IERG 4130 – 2026 Fall

香港中文大學
CUHK

PRNGs and Diffie-Hellman Key Exchange

Yajin ZHOU

https://yajin.org

Credits: the slides are from CS161 of UC Berkeley

How to Provide Integrity

- 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?

Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC) • Stream ciphers	• 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

- **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

- **To generate truly random numbers, we need a physical source of entropy**
  - An unpredictable circuit on a CPU
  - Human activity measured at very fine time scales (e.g. the microsecond you pressed a key)
- **However, true randomness is expensive and slow**



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

True Randomness

- The camera takes the live video feed of the moving lamps distorted by the spinning glass.
- Every pixel of that video is turned into a massive string of numbers.
- Because the fluid inside the lamps changes constantly, the room's lighting shifts, people walk past, and the prism distorts the image, **the resulting numbers are mathematically impossible to predict or replicate.**



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**
- **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 -- it's "about as good as if it actually were random"

PRNG: Definition

- **A PRNG has three functions:**
  - PRNG.Seed(randomness): Initializes the internal state using some random bits
    - Input: Some truly random bits
  - PRNG.Reseed(randomness): Updates the internal state using the existing state and the random bits
    - 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

PRNG: Definition

- **Properties**
  - **Correctness**: Deterministic
  - **Efficiency**: Efficient to generate pseudorandom bits
  - **Security**: Indistinguishability from random
  - **Additional security**: Rollback resistance

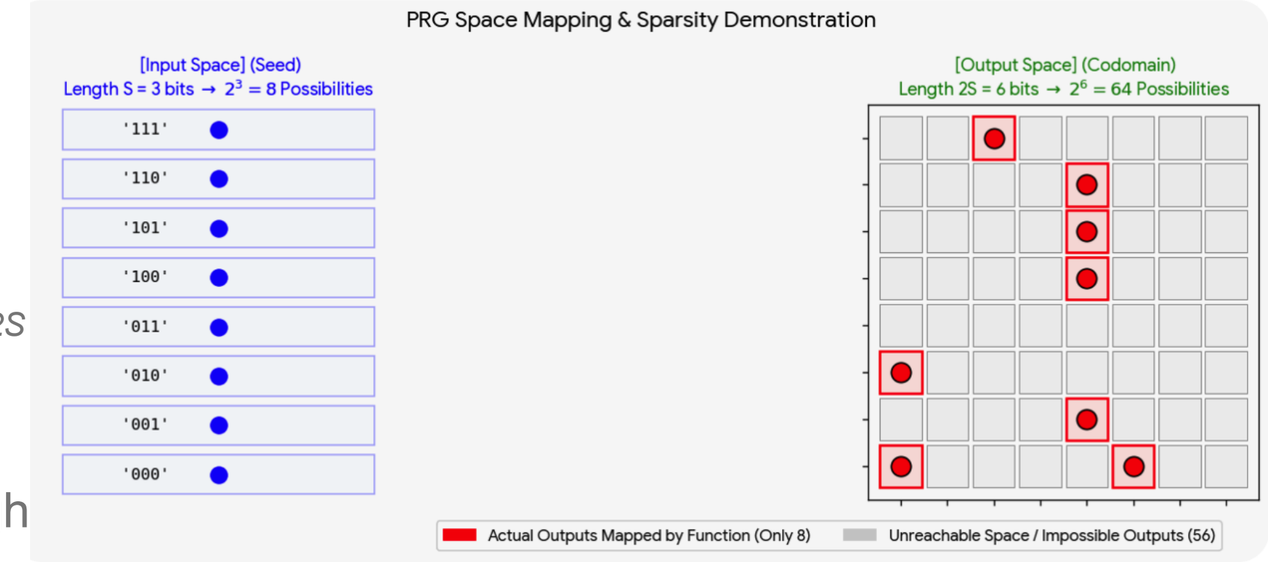
PRNG: Seeding and Reseeding



- **A PRNG should be seeded with all available sources of entropy**
  - As long as one source is good, results will be good
  - If all sources are independent, total entropy is sum of the entropy of each source, so more sources never hurts and can help
- **Reseeding is used to add even more entropy as it becomes available**

