Cryptography: From Caesar to Quantum

Mathematics, Implementation, Exploitation, and Failure Modes

0x1337 Hacking Club

January 23, 2026

"Mathematics is the only truth. Everything else is implementation detail."

Roadmap: The Evolution of Secrecy

1. I. Classical Cryptography
2. II. Mathematical Foundations
3. III. Hashing & Password Security
4. IV. Symmetric Cryptography (AES)
5. V. Asymmetric Cryptography (RSA)
6. VI. Elliptic Curve Cryptography (ECC)
7. VII. Real-World Protocols & Failures
8. VIII. Advanced Topics & Future Cryptography

The Caesar Cipher (Shift Cipher)

The Logic: Shift every letter by a fixed amount k.

$$E_k(x) = (x + k) \bmod 26$$

$$D_k(x) = (x - k) \bmod 26$$

Example ($k = 3$):

`HELLO WORLD` $\rightarrow$ `KHOOR ZRUOG`

Attack 1: Brute Force

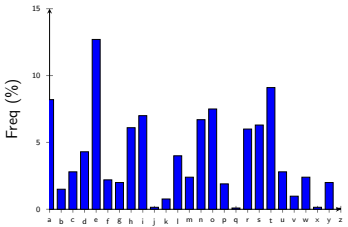
The Key Space is tiny! Only 25 possible keys (shifts 1-25). A human (or computer) can try all of them in seconds.

Historical Note: Named after Julius Caesar, who used $k = 3$ to protect military messages.

Attack 2: Frequency Analysis

Languages have fingerprints. In English, 'E' is the most common letter ($\approx 12.7\%$), followed by 'T' ($\approx 9.1\%$), 'A', 'O'.

If the ciphertext has a symbol appearing 13% of the time, that symbol likely represents 'E'.



Monoalphabetic Substitution: Even with random alphabet scrambling ($26! \approx 4 \times 10^{26}$ keys), the *frequency distribution* remains unchanged.

The Attack: Match ciphertext frequencies to known language patterns.

The Vigenère Cipher

Solution to Frequency Analysis: Polyalphabetic Substitution.
Use a keyword (e.g., "KEY") to create varying shifts:

- Position 1: Shift by 'K' = 10
- Position 2: Shift by 'E' = 4
- Position 3: Shift by 'Y' = 24
- Position 4: Repeat (back to 'K')

Effect: 'E' might encrypt as 'O' in position 1, 'I' in position 2, and 'C' in position 3. This flattens the frequency distribution.

The Crack: Kasiski Examination (1863)

1. Find repeating patterns in ciphertext (e.g., "XYZ" appears twice)
2. Measure distance between repetitions
3. Distance is likely a multiple of key length
4. Once key length L is determined, split ciphertext into L separate streams
5. Each stream is a simple Caesar cipher—apply frequency analysis to each

XOR: The Hacker's Swiss Army Knife

Exclusive OR ($\oplus$) Truth Table:

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Critical Properties:

- Self-Inverse: $A \oplus A = 0$
- Identity: $A \oplus 0 = A$
- Commutative: $A \oplus B = B \oplus A$
- Associative: $(A \oplus B) \oplus C = A \oplus (B \oplus C)$

Encryption/Decryption:

$$C = P \oplus K \quad \text{(Encrypt)}$$

$$P = C \oplus K = (P \oplus K) \oplus K = P \oplus (K \oplus K) = P \oplus 0 = P \quad \text{(Decrypt)}$$

Modern Usage: Stream ciphers (ChaCha20, RC4) generate pseudorandom keystreams to XOR with plaintext.

The One-Time Pad (OTP)

The only cipher with **proven perfect secrecy** (Shannon, 1949).
Three Absolute Requirements:

1. Key is **truly random** (not pseudorandom)
2. Key length $\geq$ message length
3. Key is **never reused** (hence "one-time")

Operation: $C = P \oplus K$
Mathematical Proof of Security: For any ciphertext C and any plaintext P', there exists a key $K' = P' \oplus C$ such that decryption yields P'.
Without the key, C is equally likely to decrypt to "ATTACK AT DAWN" or "RETREAT TONIGHT" or any message of the same length.

The Fatal Flaw: Key Distribution

Sharing a secure 1GB key is as hard as sharing a 1GB message securely. This is the problem public-key cryptography solves.

