IERG 4130 – 2026 Fall

香港中文大學
CUHK

Certificates, Password Hashing

Yajin ZHOU

https://yajin.org

Credits: the slides are from CS161 of UC Berkeley

Public-Key Encryption and Digital Signatures



- **Public-key cryptography: Two keys; one is shared with everyone**
- **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.
    - IND-CPA secure if appropriate padding is used
  - RSA: Produce a pair e and d such that $M^{ed} = M \bmod N$
    - IND-CPA secure if appropriate padding is used
- Hybrid encryption: Encrypt a symmetric key, and use the symmetric key to encrypt the message
- **Digital signatures: Integrity and authenticity**
  - RSA signature: can only be computed with private key d, but anyone can verify

Review: Public-Key Cryptography



- Public-key cryptography is great! We can communicate securely without a shared secret
  - Public-key encryption: Everybody encrypts with the public key, but only the owner of the private key can decrypt
  - Digital signatures: Only the owner of the private key can sign, but everybody can verify with the public key
- **What's the catch?**

Problem: Distributing Public Keys



- **Public-key cryptography alone is not secure against man-in-the-middle attacks**
- **Scenario**
  - Alice wants to send a message to Bob
  - Alice asks Bob for his public key
  - Bob sends his public key to Alice
  - Alice encrypts her message with Bob’s public key and sends it to Bob
- **What can Mallory do?**
  - Replace Bob's public key with Mallory’s public key
  - Now Alice has encrypted the message with Mallory’s public key, and Mallory can read it!

Problem: Distributing Public Keys



Alice

Mallory



Bob



Generate PKB, SKB

Send PKB

Send PKM

Receive PKM

Send {M}PKM

Decrypt {M}PKM

Send {M}PKB

Decrypt {M}PKB

Problem: Distributing Public Keys



- **Idea: Sign Bob's public key to prevent tampering**
- **Problem**
  - If Bob signs his public key, we need his public key to verify the signature
  - But Bob's public key is what we were trying to verify in the first place!
  - Circular problem: Alice can never trust any public key she receives
- **You cannot gain trust if you trust nothing. You need a root of trust!**
  - **Trust anchor**: Someone that we trust completely
  - From our trust anchor, we can begin to trust others

Certificates

- **Certificate: A signed endorsement of someone's public key**
  - A certificate contains at least two things: The **identity** of the person, and their **public key**
- **Abbreviated notation**
  - Encryption under a public key *PK*: {"Message"}$_{PK}$
  - Signing with a private key *SK*: {"Message"}$_{SK^{-1}}$
    - Recall: A signed message must contain the message along with the signature; you can't send the signature by itself!
- **Scenario: Alice wants Bob's public key. Alice trusts EvanBot (PK_E, SK_E)**
  - EvanBot is our trust anchor
  - If we trust PK_E, a certificate we would trust is {"Bob's public key is PK_B"}$_{SK_E^{-1}}$

Certificates: how you could have come up with them



- Suppose Alice wants Bob's key. Alice trusts EvanBot (PK_E, SK_E)
- Alice -> EvanBot: "What is Bob's public key?"
- EvanBot -> Alice: {“Bob’s public key is PK_B”}$_{SK_E^{-1}}$

Attempt #1: The Trusted Directory

- Idea: Make a central, trusted directory (TD) from where you can fetch anybody's public key
  - The TD has a public/private keypair PK_{TD}, SK_{TD}
  - The directory publishes PK_{TD} so that everyone knows it (baked into computers, phones, OS, etc.)
  - When you request Bob's public key, the directory sends a certificate for Bob's public key
    - {"Bob's public key is PK_B"}$_{SK_{TD}^{-1}}$
  - If you trust the directory, then now you trust every public key from the directory
- **What do we have to trust?**
  - We have received TD's key correctly
  - TD won't sign a key without verifying the identity of the owner

Attempt #1: The Trusted Directory



- **Let's say that Prof. Dennis Lo Yuk-ming (Vice-chancellor of CUHK) runs the TD**
  - We want my public key: Ask Prof. Lo
  - We want another Professor's public key: Ask Prof. Lo
  - Prof. Lo has better things to do (like making sure his private key isn't stolen)!
- **Problems: Scalability**
  - One directory won't have enough compute power to serve the entire world
- **Problem: Single point of failure**
  - If the directory fails, *cryptography stops working*
  - If the directory is compromised, you can't trust anyone
  - If the directory is compromised, it is difficult to recover

