

# PRNGs and Diffie-Hellman Key Exchange

# Last Time: Hashes

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- 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:  $\text{HMAC}(K, M) = H((K \oplus \text{opad}) \parallel H((K \oplus \text{ipad}) \parallel M))$
- MACs do not provide confidentiality

# Last Time: Authenticated Encryption

ITIS 6200 / 8200

- 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:  $\text{Enc}(K_1, M \parallel \text{MAC}(K_2, M))$
  - Encrypt-then-MAC:  $\text{Enc}(K_1, M) \parallel \text{MAC}(K_2, \text{Enc}(K_1, M))$
  - 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

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

	Symmetric-key	Asymmetric-key
Confidentiality	<ul style="list-style-type: none"><li>● One-time pads</li><li>● Block ciphers with chaining modes (e.g. AES-CBC)</li></ul>	<ul style="list-style-type: none"><li>● RSA encryption</li><li>● ElGamal encryption</li></ul>
Integrity, Authentication	<ul style="list-style-type: none"><li>● MACs (e.g. HMAC)</li></ul>	<ul style="list-style-type: none"><li>● Digital signatures (e.g. RSA signatures)</li></ul>

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

- Key management (certificates)
- Password management

# Randomness

ITIS 6200 / 8200

- 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)

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- 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  $> 1/2 + \text{negligible}$
- Equivalent definition: An attacker cannot predict future output of the PRNG

# Insecure PRNGs: OpenSSL PRNG bug

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- **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 has used 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

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

Seed(s):

$K = 0$

$V = 0$

Initialize internal state



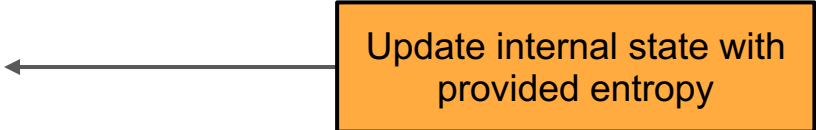
$K = \text{HMAC}(K, V \parallel 0x00 \parallel s)$

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

$K = \text{HMAC}(K, V \parallel 0x01 \parallel s)$

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


Update internal state with  
provided entropy



# HMAC-DRBG

ITIS 6200 / 8200

Reseed(s):

$$K = \text{HMAC}(K, V \parallel 0x00 \parallel s)$$
$$V = \text{HMAC}(K, V)$$
$$K = \text{HMAC}(K, V \parallel 0x01 \parallel s)$$
$$V = \text{HMAC}(K, V)$$


Update internal state with  
provided entropy


# HMAC-DRBG

ITIS 6200 / 8200

Generate( $n$ ):


```
output = ""  
while len(output) <  $n$  do  
     $V = \text{HMAC}(K, V)$   
    output = output ||  $V$   
end while
```

Call HMAC repeatedly to  
generate random-looking output

A grey arrow points from the left side of the orange box to the line  $V = \text{HMAC}(K, V)$  in the code block.

```
 $K = \text{HMAC}(K, V || 0x00)$   
 $V = \text{HMAC}(K, V)$ 
```

Update internal state with no  
extra entropy

A grey arrow points from the left side of the orange box to the line  $V = \text{HMAC}(K, V)$  in the code block.

```
return output[: $n$ ]
```

# HMAC-DRBG: Security

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- A Rust library has an interface for “secure” random number generators... but it isn’t actually secure!
- Example: ChaCha8Rng
  - 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)

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- 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
- HMAC-DRBG: Use repeated applications of HMAC to generate pseudorandom bits
- Application: UUIDs