XOR Key Reuse Attack

The Vulnerability: Reusing a key stream with XOR encryption.
Given: $C_1 = P_1 \oplus K$ and $C_2 = P_2 \oplus K$
The Attack:

$$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = P_1 \oplus P_2$$

The key cancels out! Now we have two plaintexts XORed together.
Exploitation Techniques:

- **Crib Dragging:** Try known words/phrases ("the", "and", "http://")
- If P_1 contains "the" at position i, then $(P_1 \oplus P_2)[i : i+3] \oplus$ "the" reveals $P_2[i : i+3]$
- Build up both plaintexts incrementally

Real-World Example: Microsoft's PPTP VPN vulnerability (1998).

One-Way Functions & Trapdoors

Modern cryptography relies on **computational asymmetry**—functions easy to compute but hard to reverse.

One-Way Function:

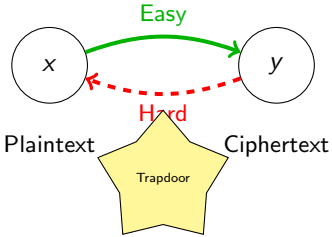
$$y = f(x) \text{ (Fast: milliseconds)}$$

$$x = f^{-1}(y) \text{ (Infeasible: millions of years)}$$

Example:

- Forward: $123456789 \times 987654321 = ?$ (easy)
- Reverse: Factor 121932631112635269 (hard)

Trapdoor Function: Reversing is hard *unless* you possess secret information (the trapdoor/private key).



RSA Trapdoor: Knowing the factorization of $N = p \times q$ makes decryption easy.

Modular Arithmetic: The Finite Field

Cryptography operates in **Finite Fields** (Galois Fields) for mathematical closure and efficiency.
The Ring $\mathbb{Z}_n$: Integers $\{0, 1, 2, \ldots, n-1\}$ with operations modulo n.
Example: In $\mathbb{Z}_7$: $5 + 4 = 9 \equiv 2 \pmod 7$, $3 \times 6 = 18 \equiv 4 \pmod 7$

The Modular Multiplicative Inverse

For $a \in \mathbb{Z}_n$, we seek x (denoted a^{-1}) such that:

$$a \cdot x \equiv 1 \pmod n$$

Existence Theorem: a^{-1} exists **if and only if** $\gcd(a, n) = 1$ (i.e., a and n are coprime).
Computation: Extended Euclidean Algorithm finds integers x, y such that:

$$ax + ny = \gcd(a, n) = 1$$

Then $x \bmod n$ is the inverse of a.
Example: Find $5^{-1} \pmod{26}$. Since $5 \times 21 = 105 = 4 \times 26 + 1$, we have $5^{-1} \equiv 21 \pmod{26}$.

Euler's Totient Function & Theorem

The mathematical backbone of RSA encryption.

Euler's Totient Function $\phi(n)$

Count of integers k where $1 \leq k < n$ and $\gcd(k, n) = 1$.

Key Properties:

- If p is prime: $\phi(p) = p - 1$
- If $n = p \times q$ (distinct primes): $\phi(n) = (p-1)(q-1)$
- Example: $\phi(15) = \phi(3 \times 5) = (3-1)(5-1) = 2 \times 4 = 8$

Euler's Theorem

If $\gcd(a, n) = 1$, then:

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

Special Case (Fermat's Little Theorem): If p is prime:

$$a^{p-1} \equiv 1 \pmod{p}$$

Consequence: $a^{k \cdot \phi(n)+1} \equiv a \pmod{n}$ — this is why RSA decryption works!

The Discrete Logarithm Problem (DLP)

The foundation of Diffie-Hellman and ElGamal cryptography.
Setup: Prime p, generator g of $\mathbb{Z}_p^*$
Forward (Easy): Given g, x, p, compute $y \equiv g^x \pmod{p}$
Using fast modular exponentiation: $O(\log x)$ operations
Reverse (Hard): Given g, y, p, find x such that $g^x \equiv y \pmod{p}$
Best known classical algorithms: subexponential time (e.g., Index Calculus)

Example

Let $p = 23, g = 5$. Computing 5^6 mod 23:

$$5^2 = 25 \equiv 2, \quad 5^4 \equiv 4, \quad 5^6 \equiv 2 \times 4 = 8 \pmod{23}$$

But given $5^x \equiv 8 \pmod{23}$, finding $x = 6$ requires trying multiple values.

Security: For 2048-bit primes, DLP is computationally infeasible with current technology.