PRNG: Security

- **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
  - Even if we can have a function: $\{0, 1\}^S \rightarrow \{0, 1\}^{2S}$, then it does not solve the problem
    - Still has 2^S inputs and 2^S outputs even though the output space is 2^{2S}



PRNG: Security

- **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**

DEMO

- Demo

Real Story

- Coldcard hardware wallet
- 70M USD loss
- How
- https://blocksec.com/blog/coldcard-entropy-failure-seed-recovery



Real Story



- #define MICROPY_HW_ENABLE_RNG (0)

```
1  extern uint32_t rng_get(void);
2  #define CHIP_TRNG_32() rng_get()
3
4  #ifndef MICROPY_HW_ENABLE_RNG
5  #error "get a HW TRNG plz"
6  #endif
```

```
#if MICROPY_HW_ENABLE_RNG
    // STM32 hardware RNG
#else
    // Yasmarang fallback
#endif
```

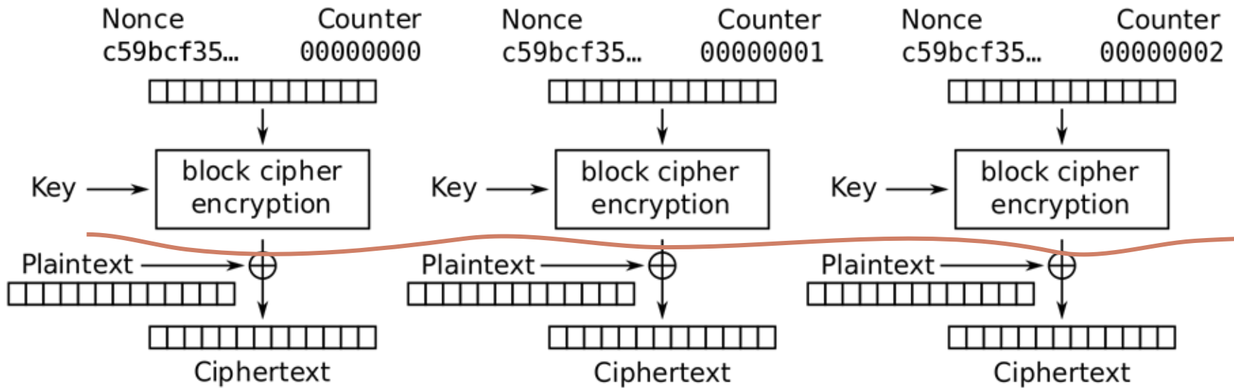
PRNG: Rollback Resistance



- **Rollback resistance**: If the attacker learns the internal PRNG state (e.g., by hacking into the machine and dumping its memory), they cannot learn anything about previous states or outputs
- Rollback resistance is not required in a secure PRNG, but it is a useful property
  - Consider:
    - Alice uses the same PRNG to generate her Bitcoin secret key and other stuff, and stores the Bitcoin secret key offline in "cold storage"
    - Mallory hacks Alice's machine and learns the internal state of the PRNG
    - If the PRNG is not rollback resistant, Mallory can derive previous PRNG output… such as the secret key

CTR-DRBG

- AES in CTR mode generates m pseudo random bits by applying AES m/128 times and concatenating the result
- **Generate(m) = E_k(IV || 1) | E_k(IV || 2) | E_k(IV || 3) … E_k(IV || ceil(m/128))**



Counter (CTR) mode encryption

Pseudorandomness, PRNG output

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

Insecure PRNGs: CVE-2019-16303



- **Relevant if you wrote an app in JHipster before 2019**
- **Password reset functions**
  - When you forget your password, receive an email with a special link to reset your password
  - The special link should contain a randomly-generated code (so attackers can't make their own link)
- **Vulnerability: Bad PRNG**
  - You can figure out the PRNG's internal state from the reset link
  - Request password reset links for other people's accounts
  - Predict the "random" reset link and take over any account you want!