# Stream Ciphers

# Stream Ciphers

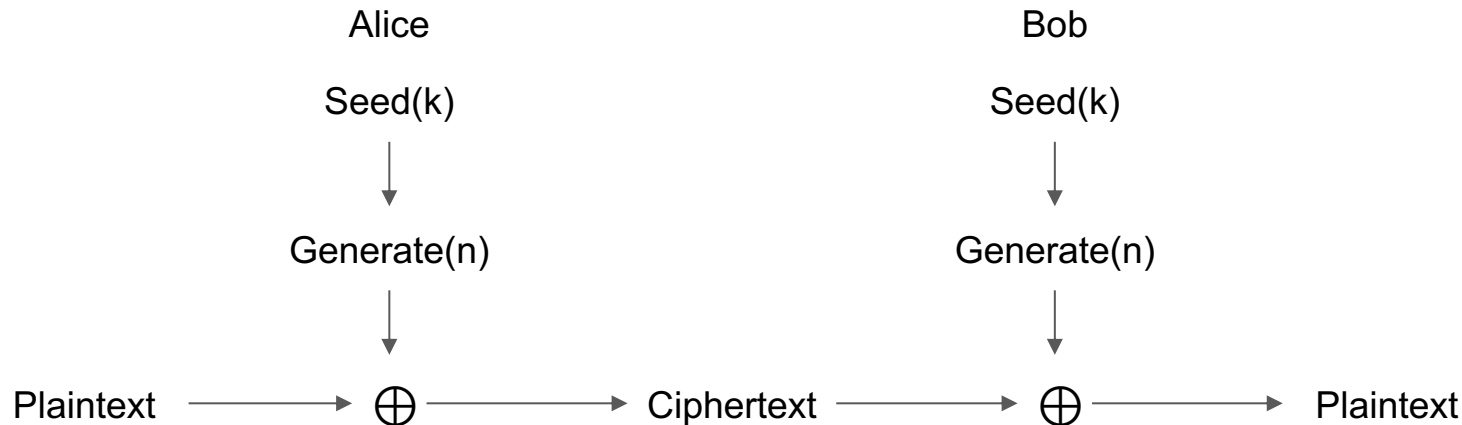
ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

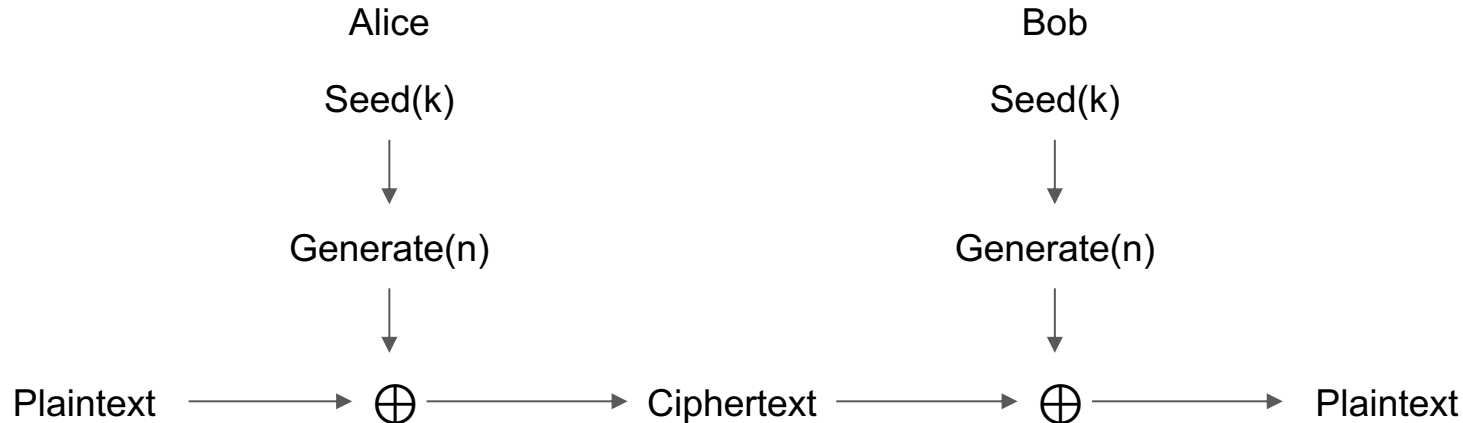
- Protocol: Alice and Bob both seed a secure PRNG with their symmetric secret key, and then use the output as the key for a one-time pad