Hash Functions: The Digital Fingerprint

Definition: A function mapping arbitrary size data to fixed size output.

$$h = H(M)$$

Critical Properties:

- **Deterministic:** Same input $\rightarrow$ Always same output.
- **Pre-image Resistance:** Given h, computationally infeasible to find M.
- **Collision Resistance:** Infeasible to find $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$.
- **Avalanche Effect:** Flipping 1 bit of input changes $\approx 50\%$ of output bits.

The Avalanche Effect (SHA-256)

```
Input: "cat"
Hash:  77af778b51abd4a3c51c5ddd97204a9c...

Input: "car" (1 bit difference from 't' to 'r')
Hash:  5c24e6c38ba1c8903c73400571343714...
```

Status: MD5 & SHA-1 (Broken), SHA-256 (Standard), SHA-3 (Future).

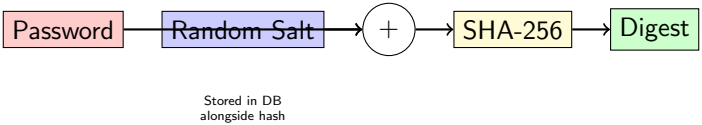
Password Storage: The Necessity of Salt

The Problem: Users pick weak passwords ("123456").

- If DB stores $H(P)$, then User A ("123456") and User B ("123456") have identical hashes.
- **Rainbow Tables:** Precomputed tables of $P \rightarrow H(P)$ allow instant lookup.

The Solution: Salting Add a unique, random string (S) to every password before hashing.

$$h = H(P \parallel S)$$



Result: Even if passwords are same, stored hashes are different. Rainbow tables are useless.

Attacking Hashes: Strategies

1. **Brute Force**
   - **Method:** Try every combination: ‘aaaa‘, ‘aaab‘...
   - **Pros:** Guaranteed to find it.
   - **Cons:** Extremely slow for long passwords ($O(N^L)$).
2. **Dictionary / Wordlist**
   - **Method:** Try words from a list (‘rockyou.txt‘).
   - **Pros:** extremely fast. Humans are predictable.
   - **Cons:** Fails on random strings.
3. **Rule-Based / Mask Attack**
   - **Method:** Smart brute force. "Word + Year" or "Capital + Word + Symbol".
   - *Example:* ‘Password123¡ fits mask ‘?u?l?l?l?l?l?l?l?d?d?d?s‘.

Tools of the Trade

- **Hashcat:** GPU-accelerated cracker. Can compute billions of hashes/sec.
- **John the Ripper:** CPU/GPU cracker, great for complex formats.

Defending Passwords: Slow it Down!

Why SHA-256 is BAD for passwords: It's too fast! An RTX 4090 can calculate billions of SHA-256 hashes per second. A hacker can bruteforce an 8-char password in minutes.
Key Stretching Algorithms: Intentionally slow functions requiring CPU/Memory work.

- **PBKDF2:** Applies HMAC-SHA256 thousands of times.

$$DK = PBKDF2(PRF, Password, Salt, c, dkLen)$$

  where c is iteration count (e.g., 600,000).
- **Bcrypt:** Based on Blowfish cipher. Tunable cost factor.
- **Argon2 (Winner of PHC):** The modern standard.
  - **Memory Hard:** Requires RAM to compute, resisting ASIC/GPU attacks.
  - Tunable: Time, Memory, Parallelism.

John the Ripper (JtR): The Heavy Hitter

Overview: The industry-standard, open-source password cracker.

- **Jumbo Version:** Supports hundreds of hash types (KRB5, NTLM, PDF, ZIP).
- **Autodetect:** Automatically identifies hash formats.

Three Main Attack Modes:

1. **Single Crack (Fastest):** Uses the user's metadata (username, GECOS) to guess passwords like "User123" or "Admin!".
2. **Wordlist (Dictionary):** Tries words from a file, applying **mangling rules** (e.g., append '1', reverse string).
3. **Incremental (Brute Force):** Tries all char combinations. Runs until stopped.

Typical Workflow (Linux Shadow File)

```
# 1. Combine /etc/passwd and /etc/shadow
unshadow /etc/passwd /etc/shadow > hashes.txt

# 2. Start cracking (Auto-mode: Single -> Wordlist -> Incr)
john hashes.txt

# 3. View cracked passwords
john --show hashes.txt

# Output:
```

AES Overview: Structure

Advanced Encryption Standard (Rijndael)

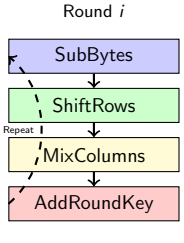
Adopted by NIST in 2001. Block size: 128 bits. Key sizes: 128, 192, 256 bits.

