CMSC414 Computer and Network Security

MACs, PRNGs and Diffie-Hellman Key Exchange

Yizheng Chen | University of Maryland
surrealyz.github.io

Apr 2, 2024

Credits: original slides from instructors and staff from CS161 at UC Berkeley. Blue slides will not be tested.

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 $H(x) = H(x')$.
  - Random/unpredictability, no predictable patterns for how changing the input affects the output
- Some hashes are vulnerable to length extension attacks
- Hashes don't provide integrity (unless you can publish the hash securely)

Length Extension Attacks

- **Length extension attack**: Given *H*(*x*) and the length of *x*, but not *x*, an attacker can create *H*(*x* || *m*) for any *m* of the attacker's choosing
  - Note: This doesn't violate any property of hash functions but is undesirable in some circumstances
- SHA-256 (256-bit version of SHA-2) is vulnerable
- SHA-3 is not vulnerable

Message Authentication Codes (MACs)

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

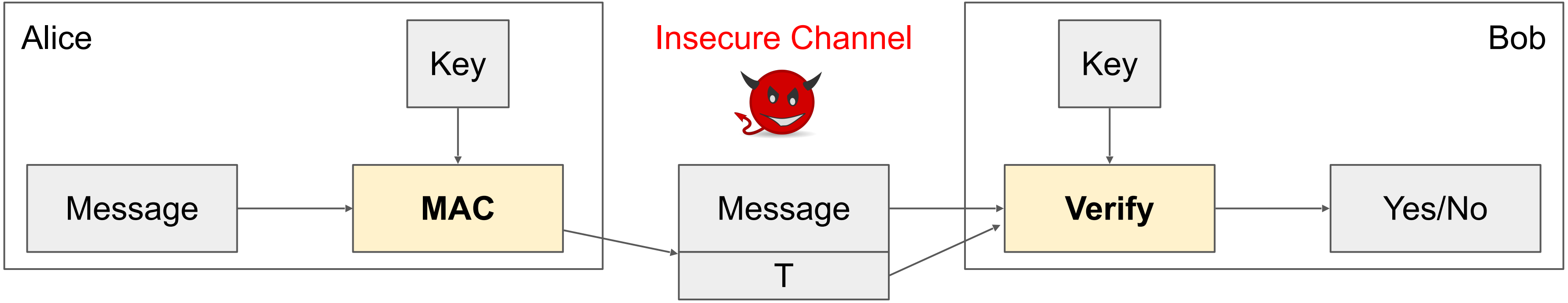
How to Provide Integrity

- Reminder: We’re still in the symmetric-key setting
  - Assume that Alice and Bob share a secret key, and attackers don’t know the key

- We want to attach some piece of information to *convince* Bob that Alice sent this message, even if Mallory is intercepting the message on the network, or to *detect* if Mallory tampered with the message
  - This piece of information can only be generated by someone with the key

MACs: Usage

- Alice wants to send *M* to Bob, but doesn't want Mallory to tamper with it
- Alice sends *M* and *T* = MAC(*K*, *M*) to Bob
- Bob recomputes MAC(*K*, *M*) and checks that it matches *T*
- If the MACs match, Bob is confident the message has not been tampered with (integrity)



MACs: Definition

- Two parts:
  - KeyGen() → *K*: Generate a key *K*
  - MAC(*K*, *M*) → *T*: Generate a tag *T* for the message *M* using key *K*
    - Inputs: A secret key and an arbitrary-length message
    - Output: A fixed-length **tag** on the message