# Stream Ciphers: Encrypting Multiple Messages

ITIS 6200 / 8200

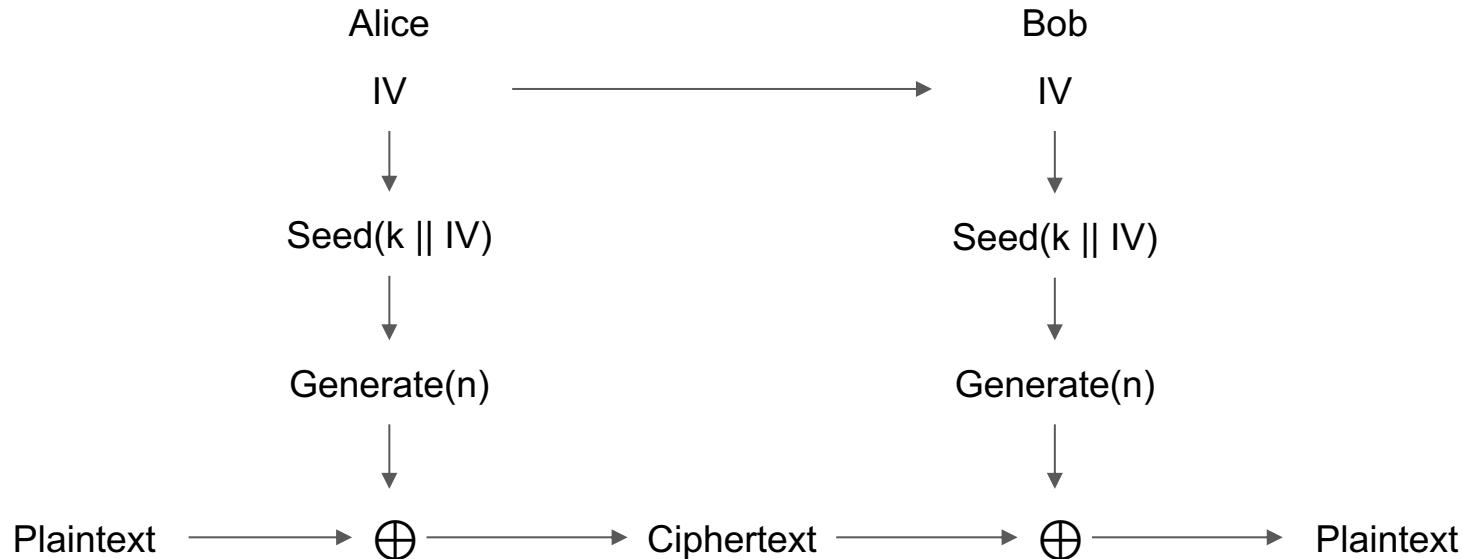
- Recall: One-time pads are insecure when the key is reused. How do we encrypt multiple messages without key reuse?



# Stream Ciphers: Encrypting Multiple Messages

ITIS 6200 / 8200

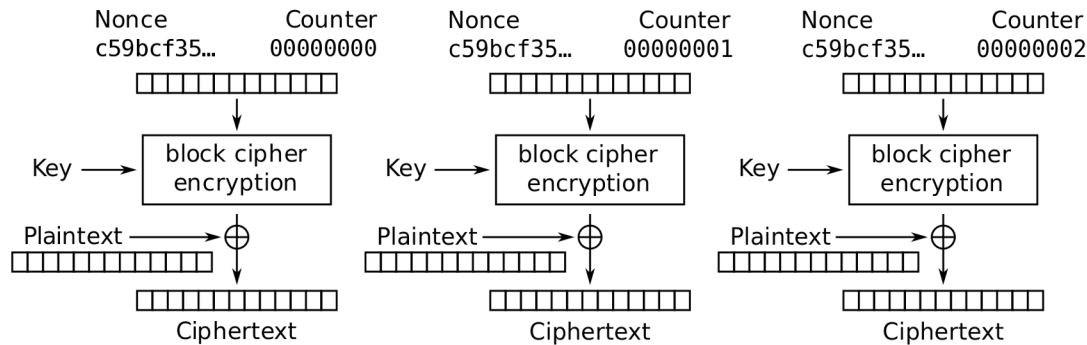
- Solution: For each message, seed the PRNG with the key and a random IV, concatenated. Send the IV with the ciphertext



# Stream Ciphers: AES-CTR

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

- 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  $E_K(\text{nonce} || i)$  and XOR with the  $i$ th 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

# Announcements

ITIS 6200 / 8200

- Assignment 1
  - Due 11:59 PM EST Today
- Assignment 2
  - Release 11:59 AM EST Today
  - Due Sep.26th 11:59 PM EST
  - More questions than the first, start early!

