Key management

Ms. Swati Mali
swatimali@somaiya.edu
Assistant Professor, Department of Computer Engineering
K. J. Somaiya College of Engineering
Somaiya Vidyavihar University

Key management

- Why build a strong cryptanalysis machine when someone can be bribed with a cheaper and easier deal?

https://arstechnica.com/tech-policy/2014/08/john-walker-the-navy-spy-who-defined-crypto-betrayal-dead-at-77/

- Ames, a 31-year CIA employee, and his wife, Rosario, were arrested Feb. 21 and charged with taking $1.5 million from the Soviets in exchange for classified information

https://library.cqpress.com/cqalmanac/document.php?id=cqal94-1102497

DiskLock program for Macintosh!!

- DiskLock program for Macintosh (version 2.1), sold at most software stores, claimed the security of DES encryption.
- It encrypts files using DES.
- However, DiskLock stores the DES key with the encrypted file.
- If you know where to look for the key, and want to read a file encrypted with DiskLock's DES, recover the key from the encrypted file and then decrypt the file.
- It doesn't matter that this program uses DES encryption—the implementation is completely insecure.

Generating keys

- The security of an algorithm rests in the key.
- If you're using a cryptographically weak process to generate keys, then the whole system is weak.
- Eve need not cryptanalyze the encryption algorithm; she can cryptanalyze the key generation algorithm.

Generating keys

- Keyspaces

Exhaustive Search of Various Keyspaces (assume one million attempts per second)

	4-Byte	5-Byte	6-Byte	7-Byte	8-Byte
Lowercase letters (26):	.5 seconds	12 seconds	5 minutes	2.2 hours	2.4 days
Lowercase letters and digits (36):	1.7 seconds	1 minute	36 minutes	22 hours	33 days
Alphanumeric characters (62):	15 seconds	15 minutes	16 hours	41 days	6.9 years
Printable characters (95):	1.4 minutes	2.1 hours	8.5 days	2.2 years	210 years
ASCII characters (128):	4.5 minutes	9.5 hours	51 days	18 years	2300 years
8-bit ASCII characters (256):	1.2 hours	13 days	8.9 years	2300 years	580,000 years

Generating keys

- Poor Key Choices
  - dictionary attack
- Random Keys
- Pass Phrases
- X9.17 Key Generation

What Is a Keyspace?

- A **keyspace** is the set of all valid keys that an algorithm can use.
- Example: If you use 128-bit keys, the keyspace has 2^128 possible keys.

What Makes a Keyspace Linear vs. Nonlinear

- **Linear Keyspace:**
  - If small changes in the key result in predictable or proportional changes in the output (ciphertext), the system is said to be **linear**.
  - Often **insecure**, as it enables attacks like **linear cryptanalysis**.
- **Nonlinear Keyspace:**
  - The relationship between the key and the cryptographic output is **non-predictable**, even with small changes.
  - Ensures that attackers cannot "interpolate" or guess keys using algebraic or statistical methods.

Nonlinear Keyspaces

- All the keys are not equally strong.
- Use an algorithm such that certain keys are stronger than others.
- Avoid predictable key patterns to prevent attacks like brute force and dictionary attacks.
- works if:
  - the algorithm is secret and the enemy can't reverse-engineer it,
  - or if the difference in key strength is subtle enough that the enemy can't figure it out.
- The NSA did this with the secret algorithms in their Overtake modules

Nonlinear Keyspaces

- A **nonlinear keyspace** ensures that keys do not produce outputs in a predictable or algebraically simple way, making the system robust against mathematical attacks.
- It's a fundamental **design principle in modern cryptography** — not a specific algorithm, but a property essential for secure key generation and cryptographic strength.

Create non-linear keyspace **Using Nonlinear Mathematical Structures**

a) Elliptic Curve Groups

- Base cryptographic operations (like scalar multiplication) on elliptic curves over finite fields.
- The keyspace is the set of scalars k such that public key = k × G.
- This space is nonlinear due to the elliptic curve point addition rules.
- ✓ **Used in**: ECC (Elliptic Curve Cryptography), ECDSA, ECDH

b) Multivariate Polynomial Systems

- Use sets of **nonlinear equations** over finite fields.
- Public and private keys are generated via nonlinear mappings.
- ✓ **Used in**: Rainbow, HFE (Hidden Field Equations), other post-quantum schemes.

c) Lattices and Ideal Lattices

- Operations like rounding or closest vector problems in high-dimensional space behave nonlinearly.
- The key derivation process includes Gaussian sampling or trapdoor functions that resist linear modeling.
- ✓ **Used in**: NTRU, Kyber, FrodoKEM, etc.

