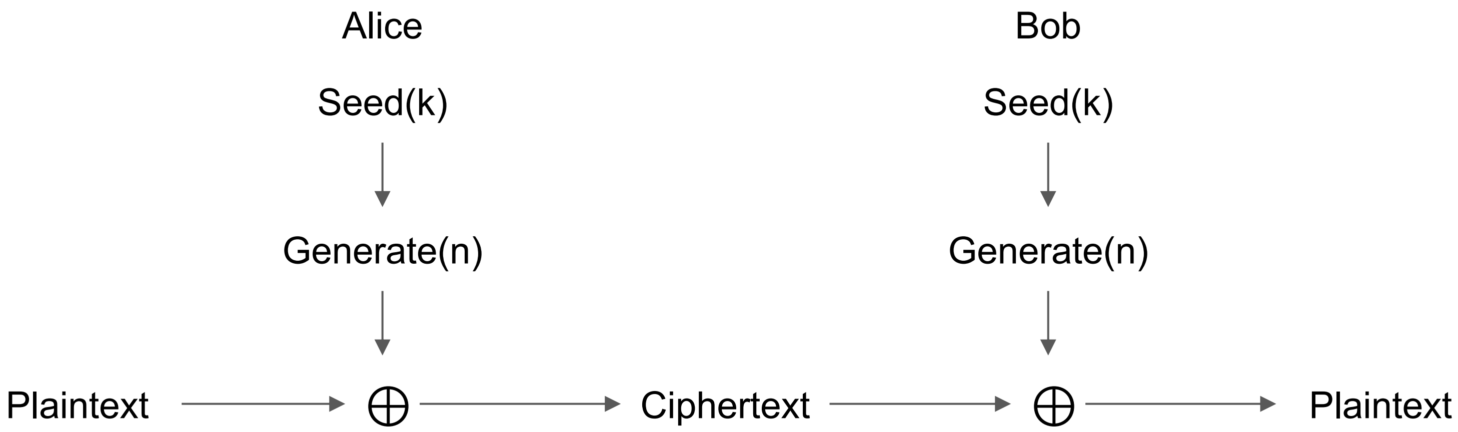
Stream Ciphers

Stream Ciphers

- Another way to construct symmetric key encryption schemes
- Idea
    - A secure PRNG produces output that looks indistinguishable from random
    - An attacker who can’t see the internal PRNG state can’t learn any output
    - What if we used PRNG output as the key to a one-time pad?
- **Stream cipher**: A symmetric encryption algorithm that uses pseudorandom bits as the key to a one-time pad

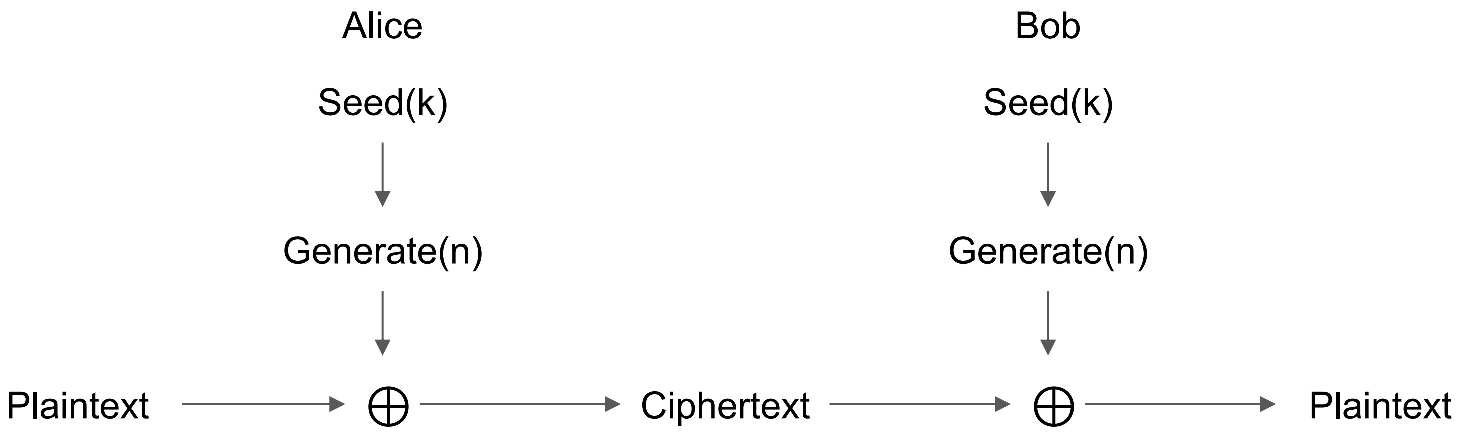
Stream Ciphers

- Protocol: Alice and Bob both seed a secure PRNG with their symmetric secret key, and then use the output as the key for a one-time pad



