Diffie-Hellman Key Exchange (continued) and Public-Key Encryption

CS 161 Spring 2022 - Lecture 10

Diffie-Hellman Key Exchange

Textbook Chapter 10

Discrete Log Problem and Diffie-Hellman Problem

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

Diffie-Hellman Key Exchange

Public: g, p

Alice	Eve	Bob
Generate a		Generate b
Calculate g^a mod p	g^a $\quad$ g^b	Calculate g^b mod p
Receive g^b mod p		Receive g^a mod p
Calculate $(g^b)^a$ mod p		Calculate $(g^a)^b$ mod p
$a, g^a, g^b \Rightarrow g^{ab}$	$g^a, g^b \nRightarrow g^{ab}$	$b, g^a, g^b \Rightarrow g^{ab}$

$$\boxed{\text{Secret key}} \rightarrow \text{Generate } a$$

$$\boxed{\text{Public key}} \rightarrow \text{Calculate } g^a \text{ mod } p$$

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 and MITM Attacks: What about Mallory?

Public: g, p

Alice	Mallory	Bob
Generate a		Generate b
Calculate $g^a \bmod p$	g^a $\qquad$ g^b	Calculate $g^b \bmod p$
Receive $g^b \bmod p$		Receive $g^a \bmod p$
Calculate $(g^b)^a \bmod p$		Calculate $(g^a)^b \bmod p$
$a, g^a, g^b \Rightarrow g^{ab}$	$g^a, g^b \nRightarrow g^{ab}$	$b, g^a, g^b \Rightarrow g^{ab}$

Diffie-Hellman and MITM Attacks

- What about Mallory?
  - Remember: Mallory can alter messages, block messages, and send her own messages
  - *Not* secure against a MITM attacker: Mallory can just do a DH with both sides!

Diffie-Hellman: Security

Alice

Generate a

Calculate g^a mod p

Receive g^m mod p

Calculate $(g^m)^a$ mod p

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

Mallory

Public: g, p

Generate m

Calculate g^m mod p

Receive g^a mod p

Calculate $(g^a)^m$ mod p

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

Receive g^b mod p

Calculate $(g^b)^m$ mod p

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

Bob

Generate b

Calculate g^b mod p

Receive g^m mod p

Calculate $(g^m)^b$ mod p

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)

Measuring Cryptographic Problem Difficulty

- We have a good understanding of how hard it is to break our symmetric-key primitives
  - Breaking symmetric-key encryption with 128-bit keys: 2^{128} difficulty (brute force)
    - We use this as the baseline for "difficulty level": What is the difficulty of brute-forcing the symmetric key?
  - Breaking a 256-bit hash: 2^{128} difficulty (birthday paradox)
- We *believe* the problems underlying public-key cryptography are hard
  - Breaking 256-bit elliptic curves: 2^{128} difficulty?
  - Breaking 2048-bit Diffie-Hellman and RSA (we'll see this later): 2^{128} difficulty?
- For our purposes, think of elliptic curve as "add the letters EC, and use fewer bits and different magic math"

Summary: Diffie-Hellman Key Exchange

- 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

Nothing-Up-My-Sleeve-Numbers

Nothing-Up-My-Sleeve-Numbers

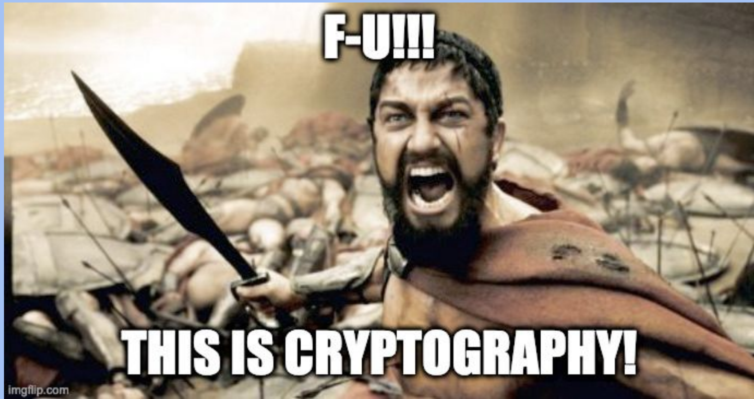
- Cryptography uses a lot of constants
  - Initial state for SHA
  - Prime *p* and generator *g* in Diffie-Hellman
  - Parameters for elliptic-curve cryptography (curve equations, points on the curve like *P* and *Q*)
  - *ipad* and *opad* in HMAC
- Usually, any value could work, but the designer needed to choose *some* constant for the algorithm
  - Example: In Diffie-Hellman, any large prime *p* and generator *g* works, but there's a default *p* and *g* that everyone uses
- Where do these default parameter values come from?