Create non-linear keyspace :Apply Nonlinear Transformations in Key Schedules

a) Use of S-boxes

- S-boxes (substitution boxes) are **nonlinear bijective functions** used in block ciphers.
- Ensure the key material goes through S-boxes before being mixed into the encryption rounds.
- ✓ **Used in**: AES, DES, Serpent, Camellia

b) Bit Permutations and Modular Arithmetic

- Combining bit-level permutation with modular additions or multiplications introduces nonlinearity.
- Example: key[i] = (S_box[key[i]] + rotate(key[i-1], r)) mod 256

Where Are Nonlinear Keyspaces Used?

- **Symmetric Ciphers:**
  - **AES**: The S-box (substitution box) is highly nonlinear.
  - Nonlinear mixing functions (e.g., in substitution–permutation networks) ensure the keyspace is complex.
- **Asymmetric Cryptography:**
  - **Elliptic Curve Cryptography (ECC)**:
    - Uses a **nonlinear algebraic structure**: elliptic curves over finite fields.
    - Scalar multiplication on the curve is nonlinear.
- **Post-Quantum Cryptography:**
  - Many lattice- and multivariate polynomial-based schemes use **nonlinear transformations** in key generation to resist quantum attacks.

Transferring Keys

- use a random-key generator at sender's and receiver's side
- Use a trusted messenger
- Use secure key exchange algorithms
  - Deffie Hellman (?)
  - ECDH
  - QCDH
- Out of band communication channel
- Ring of trust
- Digital certificate
- Key distribution center
- PGP
- Secure communication channel
- splits the key into several different parts and sends each of those parts over a different channel.

Verifying Keys

- Use a shared key-encryption key which is known only to sender and reciever
- use a digital signature protocol to sign the key, reciever has to trust the public-key database when he verifies that signature.
- Key Distribution Center (KDC) signs sender's public key

Verify **Symmetric Keys**

- Symmetric keys (e.g., for AES) are **shared secrets**, so verification is more limited but still important.
- ✔ **Common Practices:**
- **Key fingerprints**: Hash (e.g., SHA-256) of the key to verify integrity
- **MAC (Message Authentication Code) test**:
  - Encrypt & decrypt a known value using the key and verify the result.
- **Key Check Value (KCV)**: Encrypt a block of zeros and use that as a short checksum
- Example:
  AES key → encrypt 0x0000000000000000
  Resulting ciphertext = KCV (Key Check Value)

Verify Asymmetric Keys (Public/Private Key Pairs)

- Verifying that public and private keys form a **valid cryptographic pair** is essential.
- **Private Key ↔ Public Key Consistency**
- For **RSA**:
  - Verify that encrypt(decrypt(data, d), e) == data
- For **ECC**:
  - Confirm public key = private_key × G (G is the base point)
- For **DSA/ECDSA**:
  - Sign a test message and verify with the public key
- **Certificate-Based Verification**
- In PKI (Public Key Infrastructure):
  - Use a **digital certificate** signed by a Certificate Authority (CA)
  - Verify with the CA's public key that the signature on the certificate is valid

Verify **Key Fingerprints & Hashes**

- Public keys are often hashed (e.g., using SHA-256) to generate a **fingerprint**
- These fingerprints can be compared:
  - Out-of-band (e.g., phone call confirmation)
  - Through UI prompts ("Does the displayed fingerprint match?")

Verify Chain of Trust Validation (PKI)

- To verify public keys in secure communications (like HTTPS, email):
- Check certificate signature
- Verify certificate chain up to a trusted root CA
- Check expiration, revocation status (via CRL or OCSP)
- Match subject/common name to the domain or user
- Example: TLS in a browser

Verify Secure Key Storage Verification

- When keys are stored (e.g., in hardware or software keystores):
- Validate **key integrity** using HMACs or hashes
- Ensure **correct decryption** of encrypted key blobs using KDFs
- Confirm **no unauthorized access** via secure audit logs

Using keys

Principle	Goal
Purpose limitation	Prevent misuse of keys
Access control	Ensure only authorized use
Strong encryption	Guarantee confidentiality
Auditing & logging	Enable traceability
Ephemeral key usage	Reduce exposure in memory

Using Keys

- What if the encryption process is preempted over during its execution?
  - Operating system might write the encryption application to disk, along with key.
  - The key will sit on the disk, unencrypted, until the computer writes over that area of memory again.
  - Could be exploited by some adversary
- Solution: set the encryption operation to a high enough priority so it will not be interrupted.
- Many encryption devices could be designed to erase the key if tampered with.
  - For example, the IBM PS/2 encryption card has an epoxy unit containing the DES chip, battery, and memory.
- Telephone encryptors use session keys and it is discarded after the use.