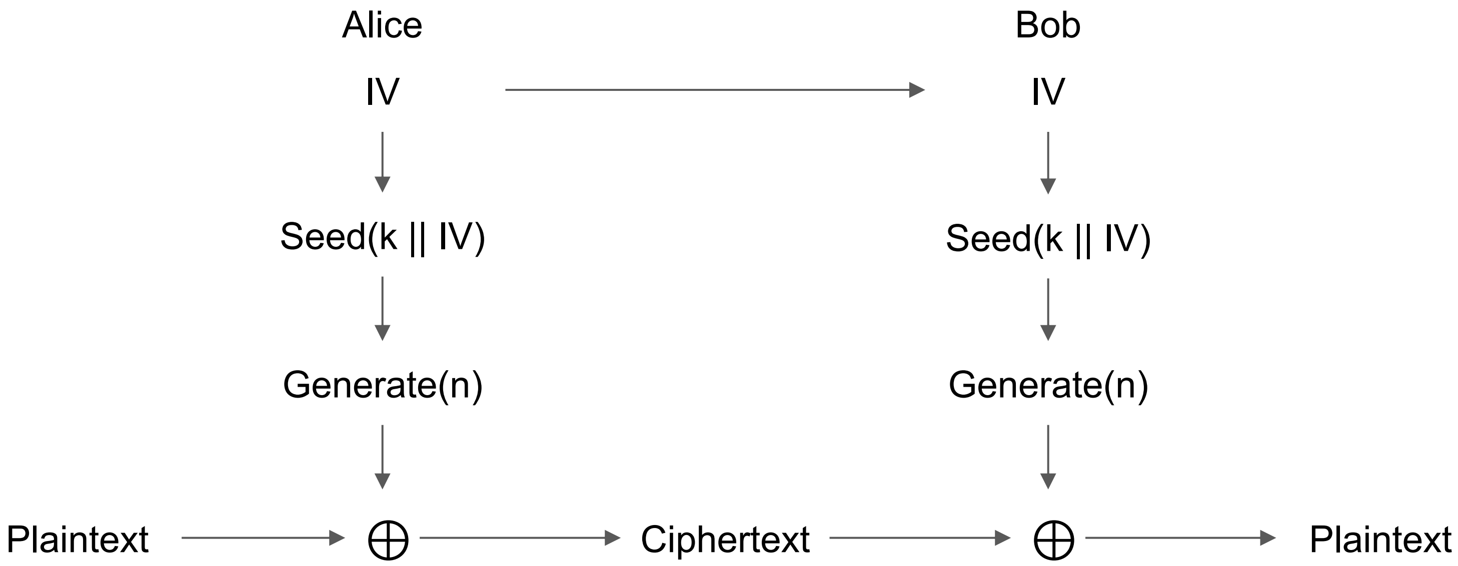
Stream Ciphers: Encrypting Multiple Messages

- Recall: One-time pads are insecure when the key is reused. How do we encrypt multiple messages without key reuse?



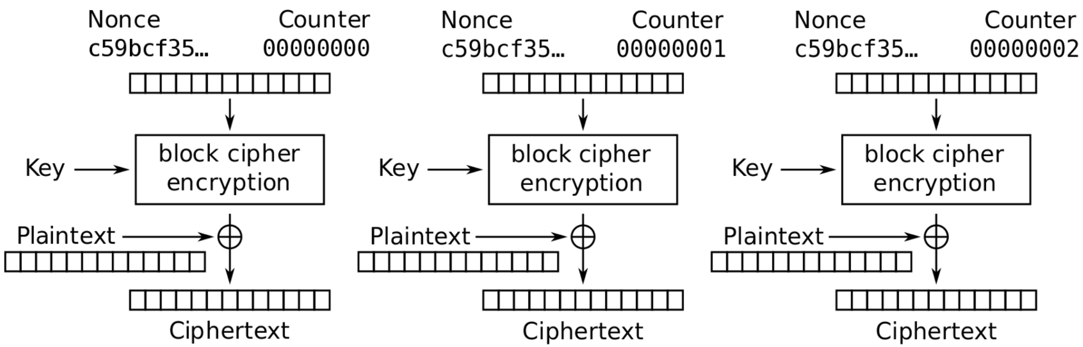
Stream Ciphers: Encrypting Multiple Messages

- Solution: For each message, seed the PRNG with the key and a random IV, concatenated. Send the IV with the ciphertext



Stream Ciphers: AES-CTR

- If you squint carefully, AES-CTR is a type of stream cipher
- Output of the block ciphers is pseudorandom and used as a one-time pad