PRNGs: Summary



- **True randomness requires sampling a physical process**
  - Slow and expensive (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**

Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC) • **Stream ciphers**	• 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

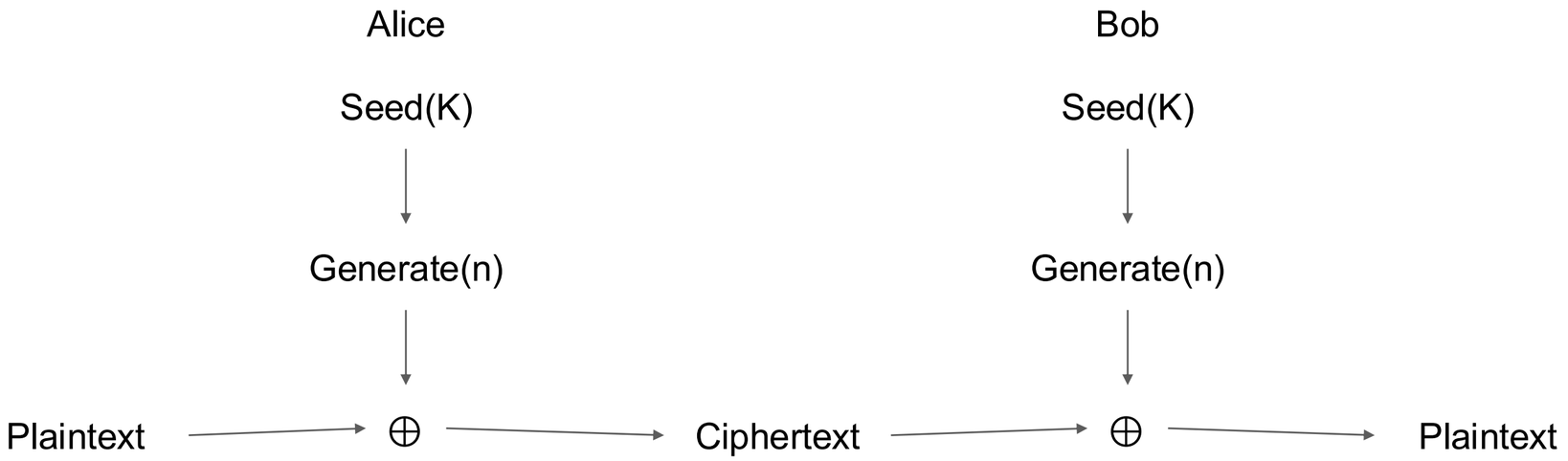
Stream Ciphers



- **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

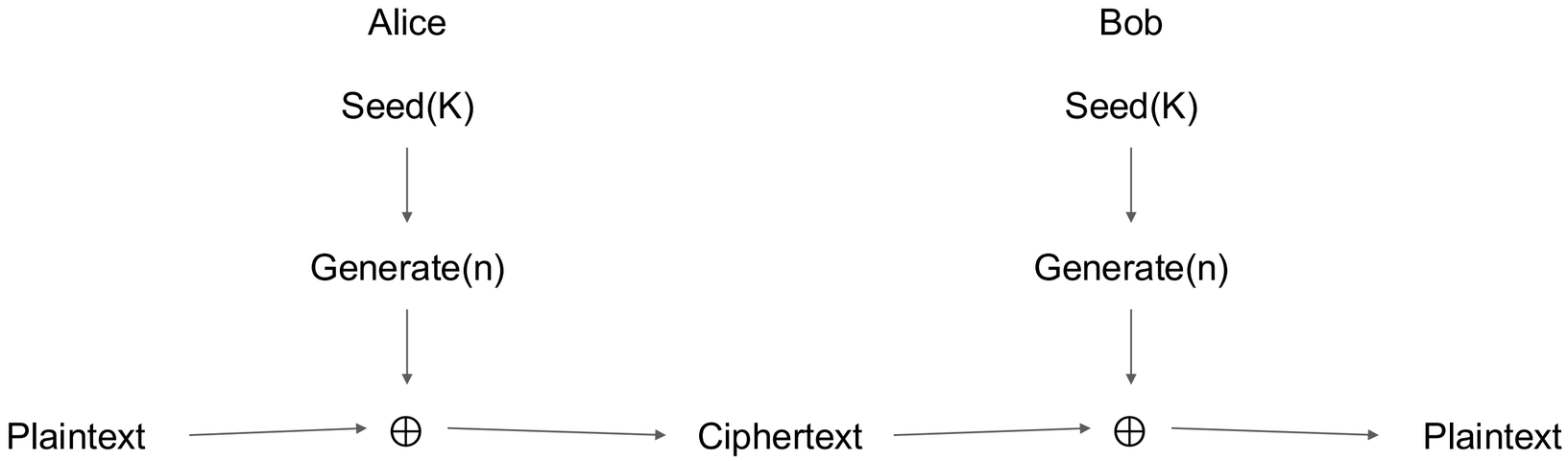
- Protocol: Alice and Bob both seed a secure PRNG with their symmetric secret key, and then use the output as the key for a one-time pad