# Next: Diffie-Hellman Key Exchange

ITIS 6200 / 8200

- 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

ITIS 6200 / 8200

	Symmetric-key	Asymmetric-key
Confidentiality	<ul style="list-style-type: none"><li>● One-time pads</li><li>● Block ciphers with chaining modes (e.g. AES-CBC)</li></ul>	<ul style="list-style-type: none"><li>● RSA encryption</li></ul>
Integrity, Authentication	<ul style="list-style-type: none"><li>● MACs (e.g. HMAC)</li></ul>	<ul style="list-style-type: none"><li>● Digital signatures (e.g. RSA signatures)</li></ul>

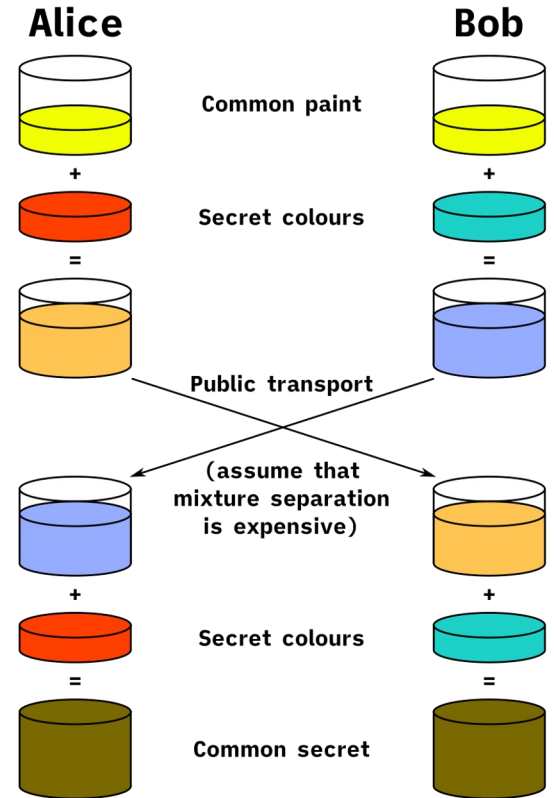
- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

- Key management (certificates)
- Password management

# Secure Color Sharing

ITIS 6200 / 8200

- Suppose Alice and Bob want a secret *paint color*, but Eve can see paint colors sent between Alice and Bob
- Alice and Bob agree on a common, public color **yellow G**
- Alice generates a secret color **red A**, and Bob generates a secret color **cyan B**
- They both mix their secret colors with **G**, so Alice has **red-yellow GA**, and Bob has **cyan-yellow 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 **GAGB**, which is not the same!