Counter (CTR) mode encryption

Stream Ciphers: Security

- Stream ciphers are IND-CPA secure, assuming the pseudorandom output is secure

- In some stream ciphers, security is compromised if too much plaintext is encrypted
  - Example: In AES-CTR, if you encrypt so many blocks that the counter wraps around, you'll start reusing keys
  - In practice, if the key is *n* bits long, usually stop after $2^{n/2}$ bits of output
  - Example: In AES-CTR with 128-bit counters, stop after 2^{64} blocks of output

Stream Ciphers: Encryption Efficiency

- Stream ciphers can continually process new elements as they arrive
  - Only need to maintain internal state of the PRNG
  - Keep generating more PRNG output as more input arrives

- Compare to block ciphers: Need modes of operations to handle longer messages, and modes like AES-CBC need padding to function, so doesn't function well on streams

Stream Ciphers: Decryption Efficiency

- Suppose you received a 1 GB ciphertext (encryption of a 1 GB message) and you only wanted to decrypt the last 128 bytes

- Benefit of some stream ciphers: You can decrypt one part of the ciphertext without decrypting the entire ciphertext
  - Example: In AES-CTR, to decrypt only block i, compute E_K(nonce || i) and XOR with the ith block of ciphertext
  - Example: ChaCha20 (another stream cipher) lets you decrypt arbitrary parts of ciphertext
  - What about HMAC-DRBG? You have to generate all the PRNG output up until the block you want to decrypt

Next: Diffie-Hellman Key Exchange

- When discussing symmetric-key schemes, we assumed Alice and Bob managed to share a secret key. How can Alice and Bob share a symmetric key over an insecure channel?

Diffie-Hellman Key Exchange

Cryptography Roadmap

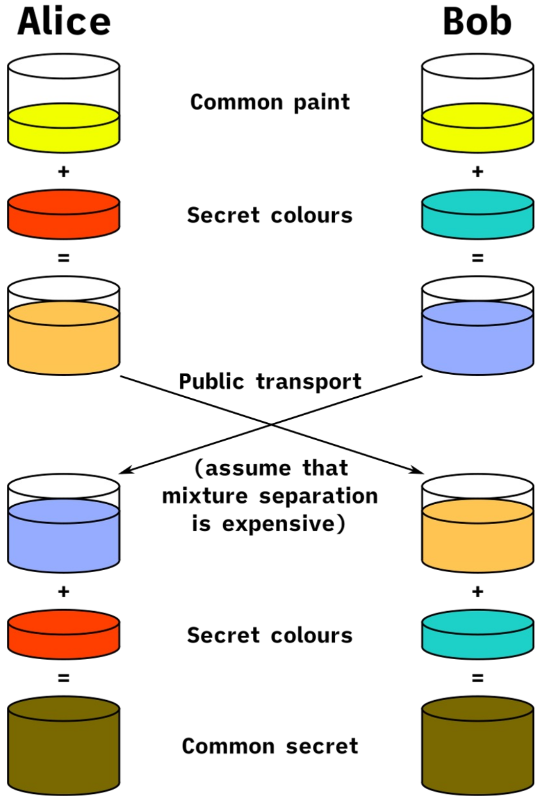
	Symmetric-key	Asymmetric-key
Confidentiality	- One-time pads - Block ciphers with chaining modes (e.g. AES-CBC)	- RSA encryption - ElGamal encryption
Integrity, Authentication	MACs (e.g. HMAC)	Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

- Key management (certificates)
- Password management

Secure Color Sharing

ITIS 6200 / 8200



Secure Color Sharing

- Suppose Alice and Bob want a secret *paint color*, but Eve can see paint colors sent between Alice and Bob
- Alice generates a secret color **amber A**, and Bob generates a secret color **blue B**
- Alice and Bob agree on a common, public color **green G**
- They both mix their secret colors with **G**, so Alice has **green-amber GA**, and Bob has **green-blue GB**
- Alice sends **GA** to Bob, and Bob sends **GB** to Alice
  - Note: Eve now knows the colors **GA** and **GB**! Assume that it is hard to separate colors.
- Alice knows **GB**, so she can mix in **A** to form green-amber-blue **GAB**. Bob knows **GA**, so he can mix in **B** to form **GAB**, as well!
  - Eve only knows **G**, **GA**, and **GB**, so she can only form **green-amber-green-blue GAGB**, which is not the same!