Stream Ciphers: Encrypting Multiple Messages

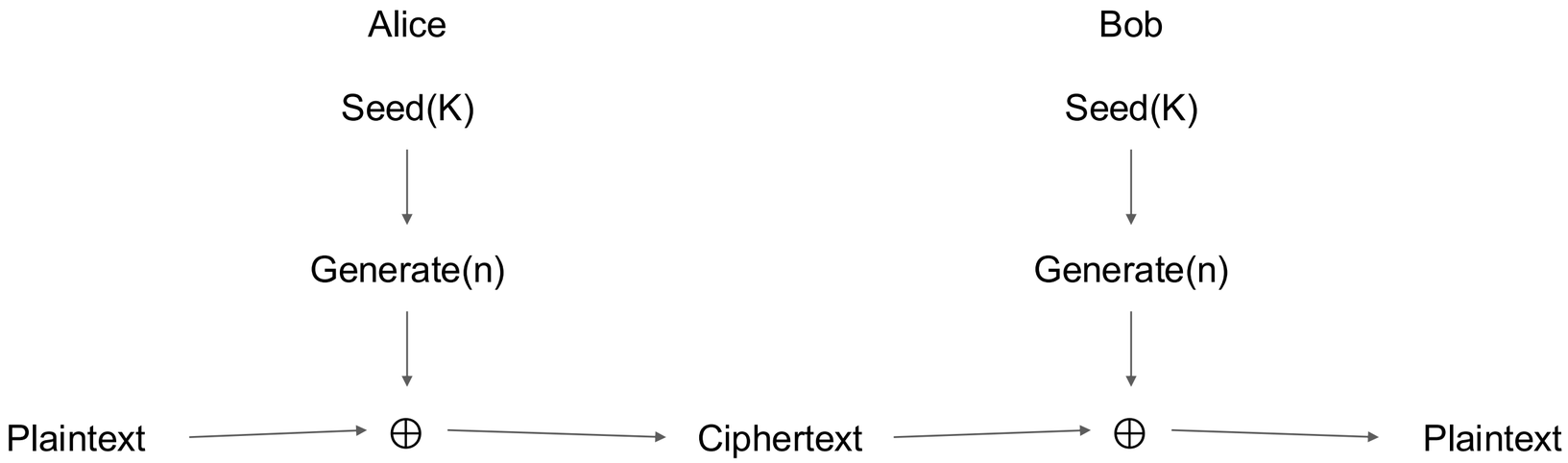


- Recall: One-time pads are insecure when the key is reused. How do we encrypt multiple messages without key reuse?