The Four Round Functions:

1. **SubBytes:** Non-linear byte substitution via S-box (provides *confusion*)
2. **ShiftRows:** Cyclically shift each row left (provides *diffusion*)
3. **MixColumns:** Matrix multiplication in $GF(2^8)$ (provides *diffusion*)
4. **AddRoundKey:** XOR with round key derived from main key

Number of Rounds:

- AES-128: 10 rounds
- AES-192: 12 rounds
- AES-256: 14 rounds



Final round omits MixColumns

AES Mathematics: The Galois Field $GF(2^8)$

AES treats bytes as polynomials over $GF(2)$ (coefficients are 0 or 1).

Byte $\rightarrow$ Polynomial Representation:

Byte $b = (b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0)_2 \implies b(x) = \sum_{i=0}^{7} b_i x^i$

Example: $0x53 = (01010011)_2 \implies x^6 + x^4 + x + 1$

Irreducible Polynomial (Modulus):

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

All polynomial operations are performed modulo $m(x)$.

SubBytes Transformation (The S-Box):

The *only* non-linear component in AES (critical for security).

1. Compute multiplicative inverse: b^{-1} in $GF(2^8)$ (by convention, $0^{-1} = 0$)
2. Apply affine transformation over $GF(2)$:

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

where $c = (01100011)_2$ is a constant

Purpose: Prevents linear and differential cryptanalysis.

AES MixColumns: Matrix Multiplication

MixColumns Operation: Treats each column of the state as a polynomial and multiplies by a fixed polynomial modulo $x^4 + 1$.
Represented as matrix multiplication in $GF(2^8)$:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

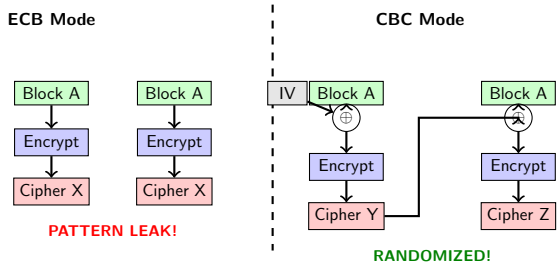
Where multiplication and addition are in $GF(2^8)$.
Example Computation:

$$b'_0 = (02 \cdot b_0) \oplus (03 \cdot b_1) \oplus (01 \cdot b_2) \oplus (01 \cdot b_3)$$

Multiplication by 02 (i.e., x): shift left and conditionally XOR with $0x1B$ if overflow.
Effect: Each output byte depends on all four input bytes—provides strong diffusion across the block.

Visualizing Modes: ECB vs CBC

ECB (Electronic Codebook): Each block encrypted independently.
Problem: Identical plaintext blocks → Identical ciphertext blocks.



Famous ECB Failure: The "ECB Penguin"—encrypting a bitmap image in ECB mode preserves visual patterns.

AES Modes of Operation

Common Modes:

- **CBC (Cipher Block Chaining):** Each plaintext block is XORed with previous ciphertext before encryption. Requires IV. Sequential encryption, parallel decryption.
- **CTR (Counter):** Turns block cipher into stream cipher. Encrypts counter values and XORs with plaintext. Fully parallelizable. Requires unique nonce.
- **GCM (Galois/Counter Mode):** CTR mode + authentication tag. Provides both confidentiality and integrity. Industry standard for TLS 1.3.
- **ECB:** Never use in production! Educational purposes only.

Critical: IV/Nonce Requirements

- CBC: IV must be unpredictable (use random IV)
- CTR/GCM: Nonce must NEVER repeat with same key (use counter or random 96-bit)

Symmetric vs. Asymmetric Cryptography

Symmetric (AES, ChaCha20)

- **Key:** Single shared secret
- **Speed:** Very fast (GB/s)
- **Key Size:** 128-256 bits
- **Problem:** Secure key exchange
- **Use:** Bulk data encryption

Asymmetric (RSA, ECC)

- **Keys:** Public (encrypt) / Private (decrypt)
- **Speed:** Slow (1000× slower)
- **Key Size:** 2048-4096 bits (RSA)
- **Advantage:** No key exchange needed
- **Use:** Key exchange, signatures

Hybrid Approach (TLS): Use RSA/ECDH to exchange a symmetric key, then use AES for actual data encryption.