PRNG Sabotage: Dual_EC_DRBG

- Dual_EC_DRBG: A PRNG published by the NSA behind the scenes
- Relies on two public hard-coded parameters *P* and *Q*
    - *P* and *Q* are points on an elliptic curve
    - If *P* and *Q* are related by $Q = eP$ (analogous to $Q = P^e \bmod n$ in discrete log), you can learn the internal state!
- It also sucked!
    - It was horribly slow (much slower than HMAC or CTR PRNGs)
    - It had subtle biases that shouldn't exist in a secure PRNG: You could distinguish the upper bits from random
    - Cryptographers spotted these flaws early on
    - Why would anyone use such a horrible PRNG?

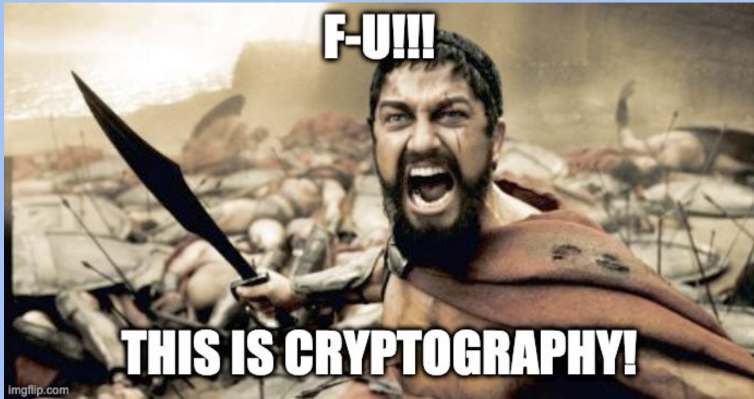
PRNG Sabotage: Dual_EC_DRBG

- Why would anyone use such a horrible PRNG?
- Story time:
  - RSA Data Security accepts $10 million from the NSA
  - In exchange, RSA Data Security implements Dual_EC in their RSA BSAFE library, and silently makes it the default PRNG
  - With RSA Data Security's support, Dual_EC became a NIST standard
    - Used in other products
  - 2013: Whistleblower Edward Snowden reveals classified NSA information
    - The New York Times vaguely mentions a crypto talk given by Microsoft people
    - Everybody quickly realized this referred to a backdoor the NSA inserted into Dual_EC
    - Backdoor: An attack inserted by an organization (e.g. NSA) so that they, but nobody else, can attack the system



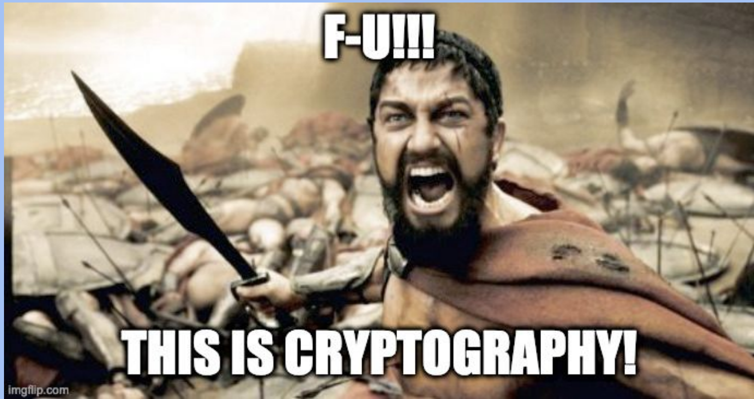
PRNG Sabotage: Dual_EC_DRBG

- With the backdoor, it's possible to predict both future and past PRNG outputs
  - No rollback resistance, unlike HMAC-DRBG
- Recall the attack when a PRNG is not rollback resistant:
  - Generate a secret key
  - Generate some other publicly visible random value (e.g. IVs, nonces, an NSA-sponsored "standard" that requires some random output)
  - Since the PRNG is not rollback-resistant, the NSA can view the public random value and use the backdoor to learn secret keys