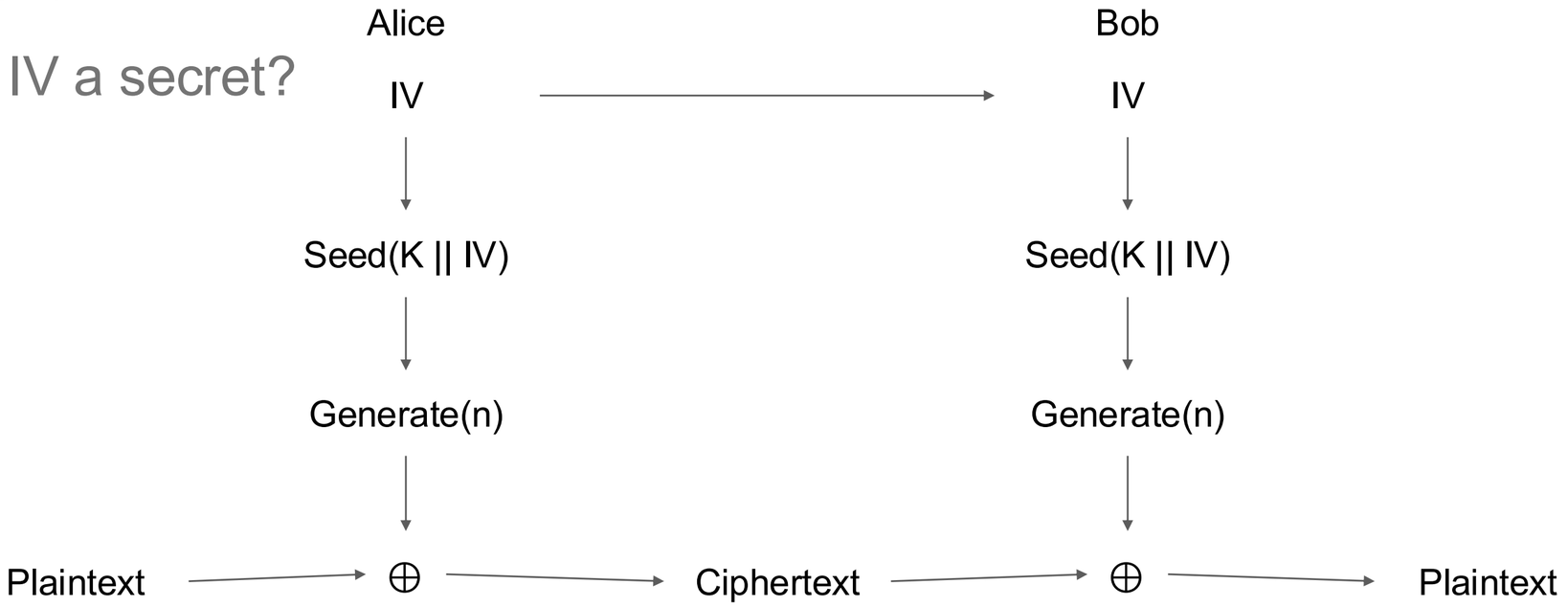
Stream Ciphers: Encrypting Multiple Messages

- Secure?



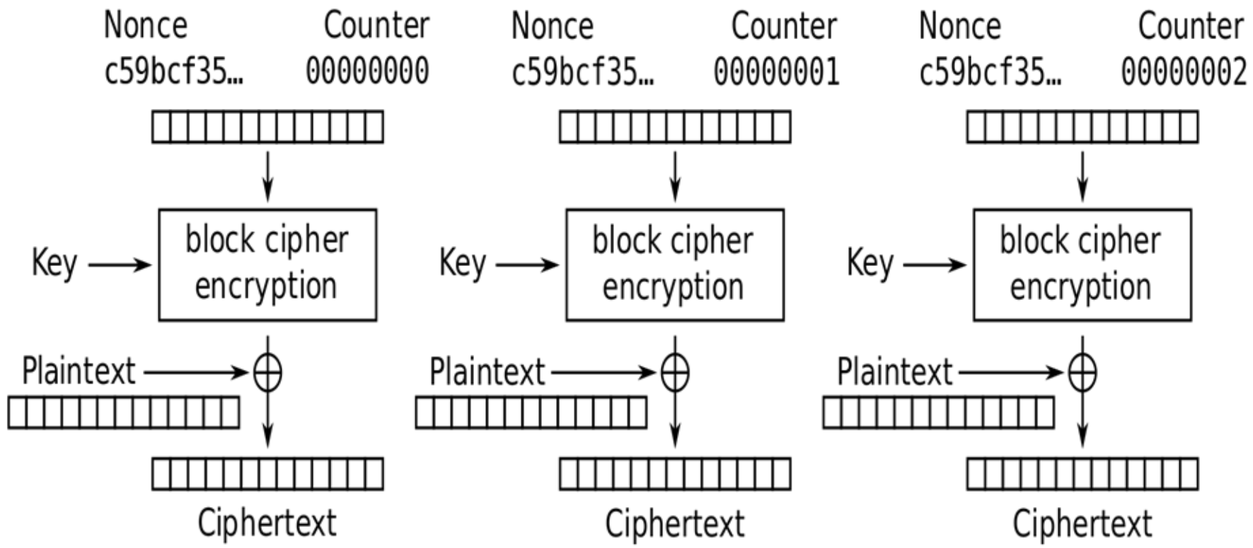
Stream Ciphers: Encrypting Multiple Messages

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



Stream Ciphers: AES-CTR

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



Counter (CTR) mode encryption

Stream Ciphers

- Remember:
  - For OTP, you cannot have the same key for different messages!!