Discrete Log Problem and Diffie-Hellman Problem

ITIS 6200 / 8200

- Recall our paint assumption: Separating a paint mixture is hard
  - Is there a mathematical version of this? Yes!
- Assume everyone knows a large prime p (e.g. 2048 bits long) and a generator g
  - Don't worry about what a generator is
- **Discrete logarithm problem** (**discrete log problem**): Given g, p, g^a mod p for random a, it is computationally hard to find a
- **Diffie-Hellman assumption**: Given g, p, g^a mod p, and g^b mod p for random a, b, no polynomial time attacker can distinguish between a random value R and g^{ab} mod p.
  - Intuition: The best known algorithm is to first calculate a and then compute $(g^b)^a$ mod p, but this requires solving the discrete log problem, which is hard!
  - Note: Multiplying the values doesn't work, since you get g^{a+b} mod $p \neq g^{ab}$ mod p

Diffie-Hellman Key Exchange

Public: g, p

Alice	Eve	Bob
Generate a		Generate b
Calculate $g^a \bmod p$	g^a $\quad$ g^b	Calculate $g^b \bmod p$
Receive $g^b \bmod p$		Receive $g^a \bmod p$
Calculate $(g^b)^a \bmod p$		Calculate $(g^a)^b \bmod p$
$a, g^a, g^b \Rightarrow g^{ab}$	$g^a, g^b \not\Rightarrow g^{ab}$	$b, g^a, g^b \Rightarrow g^{ab}$

Secret key → Generate a

Public key → Calculate $g^a \bmod p$

Shared symmetric key is g^{ab}

Ephemerality of Diffie-Hellman

- Diffie-Hellman can be used ephemerally (called Diffie-Hellman ephemeral, or DHE)
  - **Ephemeral**: Short-term and temporary, not permanent
  - Alice and Bob discard a, b, and $K = g^{ab} \bmod p$ when they're done
  - Because you need a and b to derive K, you can never derive K again!
  - Sometimes K is called a **session key**, because it's only used for a an ephemeral session
- Benefit of DHE: **Forward secrecy**
  - Eve records everything sent over the insecure channel
  - Alice and Bob use DHE to agree on a key $K = g^{ab} \bmod p$
  - Alice and Bob use K as a symmetric key
  - After they're done, discard a, b, and K
  - Later, Eve steals all of Alice and Bob's secrets
  - Eve can't decrypt any messages she recorded: Nobody saved a, b, or K, and her recording only has $g^a \bmod p$ and $g^b \bmod p$!

Diffie-Hellman: Security

Alice

Mallory

Public: g, p

Bob

Alice	Mallory		Bob
Generate a	Generate m		Generate b
Calculate g^a mod p	Calculate g^m mod p		Calculate g^b mod p
Receive g^m mod p	Receive g^a mod p	Receive g^b mod p	Receive g^m mod p
Calculate $(g^m)^a$ mod p	Calculate $(g^a)^m$ mod p	Calculate $(g^b)^m$ mod p	Calculate $(g^m)^b$ mod p
$a, g^a, g^m \Rightarrow g^{am}$	$m, g^m, g^a \Rightarrow g^{am}$	$m, g^m, g^b \Rightarrow g^{bm}$	$b, g^b, g^m \Rightarrow g^{bm}$

Diffie-Hellman: Issues

- Diffie-Hellman is not secure against a MITM adversary
- DHE is an *active protocol*: Alice and Bob need to be online at the same time to exchange keys
    - What if Bob wants to encrypt something and send it to Alice for her to read later?
    - Next time: How do we use *public-key encryption* to send encrypted messages when Alice and Bob don't share keys and aren't online at the same time?
- Diffie-Hellman does not provide *authentication*
    - You exchanged keys with someone, but Diffie-Hellman makes no guarantees about who you exchanged keys with; it could be Mallory!

Elliptic-Curve Diffie-Hellman (ECDH)

- Notice: The discrete-log problem seems hard because exponentiating integers in modular arithmetic “wraps around”
  - Diffie-Hellman can be generalized to any mathematical group that has this cyclic property
  - Discrete-log uses the “multiplicative group of integers mod *p* under generator *g*”
- Elliptic curves: A type of mathematical curve
  - Big idea: Repeatedly adding a point to itself on a curve is another cyclic group
  - You don’t need to understand the math behind elliptic curves
- **Elliptic-curve Diffie-Hellman**: A variation of Diffie-Hellman that uses elliptic curves instead of modular arithmetic
  - Based on the elliptic curve discrete log problem, the analog of the discrete log problem
  - Benefit of ECDH: The underlying problem is harder to solve, so we can use smaller keys (3072-bit DHE is about as secure as 384-bit ECDHE)

Summary: Diffie-Hellman Key Exchange

ITIS 6200 / 8200

- Algorithm:
  - Alice chooses a and sends g^a mod p to Bob
  - Bob chooses b and sends g^b mod p to Alice
  - Their shared secret is $(g^a)^b = (g^b)^a = g^{ab}$ mod p
- Diffie-Hellman provides forwards secrecy: Nothing is saved or can be recorded that can ever recover the key
- Diffie-Hellman can be performed over other mathematical groups, such as elliptic-curve Diffie-Hellman (ECDH)
- Issues
  - *Not* secure against MITM
  - Both parties must be online
  - Does not provide authenticity