Certificate Authorities

- **Addressing scalability: Hierarchical trust**
  - The roots of trust may **delegate** trust and signing power to other authorities
    - {“Prof. TSANG’s public key is PK_{TSANG}, and I trust him to sign for the faculty of engineering”}$_{SK_{LO}^{-1}}$
    - {“Prof. Xing’s public key is PK_{XING}, and I trust him to sign for the IE department”}$_{SK_{TSANG}^{-1}}$
    - {“Prof. Zhou’s public key is PK_{ZHOU} (but I don’t trust him to sign for anyone else)”}$_{SK_{XING}^{-1}}$
  - Prof. LO is still the root of trust (**root certificate authority**, or **root CA**)
  - Prof. TSANG and Prof. XING receive delegated trust (**intermediate CAs**)
  - Prof. ZHOU’s identity can be trusted (but cannot sign for others)

Certificate Authorities

- **Addressing scalability: Multiple trust anchors**
  - There are ~100 root CAs who are implicitly trusted by most devices
  - Public keys are hard-coded into operating systems and devices
  - Each delegation step can restrict the scope of a certificate's validity
  - Creating the certificates is an *offline* task: The certificate is created once in advance, and then served to users when requested

Revocation

- **What happens if a certificate authority messes up and issues a bad certificate?**
  - Example: {“Bob’s public key is PK_M”}$_{SK_{CA}^{-1}}$
  - Example: Verisign (a certificate authority) accidentally issued a certificate saying that an average Internet user’s public key belonged to Microsoft

Revocation: Expiration Dates



- **Approach #1: Each certificate has an expiration date**
  - When the certificate expires, request a new certificate from the certificate authority
  - The bad certificate will eventually become invalid once it expires
- **Benefits**
  - Mitigates damage: Eventually, the bad certificate will become harmless
- **Drawbacks**
  - Adds management burden: Everybody has to renew their certificates frequently
  - If someone forgets to renew a certificate, their website might stop working
- **Tradeoff: How often should certificates be renewed?**
  - Frequent renewal: More secure, less usable
  - Infrequent renewal: Less secure, more usable
- **LetsEncrypt (a certificate authority) chose very frequent renewal**
  - 90 days now, but will change to 64 days in Feb 2027