Using keys

1. Use Keys for Their Intended Purpose Only

Separation of Duties:
Use different keys for:

- Encryption vs. signing
- Data encryption vs. key encryption (DEK vs. KEK)

- Prevents misuse or cryptographic vulnerabilities

2. Control Key Access

Restrict access using **Access Control Lists (ACLs)** or **Role-Based Access Control (RBAC)**

Integrate with **Identity and Access Management (IAM)** systems (e.g., AWS IAM, Azure AD)

Using keys

3. Always Use Authenticated Encryption

- Use algorithms like **AES-GCM** or **ChaCha20-Poly1305**
- Prevents both data **leakage** and **tampering**

4. Ensure Secure Key Usage in Code

- Never hardcode keys in source code
- Use secure APIs and key references from **Hardware Security Modules (HSMs)** or **Key Management Services (KMSs)**

Using keys

5. Use Keys with Proper Cryptographic Protocols

- Use **TLS (Transport Layer Security)** when transmitting keys
- Use **hybrid encryption**: RSA for key transport + AES for data

6. Audit and Monitor Key Usage

- Log every key operation: encrypt, decrypt, sign, verify
- Monitor for:
  - Unexpected usage patterns
  - Unauthorized access attempts

Using keys

8. Minimize Key Lifetime in Memory

- Load keys only when needed
- Zero out memory after use to prevent leakage through memory dumps

9. Ensure Compatibility and Interoperability

- Keys may need to follow formats like **PKCS#8**, **PEM**, or **DER**
- Important in multi-vendor or cloud-hybrid environments

Updating Keys

Why Update Keys?

- **Key aging**: Over time, keys become vulnerable due to advances in computing or cryptanalysis.
- **Compromise**: If a key is suspected to be exposed or misused.
- **Policy requirements**: Security policies or regulations (e.g., PCI DSS, NIST) mandate periodic updates.
- **User/device rotation**: When roles, devices, or users change.

How to update key?

1. Key Rotation

- Replacing an old key with a new one while maintaining continuity.
- **Types**:
  - **Manual rotation**: Admin-initiated (e.g., reissuing API keys).
  - **Automatic rotation**: Scheduled or policy-driven (common in cloud KMS).

Updating Keys

2. Forward Secrecy

- Old communications cannot be decrypted even if a current key is compromised.
- Achieved by **frequent key updates** or using **ephemeral session keys** (e.g., in TLS with Diffie-Hellman).

Updating Keys

4. Key Rotation Mechanisms

- **Dual key usage** (during transition):
  - Old key is used to decrypt
  - New key is used to encrypt
- **Key wrapping**:
  - Encrypt new keys with a master Key Encryption Key (KEK)
- **Versioning**:
  - Track key versions (e.g., Key_v1, Key_v2) to manage lifecycle

Updating Keys

5. Re-encryption Strategy

- When keys are updated, **data encrypted with old keys** may need to be:
  - **Re-encrypted** with the new key
  - Or **decrypted on access**, then re-encrypted on write

6. Key Derivation for Rotation

- Derive fresh sub-keys regularly using **Key Derivation Functions (KDFs)**:
  - E.g., HKDF (HMAC-based Key Derivation Function)

Updating Keys

7. Logging and Auditing

- Record when, why, and how keys are updated
- Ensure audit trails for compliance (e.g., SOC 2, HIPAA)

8. Emergency Key Updates (Key Revocation)

- If a key is compromised:
  - Immediately revoke it
  - Notify affected systems/users
  - Initiate secure key replacement and data re-encryption

Updating Keys

9. Policy for Key Updates

- Define update intervals (e.g., every 6–12 months)
- Automate where possible
- Document procedures and assign roles

10. Tools and Services

- Use **Key Management Systems (KMS)** that support rotation:
  - AWS KMS, Google Cloud KMS, Azure Key Vault
- Use **HSMs (Hardware Security Modules)** for secure in-place updates

Updating Keys

Method	Purpose	Tools/Practices
Key Rotation	Regular update	KMS, versioning, dual usage
Key Revocation	Emergency response	CRL (Certificate Revocation List)
Re-encryption	Secure data migration	Background jobs, batch ops
Key Derivation	Subkey generation per use/session	HKDF, PBKDF2
Logging & Auditing	Compliance and traceability	SIEM, audit logs

- Replace old cryptographic keys with new ones **securely, with minimal disruption**, while maintaining **data confidentiality and integrity**.