PRNG Sabotage: Dual_EC_DRBG

- Juniper Networks used Dual_EC in their virtual private networks (VPNs)
- Juniper claims their version of Dual_EC was safe from the NSA backdoor
    - Juniper selected a different public parameter (Q) than the NSA's public parameter
- Later: Juniper is hacked
    - The hacker changed Dual_EC's public parameter (Q) to give themselves a backdoor!
        - And now the original backdoor owner can't get in…
- Later: Juniper is hacked again
    - The hacker adds an SSH backdoor
- Later: Juniper notices the SSH backdoor
    - … and also notices the changed Dual_EC backdoor
    - So they release an update to patch the Dual_EC backdoor
- Later: Everyone receives an update
    - Everyone: "Ohh, patch for a backdoor. Let's see what got fixed. Oh, these look like Dual_EC parameters…"



Diffie-Hellman Sabotage?

- Another case of potential sabotage
- 1024b Diffie-Hellman is moderately impractical
- However, for a specially chosen *p*, cracking Diffie-Hellman becomes 1 million times easier
    - Recall: *p* is a public value. Most implementations use default hard-coded values of *p*
    - The most commonly-used “example” *p* has unknown origin! (lost to time)

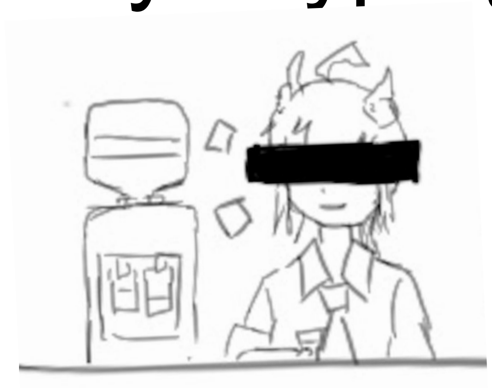
Something Up Their Sleeves… (Maybe)

- Mysterious public parameters can cast doubt *even when a design is solid*
- Recall DES (the block cipher used before AES)
  - The DES standard was developed by IBM but with input from the NSA
  - Everyone was suspicious that the NSA tampered with S-boxes (hard-coded constant values in the block cipher algorithm)
  - It turns out they did: The NSA made them stronger against an attack that they knew about but the public didn't know about
- P-256 and P-384: Two elliptic curves defined by the NSA
  - Remember: Elliptic curves are public parameters in elliptic-curve cryptography
  - The elliptic curves were chosen mysteriously and cast doubt on whether there is a backdoor

Takeaway: Nothing-Up-My-Sleeve-Numbers

- Good systems should clearly and transparently describe how public parameters are generated
  - Argue about why these values provide good security (e.g. AES designers argued this for the values in their S-boxes)
  - Choose values with obvious human significance: The developer probably doesn't have millions of values of obvious significance to brute-force and pick a value with a backdoor
    - Seed a PRNG with your name
    - The first few digits of π
    - 0x67452301, 0xefcdab89, … (SHA-1)
    - Fractional parts of the square/cube roots of prime numbers (SHA-2)

Public-Key Cryptography



Textbook Chapters 11 & 12

Public-Key Cryptography

- In public-key schemes, each person has two keys
  - **Public key**: Known to everybody
  - **Private key**: Only known by that person
  - Keys come in pairs: every public key corresponds to one private key
- Uses number theory
  - Examples: Modular arithmetic, factoring, discrete logarithm problem
  - Contrast with symmetric-key cryptography (uses XORs and bit-shifts)
- Messages are numbers
  - Contrast with symmetric-key cryptography (messages are bit strings)
- Benefit: No longer need to assume that Alice and Bob already share a secret
- Drawback: Much slower than symmetric-key cryptography
  - Number theory calculations are much slower than XORs and bit-shifts

Public-Key Encryption

Textbook Chapter 11

Public-Key Encryption

- Everybody can encrypt with the public key
- Only the recipient can decrypt with the private key