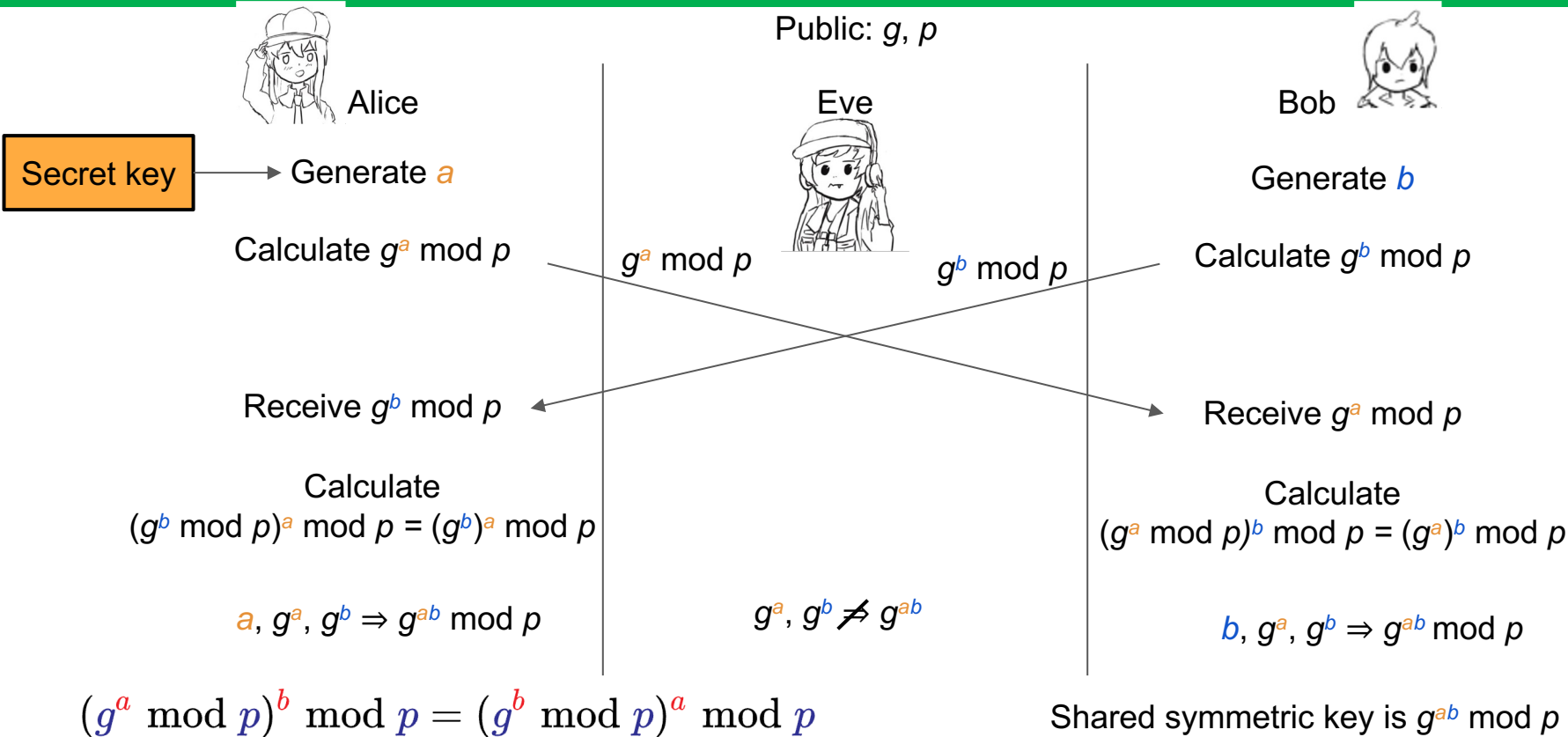
# Discrete Log Problem and Diffie-Hellman Problem

ITIS 6200 / 8200

- Recall our paint assumption: Separating a paint mixture is hard
  - Is there a mathematical version of this?
- 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  $g, p, g^a \bmod p$  for random  $a$ , it is computationally hard to find  $a$
- **Diffie-Hellman assumption:** Given  $g, p, g^a \bmod p$ , and  $g^b \bmod p$  for random  $a, b$ , no polynomial time attacker can distinguish between a random value  $R$  and  $g^{ab} \bmod p$ .
  - Intuition: The best known algorithm is to first calculate  $a$  and then compute  $(g^b)^a \bmod p$ , but this requires solving the discrete log problem, which is hard!

# Diffie-Hellman Key Exchange

ITIS 6200 / 8200





# Diffie-Hellman Example

ITIS 6200 / 8200

- Alice and Bob agree on a prime number  $p$  and a base value  $g$ . Here  $p = 23$  and  $g = 5$
- Alice chooses a secret number,  $a$ , and sends Bob  $A = g^a \bmod p$ . Here  $a = 6$ 
  - $A = 5^6 \bmod 23 = 15625 \bmod 23 = 8$
- Bob chooses a secret number,  $b$ , and sends Alice  $B = g^b \bmod p$ . Here  $b = 15$ 
  - $B = 5^{15} \bmod 23 = 30,517,578,125 \bmod 23 = 19$
- Alice computes  $s = B^a \bmod p$ 
  - $s = 19^6 \bmod 23 = 47,045,881 \bmod 23 = 2$
- Bob computes  $s = A^b \bmod p$ 
  - $s = 8^{15} \bmod 23 = 35,184,372,088,832 \bmod 23 = 2$
- Alice and Bob share a secret value  $s = 2$