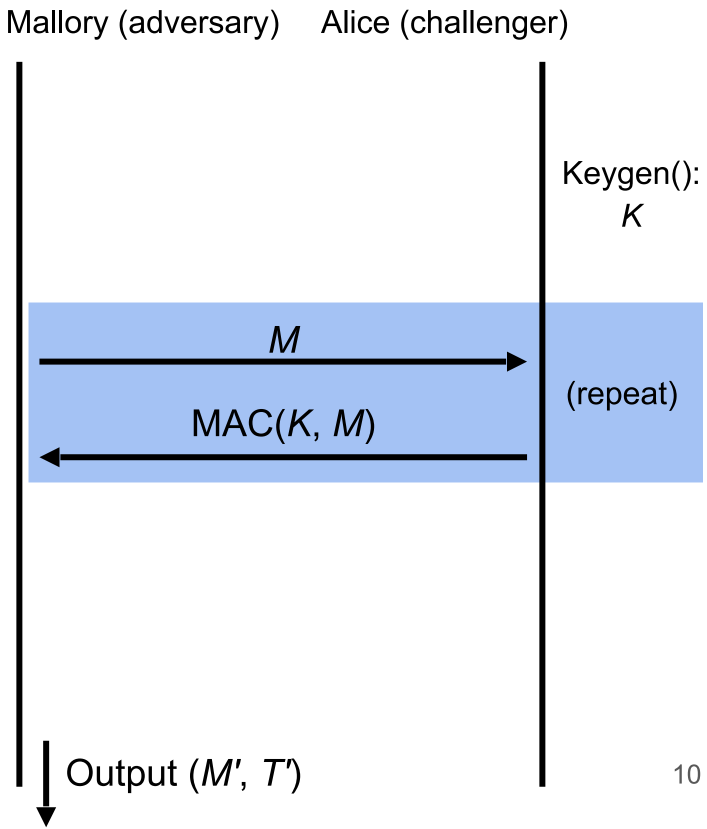
- Properties
  - **Correctness**: Determinism
    - Note: Some more complicated MAC schemes have an additional Verify(*K*, *M*, *T*) function that don't require determinism, but this is out of scope
  - **Efficiency**: Computing a MAC should be efficient
  - **Security**: EU-CPA (existentially unforgeable under chosen plaintext attack)

Defining Integrity: EU-CPA

- A secure MAC is **existentially unforgeable**: without the key, an attacker cannot create a valid tag on a message
  - Mallory cannot generate MAC(*K*, *M'*) without *K*
  - Mallory cannot find any $M' \neq M$ such that MAC(*K*, *M'*) = MAC(*K*, *M*)
- Formally defined by a security game: existential unforgeability under chosen-plaintext attack, or EU-CPA
- MACs should be unforgeable under chosen plaintext attack
  - Intuition: Like IND-CPA, but for integrity and authenticity
  - 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

Defining Integrity: EU-CPA

1. Mallory may send messages to Alice and receive their tags
2. Eventually, Mallory creates a message-tag pair (*M′*, *T′*)
    - *M′* cannot be a message that Mallory requested earlier
    - If *T′* is a valid tag for *M′*, then Mallory wins. Otherwise, she loses.
3. A scheme is EU-CPA secure if for *all* polynomial time adversaries, the probability of winning is 0 or negligible



Example: NMAC

- Can we use secure cryptographic hashes to build a secure MAC?
    - Intuition: Hash output is unpredictable and looks random, so let's hash the key and the message together
- KeyGen():
    - Output two random, n-bit keys K_1 and K_2, where n is the length of the hash output
- NMAC(K_1, K_2, M):
    - Output $H(K_1 \| H(K_2 \| M))$
- NMAC is EU-CPA secure if the two keys are different
    - Provably secure if the underlying hash function is secure
- Intuition: Using two hashes prevents a length extension attack
    - Otherwise, an attacker who sees a tag for M could generate a tag for $M \| M'$

Example: HMAC

- Issues with NMAC:
  - Recall: NMAC(K_1, K_2, M) = $H(K_1 \| H(K_2 \| M))$
  - We need two different keys
  - NMAC requires the keys to be the same length as the hash output (n bits)

- HMAC(K, M):
  - Compute K' as a version of K that is the length of the hash output
    - If K is too short, pad K with 0's to make it n bits (be careful with keys that are too short and lack randomness)
    - If K is too long, hash it so it's n bits
  - Output $H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$

Example: HMAC

- HMAC(K, M):
  - Compute K' as a version of K that is the length of the hash output
    - If K is too short, pad K with 0's to make it n bits (be careful with keys that are too short and lack randomness)
    - If K is too long, hash it so it's n bits
  - Output $H((K' \oplus opad) \| H((K' \oplus ipad) \| M))$

- Use K' to derive two different keys
  - *opad* (outer pad) is the hard-coded byte **`0x5c`** repeated until it's the same length as K'
  - *ipad* (inner pad) is the hard-coded byte **`0x36`** repeated until it's the same length as K'
  - As long as *opad* and *ipad* are different, you'll get two different keys
  - For paranoia, the designers chose two very different bit patterns, even though they theoretically need only differ in one bit

HMAC Properties

- $\text{HMAC}(K, M) = H((K' \oplus opad) \mathbin{||} \text{H}((K' \oplus ipad) \mathbin{||} M))$

- HMAC is a hash function, so it has the properties of the underlying hash too
  - It is collision resistant
  - Given HMAC(*K*, *M*), an attacker can't learn *M*
  - If the underlying hash is secure, HMAC doesn't reveal *M*, but it is still deterministic

- You can't verify a tag *T* if you don't have *K*
  - The attacker can't brute-force the message *M* without knowing *K*

Do MACs provide integrity?