Public-Key Encryption: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the public key, and *SK* is the private (secret) key
  - Enc(*PK*, *M*) → *C*: Encrypt a plaintext *M* using public key *PK* to produce ciphertext *C*
  - Dec(*SK*, *C*) → *M*: Decrypt a ciphertext *C* using secret key *SK*
- Properties
  - **Correctness**: Decrypting a ciphertext should result in the message that was originally encrypted
    - Dec(*SK*, Enc(*PK*, *M*)) = *M* for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Encryption/decryption should be fast
  - **Security**: Similar to IND-CPA, but Alice (the challenger) just gives Eve (the adversary) the public key, and Eve doesn't request encryptions, except for the pair M_0, M_1
    - You don't need to worry about this game (it's called "semantic security")

ElGamal Encryption

Textbook Chapter 11.4

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

ElGamal Encryption

- Diffie-Hellman key exchange is great: It lets Alice and Bob share a secret over an insecure channel
- Problem: Diffie-Hellman by itself can’t send messages. The secret $g^{ab} \bmod p$ is random.
- Idea: Let’s modify Diffie-Hellman so it supports encrypting and decrypting messages directly

ElGamal Encryption: Protocol

- KeyGen():
  - Bob generates private key b and public key $B = g^b \bmod p$
    - Intuition: Bob is completing his half of the Diffie-Hellman exchange
- Enc(B, M):
  - Alice generates a random r and computes $R = g^r \bmod p$
    - Intuition: Alice is completing her half of the Diffie-Hellman exchange
  - Alice computes $M \times B^r \bmod p$
    - Intuition: Alice derives the shared secret and multiples her message by the secret
  - Alice sends $C_1 = R$, $C_2 = M \times B^r \bmod p$
- Dec(b, C_1, C_2)
  - Bob computes $C_2 \times C_1^{-b} = M \times B^r \times R^{-b} = M \times g^{br} \times g^{-br} = M \bmod p$
    - Intuition: Bob derives the (inverse) shared secret and multiples the ciphertext by the inverse shared secret

ElGamal Encryption: Security

- Recall Diffie-Hellman problem: Given g^a mod p and g^b mod p, hard to recover g^{ab} mod p
- ElGamal sends these values over the insecure channel
  - Bob's public key: B
  - Ciphertext: R, $M \times B^r$ mod p
- Eve can't derive g^{br}, so she can't recover M

ElGamal Encryption: Issues

- Is ElGamal encryption IND-CPA secure?
  - No. The adversary can send $M_0 = 0$, $M_1 \neq 0$
  - Additional padding and other modifications are needed to make it semantically secure
- Malleability: The adversary can tamper with the message
  - The adversary can manipulate $C_1' = C_1$, $C_2' = 2 \times C_2 = 2 \times M \times g^{br}$ to make it look like $2 \times M$ was encrypted

RSA Encryption

Textbook Chapter 11.3

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

RSA Encryption: Definition

- KeyGen():
  - Randomly pick two large primes, p and q
    - Done by picking random numbers and then using a test to see if the number is (probably) prime
  - Compute $N = pq$
    - n is usually between 2048 bits and 4096 bits long
  - Choose e
    - Requirement: e is relatively prime to $(p - 1)(q - 1)$
    - Requirement: $2 < e < (p - 1)(q - 1)$
  - Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
    - Algorithm: Extended Euclid's algorithm (CS 70, but out of scope)
  - **Public key**: N and e
  - **Private key:** d

RSA Encryption: Definition

- Enc(e, N, M):
  - Output $M^e \bmod N$
- Dec(d, C):
  - Output $C^d = (M^e)^d \bmod N$

RSA Encryption: Correctness

1. Theorem: $M^{ed} \equiv M \bmod N$
2. Euler's theorem: $a^{\varphi(N)} \equiv 1 \bmod N$
   - $\varphi(N)$ is the totient function of N
   - If N is prime, $\varphi(N) = N - 1$ (Fermat's little theorem)
   - For a semi-prime pq, where p and q are prime, $\varphi(pq) = (p - 1)(q - 1)$
   - This is all out-of-scope CS 70 knowledge
3. Notice: $ed \equiv 1 \bmod (p - 1)(q - 1)$ so $ed \equiv 1 \bmod \varphi(N)$
   - This means that $ed = k\varphi(n) + 1$ for some integer k
4. (1) can be written as $M^{k\varphi(N) + 1} \equiv M \bmod N$
5. $M^{k\varphi(N)}M^1 \equiv M \bmod N$
6. $1M^1 \equiv M \bmod N$ by Euler's theorem
7. $M \equiv M \bmod N$