Key Updating Methods

1. Key Rotation

- Replacing keys at regular intervals or upon security events.
- Most common form of key updating.

How it works:

- Generate a new key.
- Re-encrypt new data (or all data, if needed) with the new key.
- Retire the old key (sometimes still kept to decrypt past data).

Used in:

- Symmetric systems (e.g., AES key rotation)
- TLS certificates
- API keys

2. Key Derivation Functions (KDFs) with New Salts

- Derive new keys from a master secret + new random salt.
- Provides key agility and supports key separation for different users/sessions.

Examples:

- plaintext
- CopyEdit
- new_key = HKDF(master_secret, new_salt)
- Used in TLS 1.3, SSH, and password-based encryption.

Key Updating Methods

3. Key Wrapping and Rekeying

- Instead of decrypting and re-encrypting all data, wrap the data key with a new master key.

✔ **How it works:**

- Data is encrypted with a **data encryption key (DEK)**
- DEK is encrypted (wrapped) with a **key encryption key (KEK)**
- To rotate KEK:
  - Decrypt DEK with old KEK
  - Re-encrypt DEK with new KEK
  - No need to re-encrypt the data

Used in AWS KMS, Google Cloud KMS, HSMs

4. Forward Secrecy / Ephemeral Keys

- Frequently update keys per session or message.
- No long-term key reuse.

✔ Used in secure messaging (e.g., Signal Protocol), TLS 1.3

Key Updating Methods

5. Key Expiry and Auto-Renewal

- Keys are associated with expiration timestamps.
- Systems auto-generate and distribute new keys before expiry.
- Example:
  X.509 TLS certificates have a validity period (e.g., 90 days for Let's Encrypt)

6. Key Update Protocols (KUPs)

- Specific protocols for securely updating keys between parties.
- Examples:
  - IKEv2 Rekeying: In IPsec VPNs
  - TLS Session Ticket Update
  - Zigbee Key Update Procedure

7. Key Escrow and Re-encryption

- Old keys are archived in escrow.
- When updating keys, data is **re-encrypted in bulk** using new keys.
- Risky if not done securely – requires full access to plaintext.

When to Update Keys

Trigger Type	Examples
Time-based	Rotate keys every X days (e.g., every 90 days)
Security Event	Suspected breach or key compromise
Policy-based	Regulatory compliance (e.g., NIST mandates)
Usage-based	After N encryptions (as in AES-GCM nonce limits)
Personnel change	Key holder leaves the organization

Best Practices for key update strategy

- Automate key rotation using secure KMS (AWS KMS, HashiCorp Vault)
- Use **versioning** to track old and new keys
- Audit all key updates and store logs securely
- Ensure **backward compatibility** (e.g., for decrypting older data)
- Avoid simultaneous rotation across multiple systems unless coordinated

Summary : Key Update Methods

Method	Description	Used In
Key Rotation	Regular replacement of keys	TLS, APIs, symmetric crypto
KDF with new salt	Derive fresh keys from master	TLS, password encryption
Key Wrapping	Rotate KEKs without touching encrypted data	HSMs, cloud KMS
Forward Secrecy	Per-session or per-message ephemeral keys	Signal, TLS 1.3
Expiry & Auto-Renewal	Timed expiry triggers rotation	Certificates, JWTs
Key Update Protocols	Secure rekeying messages	IKEv2, Zigbee, TLS

Storing Keys

- Memorise the key(s) (!!!)
  - Keys are longer, not very easy to remember
  - Could turn out Single of failure
- key in a magnetic stripe card, plastic key with an embedded ROM chip (called a ROM key), or smart card
- inserting the physical token into a special reader in his encryption box or attached to the computer terminal.
- Use key escrow technique
- regenerate the keys from an easy-to-remember password every time they are required.
- A key should never appear unencrypted outside the encryption device.

Key Storage Requirements

Property	Description
Confidentiality	Prevent unauthorized access to the key
Integrity	Ensure the key is not modified or corrupted
Availability	Ensure the key can be retrieved when needed by authorized systems/users
Auditability	Ensure every access, change, and operation on the key is logged and traceable

Key Storage Techniques

1. Hardware Security Module (HSM)

- A physical device would securely generate, store, and use cryptographic keys.
- Will Ensure that private keys **never leave the secure hardware boundary**.
- Certified devices often comply with **Federal Information Processing Standard (FIPS) 140-2 or 140-3**.