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 Support Streaming



- **Stream ciphers can continually process new data as it arrives**
  - Only need to maintain internal state of the PRNG
  - Keep generating more PRNG output as more input arrives

Stream Ciphers Support Seeking



- 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 *i*th block of ciphertext
  - Example: ChaCha20 (another stream cipher) lets you decrypt arbitrary parts of ciphertext
  - With HMAC-DRBG, you have to generate all the PRNG output up until the block you want to decrypt

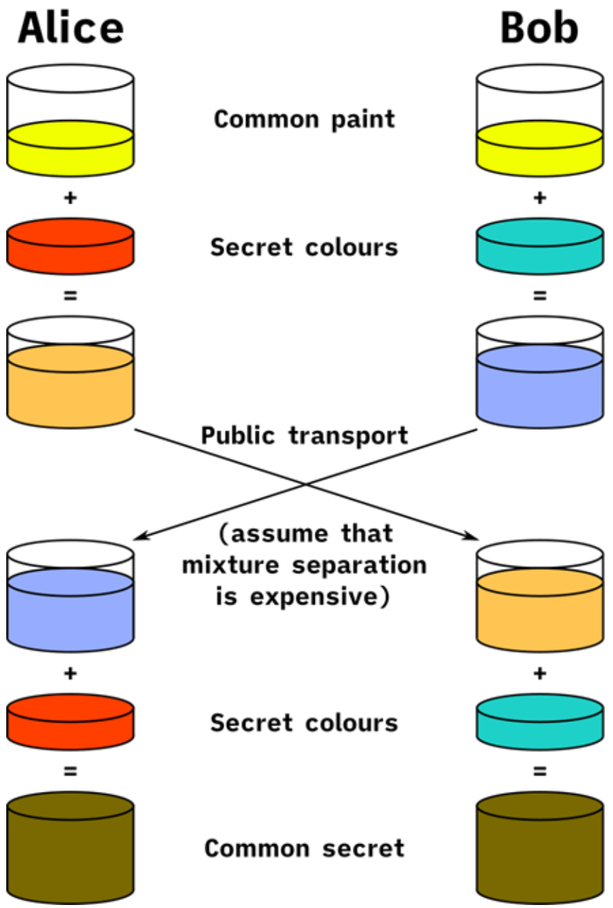
Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC) • Stream ciphers	• 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