RSA Encryption: Security

- **RSA problem**: Given n and $C = M^e \bmod N$, it is hard to find M
  - No harder than the factoring problem (if you can factor N, you can recover d)
- Current best solution is to factor N, but unknown whether there is an easier way
  - If the RSA problem is as hard as the factoring problem, then the scheme is secure as long as the factoring problem is hard
  - Factoring problem is assumed to be hard (if you don't have a massive quantum computer, that is)

RSA Encryption: Issues

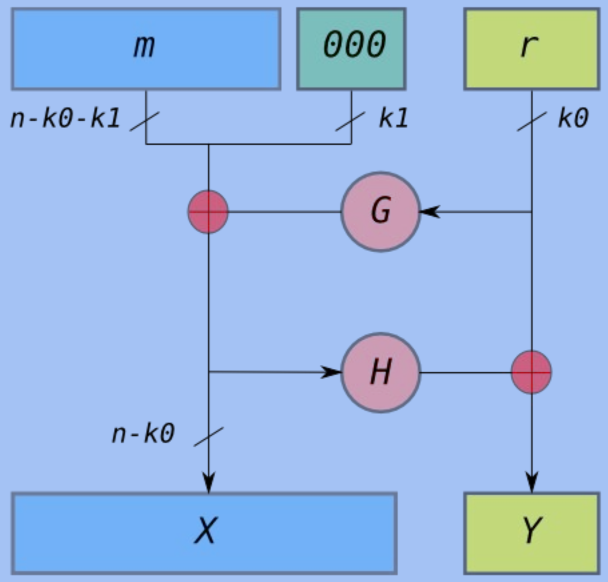
- Is RSA encryption IND-CPA secure?
  - No. It's deterministic. No randomness was used at any point!
- Sending the same message encrypted with different public keys also leaks information
  - m^{e_a} mod N_a, m^{e_b} mod n_b, m^{e_c} mod N_c
  - Small *m* and *e* leaks information
    - *e* is usually small (~16 bits) and often constant (3, 17, 65537)
- Side channel: A poor implementation leaks information
  - The time it takes to decrypt a message depends on the message and the private key
  - This attack has been successfully used to break RSA encryption in OpenSSL
- Result: We need a probabilistic padding scheme

OAEP

- **Optimal asymmetric encryption padding** (**OAEP**): A variation of RSA that introduces randomness
  - Different from “padding” used for symmetric encryption, used to add randomness instead of dummy bytes
- Idea: RSA can only encrypt “random-looking” numbers, so encrypt the message with a random key

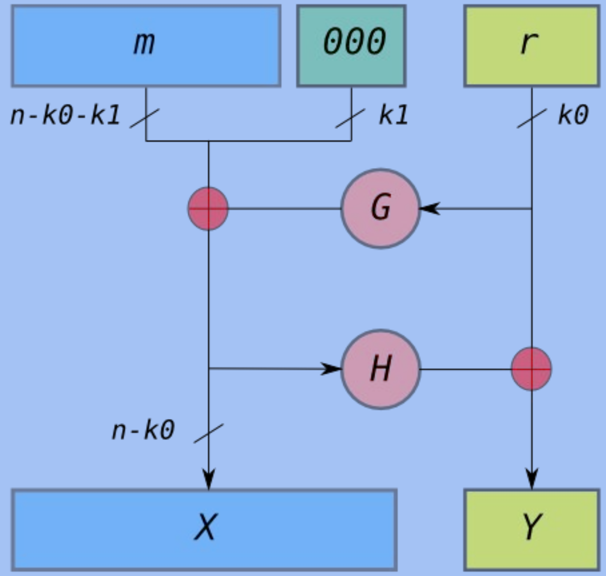
OAEP: Padding

1. k_0 and k_1 constants defined in the standard, and G and H are hash functions
   - M can only be $n - k_0 - k_1$ bits long
   - G produces a ($n - k_0$)-bit hash, and H produces a k_0-bit hash
2. Pad M with k_0 0's
   - Idea: We should see 0's here when unpadding, or else someone tampered with the message
3. Generate a random, k_1-bit string r
4. Compute $X = M \,||\, 00...0 \oplus G(r)$
5. Compute $Y = r \oplus H(X)$
6. Result: $X \,||\, Y$