Revocation: Announcing Revoked Certificates

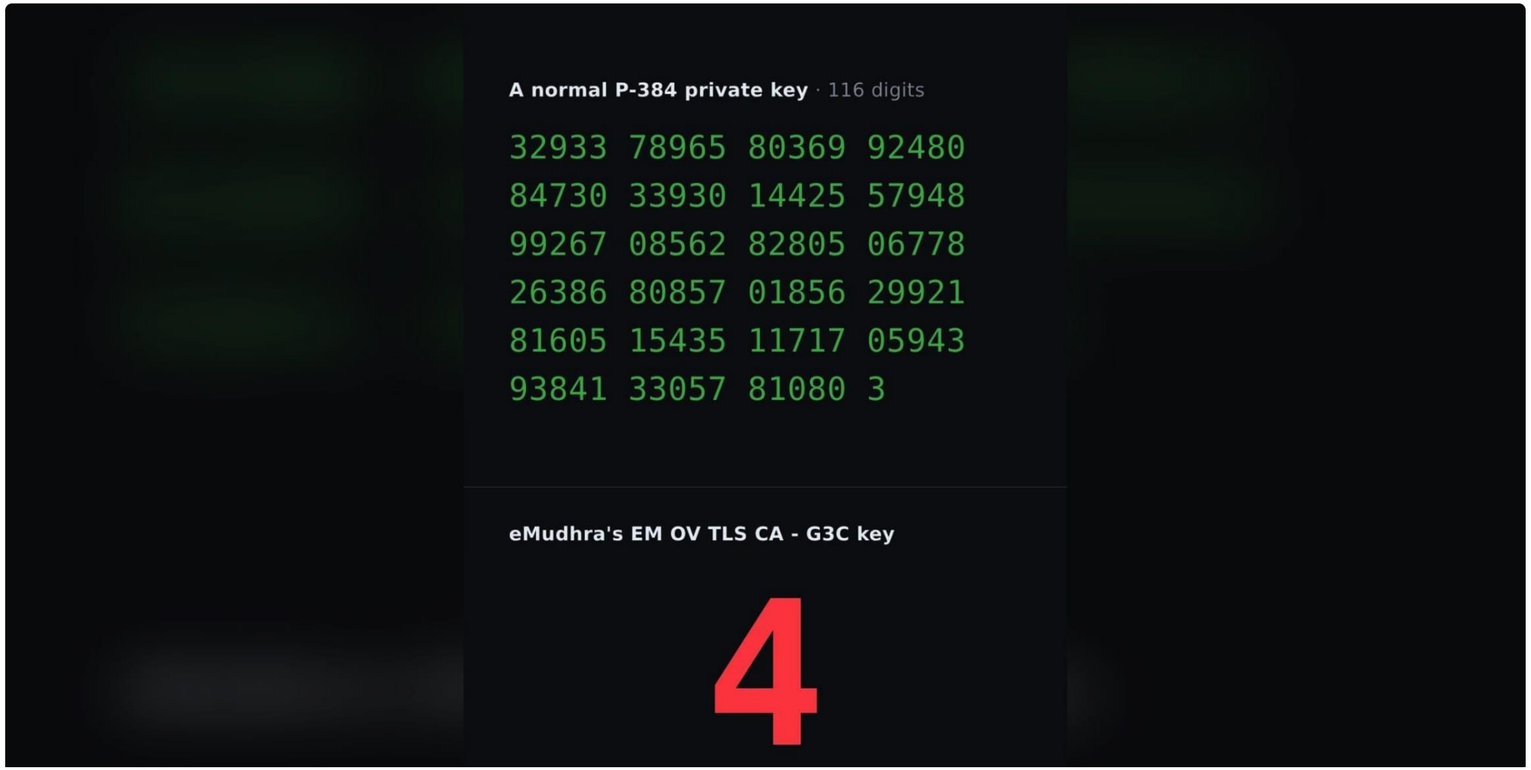


- **Approach #2: Periodically release a list of invalidated certificates**
  - Users must periodically download a Certification Revocation List (CRL)
- **How do we authenticate the list?**
  - The certificate authority signs the list!
    - {“The certificate with serial number 0xdeadbeef is now revoked”}$_{SK_{CA}^{-1}}$
- **Drawbacks**
  - Lists can get large
    - Mitigated by shorter expiration dates (don’t have to list them once they expire)
  - Until a user downloads a list, they won’t know which certificates are revoked
- **What happens if the certificate authority is unavailable?**
  - Fail-safe default: Assume all certificates are invalid? Now we can’t trust anybody!
    - Possible attack: Attacker forces the CA to be unavailable (denial of service attack)
  - Use old list: Potentially dangerous if the old list is missing newly revoked certificates

What If CA Did Something Wrong

NEWSLETTER 24 October 2026 at 16:48

An Indian CA's Private Key Was Just the Number 4



Summary: Certificates



- **Certificates: A signed attestation of identity**
- Trusted directory: One server holds all the keys, and everyone has the TD's public key
  - Not scalable: Doesn't work for billions of keys
  - Single point of failure: If the TD is hacked or is down, cryptography is broken
- **Certificate authorities: Delegated trust from a pool of multiple root CAs**
  - Root CAs can sign certificates for intermediate CAs
  - Revocation: Certificates contain an expiration date
  - Revocation: CAs sign a list of revoked certificates

Review: Cryptographic Hashes



- **Hash functions accept arbitrarily large inputs**
- **Hashes "look" random**
  - Change a single bit on the input and each output bit has a 50% chance of flipping
  - And until you change the input, you can't predict which output bits are going to change
- **Hash functions are *fast***
  - Can operate at 100 MB/s - 3 GB/s
- **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 any pair of inputs $x' \neq x$ such that $H(x) = H(x')$.