Secure Color Sharing



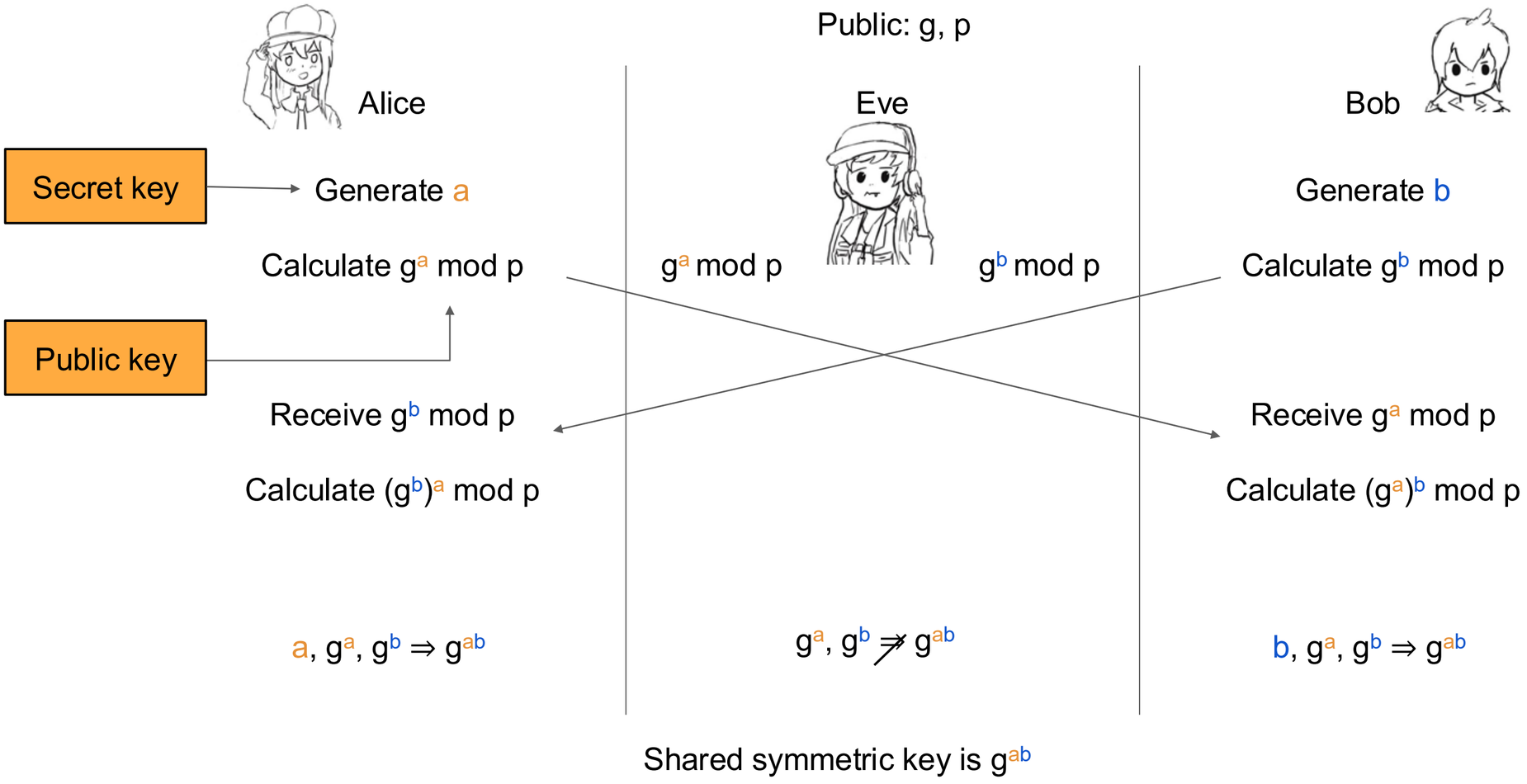
- Suppose Alice and Bob want a secret *paint color*, but Eve can see paint colors sent between Alice and Bob
- **Alice generates a secret color amber A, and Bob generates a secret color blue B**
- **Alice and Bob agree on a common, public color green G**
- They both mix their secret colors with **G**, 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



- **There is a mathematical version of this**
- **Assume everyone knows a large prime *p* (e.g. 2048 bits long) and a number *g***
- **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



Ephemeral Diffie-Hellman



- Diffie-Hellman can be used to generate a symmetric key whenever Alice and Bob want to communicate (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
  - Sometimes K is called a **session key**, because it's only used for a single session
- **Benefit of DHE: Forward secrecy**
  - Suppose Alice and Bob use DHE to agree on a key $K = g^{ab} \bmod p$, then encrypt messages with K, and after they're done, discard a, b, and K
  - If Eve records everything sent over the public channel, and Eve later steals all of Alice and Bob's secrets, Eve still 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



Generate a

Calculate $g^a \bmod p$

Generate m

Calculate $g^m \bmod p$

Generate b

Calculate $g^b \bmod p$

Receive $g^m \bmod p$

Calculate $(g^m)^a \bmod p$

Receive $g^a \bmod p$

Calculate $(g^a)^m \bmod p$

Receive $g^b \bmod p$

Calculate $(g^b)^m \bmod p$

Receive $g^m \bmod p$

Calculate $(g^m)^b \bmod p$

$a, g^a, g^m \Rightarrow g^{am}$

$m, g^m, g^a \Rightarrow g^{am}$

$m, g^m, g^b \Rightarrow g^{bm}$

$b, g^b, g^m \Rightarrow g^{bm}$

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!

Summary: Diffie-Hellman Key Exchange



- **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

In-class Quiz

- https://yajin.org/teaching/IERG4130/2026/q/q5-mac-prng-dh-m9Qk7V2x.html