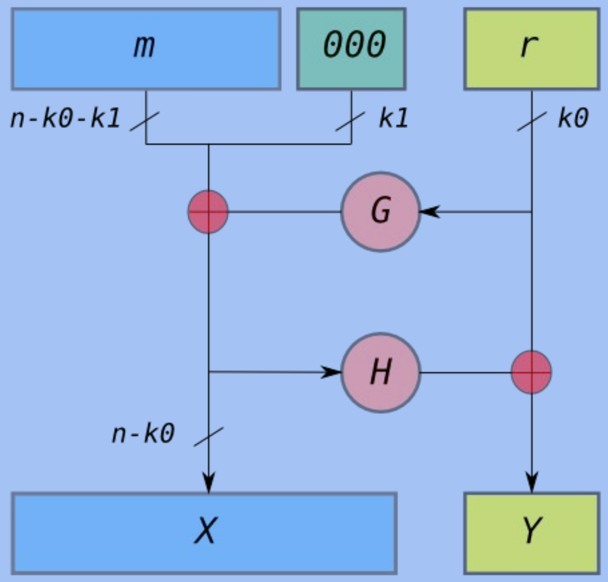
OAEP: Unpadding

1. Compute $r = Y \oplus H(X)$
2. Compute $M \,||\, 00...0 = X \oplus G(r)$
3. Verify that $M \,||\, 00...0$ actually ends in k_1 0's
    - Error if not



OAEP

- Even though *G* and *H* are irreversible, we can recover their inputs using XOR and work backwards
- This structure is called a **Feistel network**
  - Can be used for encryption algorithms if *G* and *H* depend on a key
    - Example: DES (out of scope)
- **Takeaway**: To fix the problems with RSA (it's only secure encrypting random numbers and isn't IND-CPA), use RSA with OAEP, abbreviated as RSA-OAEP



Hybrid Encryption

- Issues with public-key encryption
  - Notice: We can only encrypt small messages because of the modulo operator
  - Notice: There is a lot of math, and computers are slow at math
  - Result: Asymmetric doesn't work for large messages
- **Hybrid encryption**: Encrypt data under a randomly generated key *K* using symmetric encryption, and encrypt *K* using asymmetric encryption
  - Benefit: Now we can encrypt large amounts of data quickly using symmetric encryption, and we still have the security of asymmetric encryption
- Almost all cryptographic systems use hybrid encryption

Digital Signatures

Textbook Chapter 12

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

Digital Signatures

- Asymmetric cryptography is good because we don’t need to share a secret key
- Digital signatures are the asymmetric way of providing integrity/authenticity to data
- Assume that Alice and Bob can communicate public keys without Mallory interfering
    - We will see how to fix this limitation later

Public-key Signatures

- Only the owner of the private key can sign messages with the private key
- Everybody can verify the signature with the public key