- Do MACs provide integrity?
  - Yes. An attacker cannot tamper with the message without being detected

- Do MACs provide authenticity?
  - It depends on your threat model
  - If a message has a valid MAC, you can be sure it came from *someone with the secret key*, but you can't narrow it down to one person
  - If only two people have the secret key, MACs provide authenticity: it has a valid MAC, and it's not from me, so it must be from the other person

- Do MACs provide confidentiality?
  - MACs are deterministic ⇒ No IND-CPA security
  - MACs in general have no confidentiality guarantees; they can leak information about the message

MACs: Summary

- 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

Authenticated Encryption

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

Authenticated Encryption: Definition

- **Authenticated encryption** (**AE**): A scheme that simultaneously guarantees confidentiality and integrity (and authenticity, depending on your threat model) on a message

- Two ways of achieving authenticated encryption:
  - Combine schemes that provide confidentiality with schemes that provide integrity
  - Use a scheme that is designed to provide confidentiality and integrity

Combining Schemes: Let's design it together

- You can use:
  - An IND-CPA encryption scheme (e.g. AES-CBC): Enc(*K*, *M*) and Dec(*K*, *M*)
  - An unforgeable MAC scheme (e.g. HMAC): MAC(*K*, *M*)
- First attempt: Alice sends Enc(K_1, *M*) and MAC(K_2, *M*)
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? No, the MAC is not IND-CPA secure
- Idea: Let's compute the MAC on the *ciphertext* instead of the plaintext:
  Enc(K_1, *M*) and MAC(k_2, Enc(K_1, *M*))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, the MAC might leak info about the ciphertext, but that's okay
- Idea: Let's encrypt the MAC too: Enc(K_1, *M* || MAC(K_2, *M*))
  - Integrity? Yes, attacker can't tamper with the MAC
  - Confidentiality? Yes, everything is encrypted

Encrypt-then-MAC or MAC-then-Encrypt?

- Encrypt-then-MAC
  - First compute $\text{Enc}(K_1, M)$
  - Then MAC the ciphertext: $\text{MAC}(K_2, \text{Enc}(K_1, M))$
- MAC-then-encrypt
  - First compute $\text{MAC}(K_2, M)$
  - Then encrypt the message and the MAC together: $\text{Enc}(K_1, M \| \text{MAC}(K_2, M))$
- Which is better?
  - In theory, both are IND-CPA and EU-CPA secure if applied properly
  - MAC-then-encrypt has a downside: You don't know if tampering has occurred until after decrypting
    - Attacker can supply arbitrary tampered input, and you always have to decrypt it
    - Passing attacker-chosen input through the decryption function can cause side-channel leaks
- **Always use encrypt-then-MAC** because it's more robust to mistakes

Key Reuse

- **Key reuse problem**: Using the same key in two different use cases
  - Note: Using the same key multiple times for the same use (e.g. computing HMACs on different messages in the same context with the same key) is not key reuse problem

- Reusing keys can cause the underlying algorithms to interfere with each other and affect security guarantees
  - Example: If you use a block-cipher-based MAC algorithm and a block cipher chaining mode, the underlying block ciphers may no longer be secure
  - Thinking about these attacks is hard

Key Reuse

- Simplest solution: Do not reuse keys across schemes! One key per *scheme instance*.
  - Encrypt a piece of data and MAC a piece of data?
    - Different use; different key
  - MAC one of Alice’s messages to Bob and MAC one of Bob’s messages to Alice?
    - Different use; different key

TLS 1.0 “Lucky 13” Attack

- TLS: A protocol for sending encrypted and authenticated messages over the Internet (we’ll study it more in the networking unit)
- TLS 1.0 uses MAC-then-encrypt: $\text{Enc}(K_1, M \,||\, \text{MAC}(K_2, M))$
  - The encryption algorithm is AES-CBC
- The Lucky 13 attack abuses MAC-then-encrypt to read encrypted messages
  - Guess a byte of plaintext and change the ciphertext accordingly
  - The MAC will error, but the time it takes to error is different depending on if the guess is correct
  - Attacker measures how long it takes to error in order to learn information about plaintext
  - TLS will send the message again if the MAC errors, so the attacker can guess repeatedly
- Takeaways
  - Side channel attack: The algorithm is proved secure, but poor implementation made it vulnerable
  - Always encrypt-then-MAC