Storing Passwords



- **Password: A secret string a user types in to prove their identity**
  - When you create an account with a service: Create a password
  - When you later want to log in to the service: Type in the same password again
- **How does the service check that your password is correct?**
- **Bad idea #1: Store a file listing every user's password**
  - Problem: What if an attacker hacks into the service? Now the attacker knows everyone's passwords! And people probably used the same password on many sites.
- **Bad idea #2: Encrypt every user's password before storing it**
  - Problem: The attacker could steal the passwords file *and* the key and decrypt everyone's passwords!
- We need a way to verify passwords *without* storing any information that would allow someone to recover the original password

Password Hashing



- **For each user, store a *hash* of their password**
- **Verification process**
  - Hash the password submitted by the user
  - Check if it matches the password hash in the file
- **What properties do we need in the hash?**
  - Deterministic: To verify a password, it has to hash to the same value every time
  - One-way: We don’t want an attacker who hacks the server and downloads all password hashes to be able to reverse hashes into original passwords

Password Hashing: Attacks



- **What if two different users decide to use password123 as their password?**
  - Hashes are deterministic: They'll have the same password hash
  - An attacker can see which users are using the same password
- **Brute-force attacks**
  - Most people use insecure, common passwords
  - An attacker can pre-compute hashes for common passwords: *H*(**"password123"**), *H*(**"password1234"**), *H*(**"1234567890"**), etc.
  - **Dictionary attack**: Hash an entire dictionary of common passwords
- **Rainbow tables**: A data structure for storing the list of hashes of common passwords, to make brute-force attacks easier

Salted Hashes



- **Solution #1: Add a unique, random salt for each user**
- **Salt: A random, public value designed to make brute-force attacks harder**
  - For each user, store: username, salt, *H*(password || salt)
  - To verify a user: look up their salt in the passwords file, compute *H*(password || salt), and check it matches the hash in the file
  - Salts should be long and random
  - Salts are not secret (think of them like nonces or IVs)
- **Brute-force attacks are now harder**
  - Assume there are *M* possible passwords and *N* users in the database
  - Unsalted database: Hash all possible passwords, then lookup all users' hashes ⇒ $O(M + N)$
  - Salted database: Hash all passwords for each user's salt ⇒ $O(MN)$

Slow Hashes



- **Solution #2: Use slower hashes**
- **Cryptographic hashes are usually designed to be fast**
  - SHA is designed to produce a checksum of your 1 GB document as fast as possible
- **Password hashes are usually designed to be slow**
  - Legitimate users only need to submit a few password tries. Users won't notice if it takes 0.0001 seconds or 0.1 seconds for the server to check a password.
  - Attackers need to compute millions of hashes. Using a slow hash can slow the attacker by a factor of 1,000 or more!
  - Note: We are not changing the asymptotic difficulty of attacks. We're adding a large constant factor, which can have a huge practical impact for the attacker

Offline and Online Attacks



- **Offline attack: The attacker performs all the computation themselves**
  - Example: Mallory steals the password file, and then computes hashes herself to check for matches.
  - The attacker can try a huge number of passwords (e.g., use many GPUs in parallel)
  - Defenses: Salt passwords, use slow hashes
  - If an attacker can do an offline attack, you need a really strong password (e.g. 7 or more random words)
- **Online attack: The attacker interacts with the service**
  - Example: Mallory tries to log in to a website by trying every different password. Mallory is forcing the server to compute the hashes.
  - The attacker can usually only try a few times per second, with no parallelism
  - Defenses: Add a timeout or rate limit the number of tries to prevent the attacker from trying too many times

Summary: Password Hashing



- Store hashes of passwords so that you can verify a user's identity without storing their password
- Attackers can use brute-force attacks to learn passwords (especially when users use weak passwords)
  - Defense: Add a different **salt** for each user: A random, public value designed to make brute-force attacks harder
- **Offline attack: The attacker performs all the computation themselves**
  - Defense: Use salted, slow hashes instead of unsalted, fast hashes
- **Online attack: The attacker interacts with the service**
  - Defense: Use timeouts

Passwords Suck

- There is broad understanding that passwords have many problems: users choose guessable passwords, they reuse them on many sites, and users can be tricked into revealing their password (phishing emails, fake websites)
- Why are passwords still so widely used? They are easy for users to understand, familiar, and don't lock users into a particular device or technology
  - If I drop my phone in the ocean, can I still get into my accounts?
- **Nonetheless, the tech community is developing better alternatives**

Password Alternatives

- Password managers: your browser remembers and syncs your passwords, so you can feel free to choose hard-to-guess passwords
- **Email-based authentication: the website emails you a magic link to log in**
- **Two-factor authentication: you must provide both a password, and also authorize the login on an app on your phone**
- Passkeys: store a personal private key on your phone, server remembers your public key, uses public-key signatures + encryption to authenticate you; private key is often stored and protected with special hardware