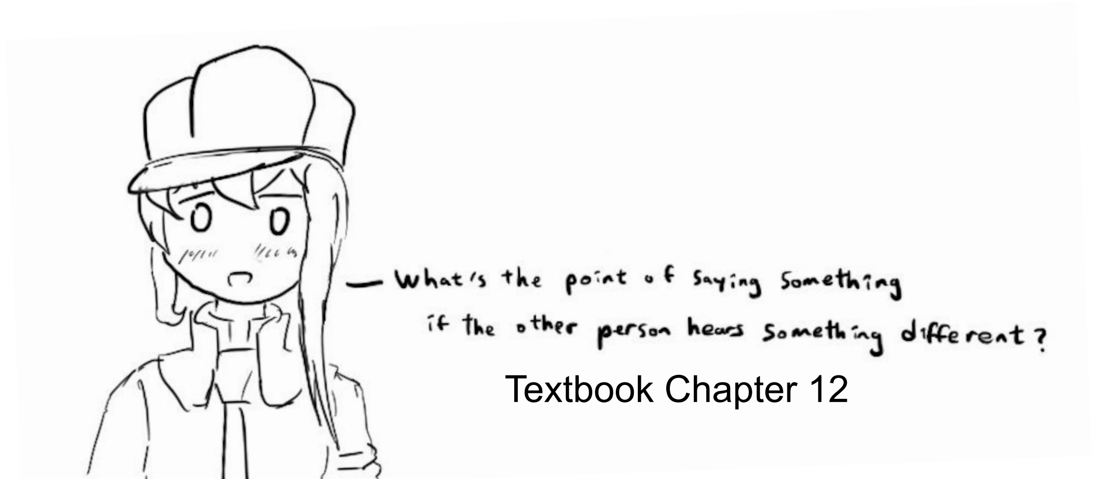
Digital Signatures: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the verify (public) key, and *SK* is the signing (secret) key
  - Sign(*SK*, *M*) → *sig*: Sign the message *M* using the signing key *SK* to produce the signature *sig*
  - Verify(*PK*, *M*, *sig*) → {0, 1}: Verify the signature *sig* on message *M* using the verify key *PK* and output 1 if valid and 0 if invalid
- Properties
  - **Correctness**: Verification should be successful for a signature generated over any message
    - Verify(*PK*, *M*, Sign(*SK*, *M*)) = 1 for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Signing/verifying should be fast
  - **Security**: EU-CPA, same as for MACs

Digital Signatures in Practice

- If you want to sign message *M*:
  - First hash *M*
  - Then sign H(*M*)
- Why do digital signatures use a hash?
  - Allows signing arbitrarily long messages
- Digital signatures provide integrity *and authenticity* for *M*
  - The digital signature acts as proof that the private key holder signed H(*M*), so you know that *M* is authentically endorsed by the private key holder

RSA Signatures


— What's the point of saying something if the other person hears something different?

Textbook Chapter 12

RSA Signatures

- Recall RSA encryption: $M^{ed} \equiv M \bmod N$
  - There is nothing special about using *e* first or using *d* first!
  - If we encrypt using *d*, then anyone can "decrypt" using *e*
    - Given x and $x^d \bmod N$, can't recover *d* because of discrete-log problem, so *d* is safe

RSA Signatures: Definition

- KeyGen():
  - Same as RSA encryption:
    - **Public key**: N and e
    - **Private key:** d
- Sign(d, M):
  - Compute $H(M)^d \bmod N$
- Verify(e, N, M, sig)
  - Verify that $H(M) \equiv sig^e \bmod N$

RSA Signatures: Definition

- Recall RSA encryption: $M^{ed} \equiv M \bmod N$
    - There is nothing special about using *e* first or using *d* first!
    - If we encrypt using *d*, then anyone can “decrypt” using *e*
        - Given *x* and x^d mod *N*, can’t recover *d* because of discrete-log problem, so *d* is safe

DSA Signatures

DSA Signatures

- A signature scheme based on Diffie-Hellman
  - The details of the algorithm are out of scope
- Usage
  - Alice generates a public-private key pair and publishes her public key
  - To sign a message, Alice generates a random, secret value *k* and does some computation
  - Note: *k* is not Alice's private key
  - Note: *k* is sometimes called a nonce but it is not: it must be *random* and never reused
  - The signature itself does not include *k*!
- *k* must be random and secret for each message
  - An attacker who learns *k* can also learn Alice's private key
  - If Alice reuses *k* on two signatures, an attacker can learn *k* (and use *k* to learn her private key)

DSA Signatures: Attacks

- Sony PlayStation 3 (PS3)
- Digital rights management (DRM)
  - Prevent unauthorized code (e.g. pirated software) from running
  - The PS3 was designed to only run signed code
  - Signature algorithm: Elliptic-curve DSA
- Running alternate operating systems
  - The PS3 had an option to run alternate operating systems (Linux) that was later removed
  - This was catnip to reverse engineers (“The best way to get people interested is *removing* Linux from a device”)
- One of the authentication keys used to sign the firmware reused *k* for multiple signatures → security lost!