Use Cases:

- Root key storage
- Payment systems
- Digital certificates (Public Key Infrastructure)

Pros:

- Physically tamper-resistant
- High-performance cryptographic operations
- Non-extractable keys

Cons:

- Expensive
- Requires physical and network integration

2. Trusted Platform Module (TPM)

- A dedicated microcontroller integrated into computers
- Stores cryptographic keys, digital signatures, and integrity measurements

Use Cases:

- Full-disk encryption (e.g., BitLocker)
- Platform attestation
- Secure boot and identity keys

3. Key Management Service (KMS)

- A cloud-based or on-premises service for centralized key storage and operations
- Often backed by HSMs or software-based secure enclaves

Examples:

- Amazon Web Services Key Management Service (AWS KMS)
- Microsoft Azure Key Vault
- Google Cloud Key Management Service (GCP KMS)

Features:

- Access control using **Identity and Access Management (IAM)**
- Automatic key rotation
- Audit logs via **CloudTrail**, **Azure Monitor**, or similar

4. Operating System–Level Keystores

- Windows- Data-Protection API (DPAPI)
- macOS / iOS- Keychain
- Linux-GNOME Keyring, KWallet
- Android -Android Keystore System

5. Application-Level or Software-Based Keystores

- Store encrypted keys within application configuration or files
- Encrypt keys using a **Key Encryption Key (KEK)**, often derived from user input via a **Key Derivation Function (KDF)** like **PBKDF2**, **scrypt**, or **Argon2**

Drawback: Security depends on how well the KEK is protected

6. Key Wrapping

Instead of storing a raw key directly, **key wrapping** means:

- Encrypting the target key with a **Key Encryption Key (KEK)**
- Storing the **wrapped key** instead

- This technique enables:
  - Layered protection
  - Separation of data and key encryption
  - Easier key rotation (re-wrap with new KEK)

Best Practices for Storing Keys

Practice	Description
Encrypt keys at rest	Always encrypt stored keys with a strong KEK or using HSM capabilities
Use non-exportable keys	Ensure keys can’t be exported from the storage module (if supported)
Use key hierarchy	Use a Data Encryption Key (DEK) for actual data, and a KEK to protect it
Enable auditing	Track key creation, access, rotation, and deletion
Validate integrity	Use Message Authentication Codes (MAC) or Hash-based Message Authentication Code (HMAC) to detect tampering
Rotate keys regularly	Reduce exposure in case a key is compromised

Common pitfalls to avoid

Pitfall	Risk
Storing plaintext keys	Immediate compromise if storage is accessed
Hardcoding keys in source code	Allows attackers to extract them from decompiled code
Weak KEKs	An encrypted key is only as strong as the KEK that protects it
No key rotation	Prolonged exposure to potential compromise
No access logs	Lack of audit trail impairs incident response

Backup Keys

- key escrow
- Split the key and store them with different people
- store the split pieces, encrypted with each of the officer's different public keys, on owner's hard disk. That way, no one gets involved with key management until it becomes necessary.

Compromised Keys

- If Alice's key is lost, stolen or otherwise compromised, then all her security is gone.
- If the compromised key was for a symmetric cryptosystem, Alice has to change her key and hope the actual damage was minimal.
- If it was a private key, she has bigger problems
- if a key-encryption key is compromised, the potential loss is extreme
- a private key's compromise must propagate quickly throughout the network.
- Any databases of public keys must immediately be notified
- Inform KDC if it's involved in key distribution
- Use Certificate revocation list- CRL

- Compromised Key Response:
  - Revoke and replace immediately.
  - Notify affected parties.
  - Check logs for unauthorized access.

Lifetime of Keys

- No encryption key should be used for an indefinite period.
- It should expire automatically like passports and licenses.
- Key Lifetime Considerations:
  - Shorter key lifetimes reduce exposure to attacks.
  - Use key expiration policies for automated rotation.
- Reasons-
  - The longer a key is used, the greater the chance that it will be compromised.
  - The longer a key is used, the greater the loss if the key is compromised.
  - The longer a key is used, the greater the temptation for someone to spend the effort necessary to break it
  - It is generally easier to do cryptanalysis with more ciphertext encrypted with the same key.

Lifetime of Keys

- Session keys – last only for a session
- Key-encryption keys don't have to be replaced as frequently.
- balance the inherent danger in keeping a key around for a while with the inherent danger in distributing a new one.
- Encryption keys used to encrypt data files for storage cannot be changed often, as The files may sit encrypted on disk for months or years

Lifetime of Keys

- Private keys used for digital signatures and proofs of identity may have to last years (even a lifetime).
- Private keys used for coin-flipping protocols can be discarded immediately after the protocol is completed.
- Even if a key's security is expected to last a lifetime, it may be prudent to change the key every couple of years. T
- he private keys in many networks are good only for two years; after that the user must get a new private key.
- The old key would still have to remain secret, in case the user needed to verify a signature from that period.
- But the new key would be used to sign new documents, reducing the number of signed documents a cryptanalyst would have for an attack.