AEAD Encryption

- Second method for authenticated encryption: Use a scheme that is designed to provide confidentiality, integrity, and authenticity
- **Authenticated encryption with additional data** (**AEAD**): An algorithm that provides both confidentiality and integrity over the plaintext and integrity over *additional data*
  - Additional data is usually context (e.g. memory address), so you can't change the context without breaking the MAC
- Great if used correctly: No more worrying about MAC-then-encrypt
  - If you use AEAD incorrectly, you lose *both* confidentiality and integrity/authentication
  - Example of correct usage: Using a crypto library with AEAD

Authenticated Encryption: Summary

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

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
  - …
- 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
  - Toss a fair coin?
  - Find an unpredictable circuit on a CPU?
  - Measure the microsecond you pressed a key?

- **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)

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

- A PRNG has two functions:
  - PRNG.Seed(randomness): Initializes the internal state using the entropy
    - Input: Some truly random bits
  - PRNG.Generate(*m*): Generate *m* pseudorandom bits
    - Input: A number *m*
    - Output: *m* pseudorandom bits
    - Updates the internal state as needed

Properties

- **Correctness**: Deterministic
- **Efficiency**: Efficient to generate pseudorandom bits
- **Security**: Indistinguishability from random

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

- 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 > ½+negl

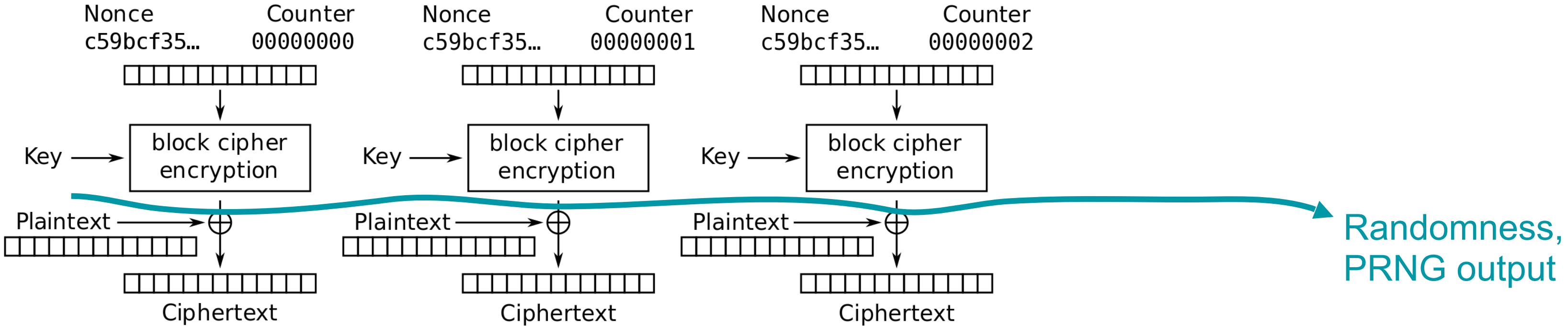
- Equivalence: An attacker cannot predict future output of the PRNG

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

Example construction of PRNG

- Using block cipher in CTR mode:
- If you want m random bits, and a block cipher with E_k has n bits, apply the block cipher m/n times and concatenate the result:
- PRNG.Seed(K | IV);
- Generate(m) = E_k(IV|1) | E_k(IV| 2) | E_k(IV|3) … E_k(IV| ceil(m/n)),
    - | is concatenation



Counter (CTR) mode encryption

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
    - Generate(n): Generate n bits of pseudorandom output
- Security: computationally indistinguishable from truly random bits
- Example using AES in CTR mode
- Application: UUIDs

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

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

Diffie-Hellman Key Exchange

Public: g, p

Alice 	Eve 	Bob 
Generate a		Generate b
Calculate $g^a \bmod p$	g^a 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}

Discrete Log Problem and Diffie-Hellman Problem

- 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 or b, …
  - Note: Multiplying the values doesn't work, since you get g^{a+b} mod $p \neq g^{ab}$ mod p

Discrete Log Problem and Diffie-Hellman Problem

For a random a, b, R:

$g, p, \quad g^a \bmod p, \quad g^b \bmod p, \quad g^{ab} \bmod p$

$\sim$ ← Indistinguishable from the perspective of a polynomial time attacker

$g, p, \quad g^a \bmod p, \quad g^b \bmod p, \quad R$