DSA Signatures: Attacks

- Android OS vulnerability (2013)
  - The "SecureRandom" function in its random number generator (RNG) wasn’t actually secure!
  - Not only was it low entropy, it would sometimes return the same value multiple times
- Multiple Bitcoin wallet apps on Android were affected
  - Bitcoin payments are signed with elliptic-curve DSA and published publicly
  - Insecure RNG caused multiple payments to be signed with the same *k*
- Attack: Someone scanned for all Bitcoin transactions signed insecurely
  - Recall: When multiple signatures use the same *k*, the attacker can learn *k* and the private key
  - In Bitcoin, your private key unlocks access to all your money

DSA Signatures: Attacks

- Chromebooks have a built-in U2F (universal second factor) security key
  - Uses signatures to let the user log in to particular websites
  - Signature algorithm: 256-bit elliptic-curve DSA
- There was a bug in the secure hardware!
  - Instead of using 256-bit *k*, a bug caused *k* to be 32 bits long!
  - An attacker with a signature could simply try all possible values of *k*
- Fortunately the damage was slight
  - Each signature is only valid for logging into a single website
  - Each website used its own private key
- **Takeaway**: DSA (or ECDSA) is particularly vulnerable to incorrect implementations, compared with RSA signatures

Nick’s preferred algorithms

Nick’s preferred algorithms

- CNSA: The NSA’s latest cryptographic suite (2018)
  - Successor to Suite B (2005-2018)
  - Algorithms used for top-secret classified information
  - NSA requires that top-secret ciphertext captured today should not be decryptable by an adversary 40 years from now!
- CNSA algorithms
  - Symmetric-key encryption: AES with 256-bit keys
  - Hashing: SHA-384
  - RSA/Diffie-Hellman: >= 3072-bit keys
  - Elliptic-curve Diffie-Hellman: 384-bit keys over P-384 (a public parameter curve defined by the NSA)
- Ideally use a good library that implements full protocols that use these algorithms under the hood

Nick’s preferred algorithms

- In an ideal world, I’d only use CNSA parameters, but this is extra paranoid
- In practice, many commercial algorithms are excellent
  - Symmetric-key encryption: AES with 128-bit keys
  - Hashing: SHA-256
  - RSA/Diffie-Hellman: 2048-bit keys
  - Elliptic-curve Diffie-Hellman: 256-bit keys over DJB curves (public parameters defined by Daniel J. Bernstein)
  - ChaCha20: a stream cipher developed by Daniel J. Bernstein

Nick’s preferred algorithms

- Symmetric-key block cipher modes of operation
  - Nick’s preferred mode of operation: CFB
  - Avoid CTR (counter) mode and modes that include counter modes, because failure is catastrophic
- AEAD (authenticated encryption with additional data) modes
  - Recall AEAD: provides confidentiality and integrity together
  - Nick’s preferred algorithm: AES-128-GCM (Galois Counter Mode)
  - Warning: GCM is not resistant to screw-ups
- Hashing
  - SHA-2 or SHA-3 family (256-bit, 384-bit, or 512-bit)
  - Never use SHA-1 or MD5: these are broken

Nick's preferred algorithms

- MACs
    - HMAC-SHA256 or HMAC-SHA3
    - Benefit: HMACs use hashes as the underlying function, and AES modes of operation use block ciphers as the underlying function. If you accidentally use the same key, it's probably safe, though you should never reuse keys anyway.
    - Always Encrypt-then-MAC!
- PRNGs
    - The best PRNG available: HMAC-SHA256-DRBG or HMAC-SHA3-DRBG
    - Seed using every source of entropy you have (e.g. built-in processor RNG)
    - When building software, avoid using only the processor's RNG. The processor can be sabotaged and uses designs without rollback resistance.
- Signatures
    - If possible, use RSA over DSA (or ECDSA): It resists screw-ups better

Summary

- Public-key cryptography: Two keys; one undoes the other
- Public-key encryption: One key encrypts, the other decrypts
  - Security properties similar to symmetric encryption
  - ElGamal: Based on Diffie-Hellman
    - The public key is g^b, and C_1 is g^r.
    - Not IND-CPA secure on its own
  - RSA: Produce a pair e and d such that $M^{ed} = M \bmod N$
    - Not IND-CPA secure on its own
- Hybrid encryption: Encrypt a symmetric key, and use the symmetric key to encrypt the message
- Digital signatures: Integrity and authenticity for asymmetric schemes
  - RSA: Same as RSA encryption, but encrypt the hash with the *private* key