Destroying Keys

- Given that keys must be replaced regularly, old keys must be destroyed.
- Old keys are valuable, even if they are never used again.
- If the key is written on paper, the paper should be shredded or burned
- If the key is in a hardware EEPROM, the key should be overwritten multiple times.
- If the key is in a hardware EPROM or PROM, the chip should be smashed into tiny bits and scattered to the four winds.
- If the key is stored on a computer disk, the actual bits of the storage should be overwritten multiple times or the disk should be shredded.

Destroying Keys

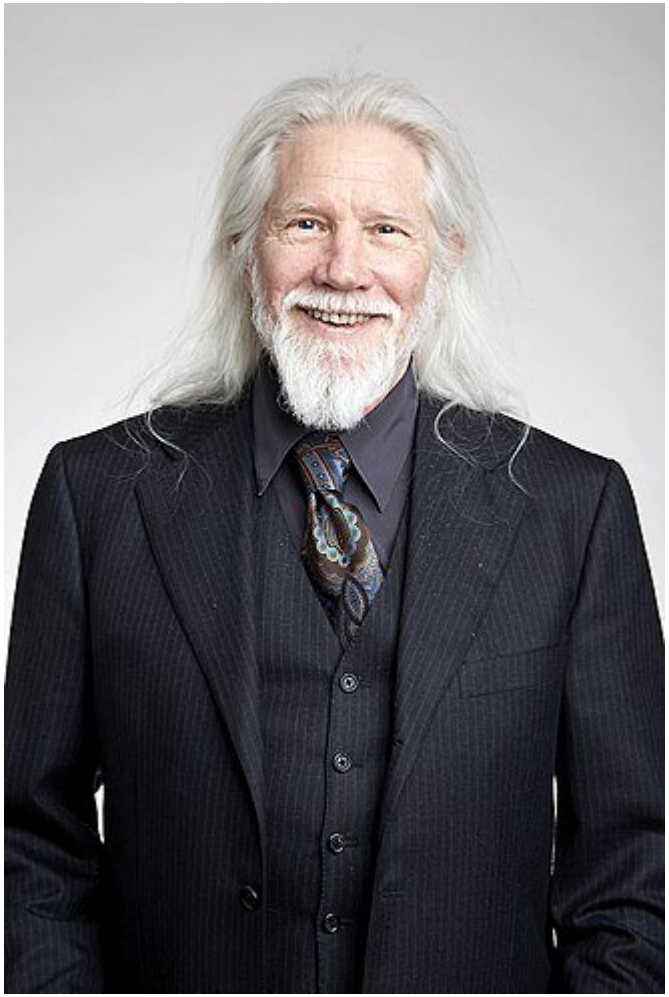
- Key Destruction:
  - Use secure wiping tools (e.g., overwrite with random data).
  - Physically destroy hardware storing sensitive keys.
- Public-Key Key Management:
  - Use Certificate Authorities (CAs) to issue and revoke public keys.
  - Implement Public Key Infrastructure (PKI) for secure identity management.