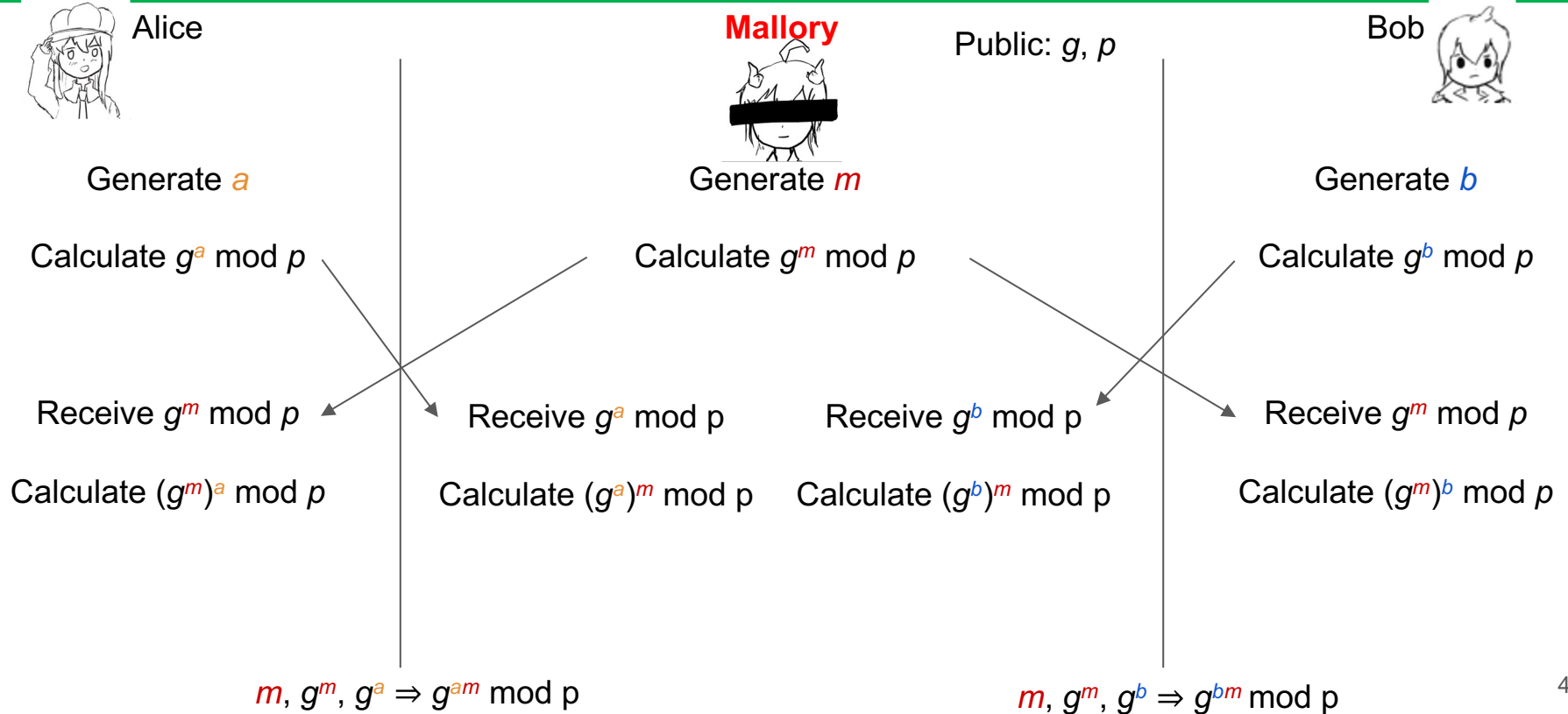
# Ephemerality of Diffie-Hellman

ITIS 6200 / 8200

- 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 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 (Man In the Middle Attack)

ITIS 6200 / 8200



# Diffie-Hellman: Issues

ITIS 6200 / 8200

- 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!

# Summary: Diffie-Hellman Key Exchange

ITIS 6200 / 8200

- Algorithm:
  - Alice chooses  $a$  and sends  $g^a \bmod p$  to Bob
  - Bob chooses  $b$  and sends  $g^b \bmod p$  to Alice
  - Their shared secret is  $(g^a)^b = (g^b)^a = g^{ab} \bmod p$
- Diffie-Hellman provides forwards secrecy: Nothing is saved or can be recorded that can ever recover the key
- Issues
  - *Not* secure against MITM
  - Both parties must be online
  - Does not provide authenticity