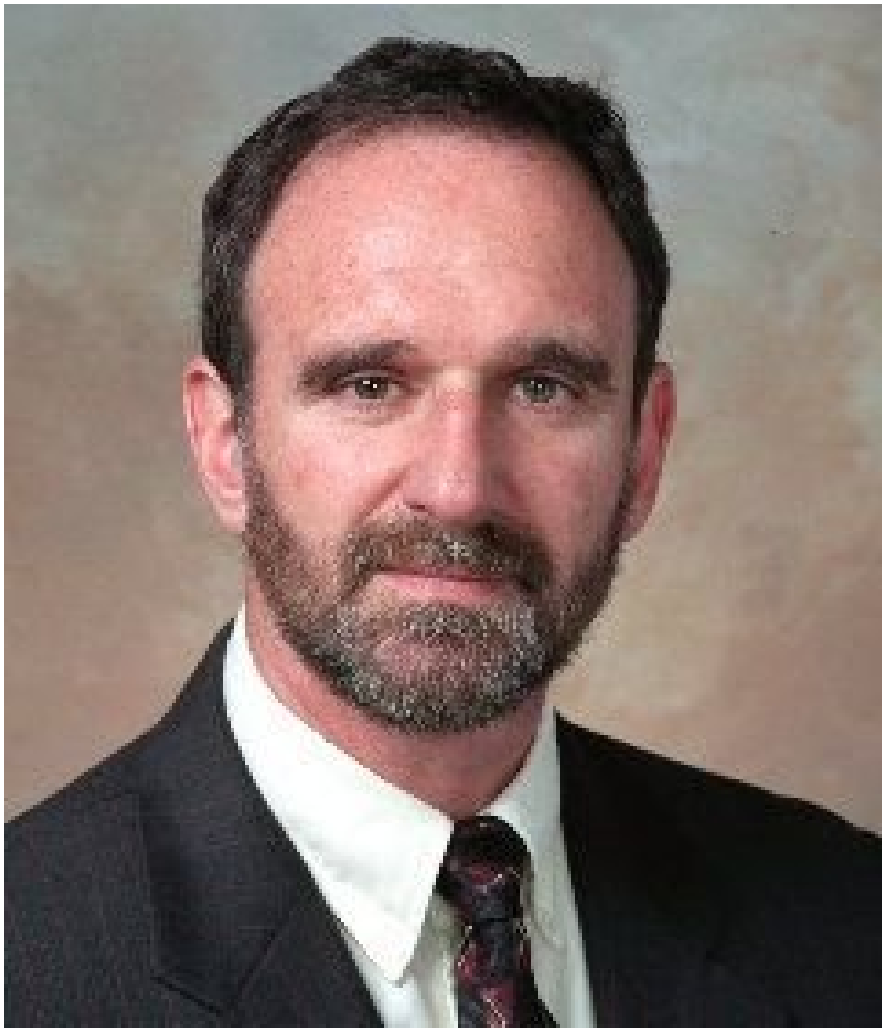
Key Exchange Algorithm : Diffie-Hellman Key Exchange Algorithm

Diffie-Hellman Key Exchange

- The **Diffie-Hellman Key Exchange** (DHKE) allows two parties to securely establish a shared secret key over an insecure channel.
- This shared key can then be used for encryption and decryption of messages.
- Invented in 1976
- relies on the mathematical properties of modular exponentiation and discrete logarithms, which are computationally hard to reverse.

Whitfield Diffie & Martin Hellman





Primitive root(generator)

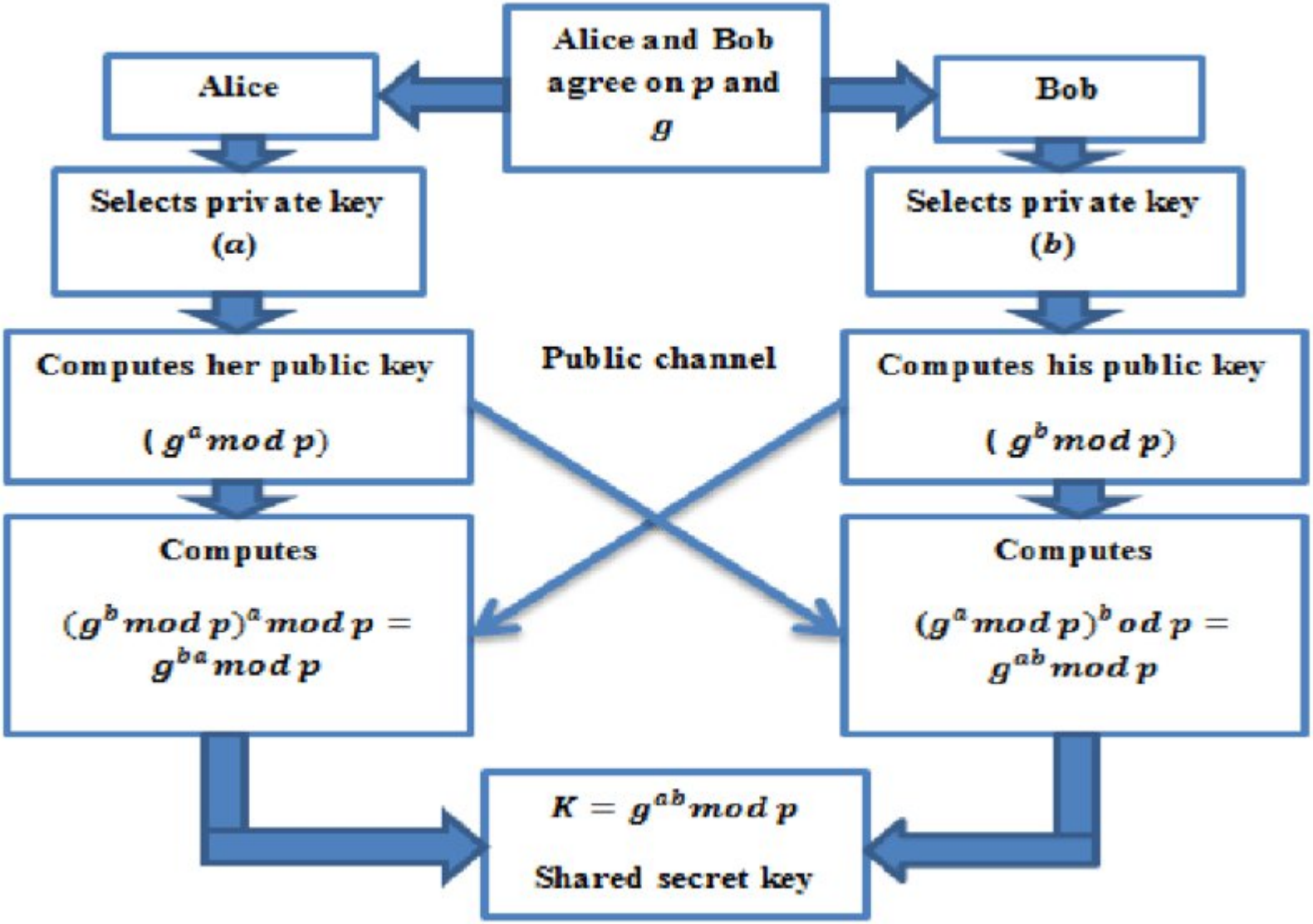
- Every prime number has a primitive root
- A **primitive root** of a prime number p is an integer g such that its powers generate all the coprime numbers from 1 to p−1 modulo p.
- That is, for every integer **a** such that $1 \leq a < p$, there exists some exponent k such that:

$$g^k \equiv a \bmod p$$

- A primitive root **g** modulo n, where n is a positive integer, is a number whose powers generate all the integers relatively prime to n.
- For example, 2 is a primitive root modulo 5
  - 2^1 mod 5 = 2
  - 2^2 mod 5 = 4,
  - 2^3 mod 5 = 3,
  - 2^4 mod 5 = 1
  - all the numbers relatively prime to 5 (which are 1, 2, 3, and 4).

- 3 is a primitive root modulo 7
  - 3^1 mod 7 = 3,
  - 3^2 mod 7 = 2,
  - 3^3 mod 7 = 6,
  - 3^4 mod 7 = 4,
  - 3^5 mod 7 = 5,
  - 3^6 mod 7 = 1
  - i.e. it generates all the numbers relatively prime to 7 (1, 2, 3, 4, 5, and 6)

D-H Algorithm



Let's assume:

p=23, g=5

- Alice picks a=6, Bob picks b=15

Compute Public Keys

Alice computes

$A=5^6 \bmod 23 = 15625 \bmod 23 = 8$

Bob computes

$B=5^{15} \bmod 23 = 30517578125 \bmod 23 = 19$

Compute Shared Key

Alice computes $S=19^6 \bmod 23 = 47045881 \bmod 23 = 2$

Bob computes $S=8^{15} \bmod 23 = 35184372088832v \bmod 23 = 2$

Both end up with the same shared secret key **S=2** which can now be used for encryption-decryption process

Agree upon two numbers:

P Prime Number 13

G Generator of P 6

Private = 5

Randomly generate a Private Key

Private = 4

(6^{5}) MOD 13

(7776) MOD 13

Public = 2

Calculate Public Key:

$(G^{Private})$ MOD P

(6^{4}) MOD 13

(1296) MOD 13

Public = 9

Exchange Public Keys

(9^{5}) MOD 13

(59049) MOD 13

Shared Secret = 3

Calculate the Shared Secret

$(Shared\ Public^{Private})$ MOD P

(2^{4}) MOD 13

(16) MOD 13

Shared Secret = 3

PRACTICAL NETWORKING .NET

Security of Diffie-Hellman

- The security is based on the difficulty of solving the **Discrete Logarithm Problem (DLP)**.
- Given $A=g^a \bmod p$, finding a (the private key) is computationally hard when p is large.
- However, Diffie-Hellman is vulnerable to **Man-in-the-Middle (MITM) attacks**,
  - MITM-where an attacker intercepts and replaces public keys.

Improvements over D-H

- Elliptic Curve Diffie-Hellman (ECDH)
- Quantum cryptography Diffie-Hellman (QCDH)
- Digital Signatures

Applications of Diffie-Hellman

- **Secure Web Browsing (HTTPS/TLS** for establishing secure communication between a browser and a server.
- **VPNs & Secure Messaging**: Used in **IPsec VPNs** and encrypted chat applications like **Signal and WhatsApp**.
- **Blockchain & Cryptocurrencies**: Secure peer-to-peer transactions.

Questions?