RSA: Key Generation

Named after: Rivest, Shamir, Adleman (1977)

Key Generation Algorithm:

1. Choose two large distinct primes: p, q (typically 1024 bits each)
2. Compute modulus: $N = p \times q$ (2048 bits)
3. Compute Euler's totient: $\phi(N) = (p-1)(q-1)$
4. Choose public exponent: $e = 65537$ (common choice: $2^{16}+1$)
5. Verify: $\gcd(e, \phi(N)) = 1$
6. Compute private exponent: $d \equiv e^{-1} \pmod{\phi(N)}$ using Extended Euclidean Algorithm

Keys:

- Public Key: (N, e) — can be shared with anyone
- Private Key: (N, d) — must be kept secret (p, q are discarded but must remain secret)

Operations:

- Encrypt: $C \equiv M^e \pmod{N}$ (anyone can encrypt with public key)
- Decrypt: $M \equiv C^d \pmod{N}$ (only private key holder can decrypt)

RSA: Mathematical Proof of Correctness

Why does RSA work? We must show: $C^d \equiv M \pmod{N}$
Proof:

$$C^d \equiv (M^e)^d \equiv M^{ed} \pmod{N}$$

By key generation, $ed \equiv 1 \pmod{\phi(N)}$, so:

$$ed = 1 + k\phi(N) \text{ for some integer } k$$

Therefore:

$$\begin{aligned} M^{ed} &= M^{1+k\phi(N)} \\ &= M \cdot (M^{\phi(N)})^k \end{aligned}$$

Case 1: If $\gcd(M, N) = 1$, by Euler's Theorem: $M^{\phi(N)} \equiv 1 \pmod{N}$

$$M^{ed} \equiv M \cdot 1^k \equiv M \pmod{N} \quad \checkmark$$

Case 2: If $\gcd(M, N) \neq 1$, then M shares a factor with N (extremely rare for random $M < N$). Can be proven using Chinese Remainder Theorem. $\square$

RSA Security: Why is it Hard to Break?

The Security Assumption: Factoring large semiprimes is computationally infeasible.
Attack Vector: Given public key (N, e), attacker wants to find d.
The Problem:

- To compute d, need $\phi(N) = (p-1)(q-1)$
- To compute $\phi(N)$, need to know p and q
- To find p and q, must factor N

Best Known Classical Factoring Algorithm: General Number Field Sieve (GNFS)
Complexity: $O\left(\exp\left((c + o(1))(\ln N)^{1/3}(\ln \ln N)^{2/3}\right)\right)$ — subexponential but still infeasible for large N
Computational Reality:

- 768-bit RSA factored in 2009 (hundreds of CPU-years)
- 1024-bit: Estimated 10^9 CPU-years
- 2048-bit: Far beyond current capabilities

Quantum Threat: Shor's algorithm factors in polynomial time on quantum computers.

RSA Attack 1: Small Exponent ($e = 3$)

Scenario: Using small public exponent $e = 3$ without proper padding.
The Vulnerability: If message M is small such that $M^3 < N$:

$$C = M^3 \pmod{N} = M^3 \quad \text{(no modular reduction occurs!)}$$

The Attack: Attacker simply computes $\sqrt[3]{C}$ over the integers—no need to break RSA!
Example:

- $N = 100000$ (small for demo), $e = 3$
- Message: $M = 42$
- Ciphertext: $C = 42^3 = 74088$
- Attack: $\sqrt[3]{74088} = 42$ ✓

Mitigation: Use proper padding (OAEP) to ensure $M^e > N$, forcing modular reduction.
Modern Practice: Use $e = 65537$ and OAEP padding (RSA-OAEP).

RSA Attack 2: Common Modulus

Scenario: Same message M encrypted to two users with same N but different exponents e_1, e_2.
Given: $C_1 \equiv M^{e_1} \pmod{N}$ and $C_2 \equiv M^{e_2} \pmod{N}$
Condition: If $\gcd(e_1, e_2) = 1$ (coprime exponents)
The Attack: Use Extended Euclidean Algorithm to find a, b such that:

$$a \cdot e_1 + b \cdot e_2 = 1$$

Compute:

$$M \equiv C_1^a \cdot C_2^b \equiv (M^{e_1})^a \cdot (M^{e_2})^b \equiv M^{ae_1+be_2} \equiv M^1 \equiv M \pmod{N}$$

Example: $e_1 = 3, e_2 = 5$

- Extended Euclidean: $2 \cdot 3 + (-1) \cdot 5 = 1$
- Recovery: $M \equiv C_1^2 \cdot C_2^{-1} \pmod{N}$

Lesson: Never reuse modulus N across different key pairs!

RSA Attack 3: Padding Oracle (Bleichenbacher 1998)

PKCS#1 v1.5 Padding Format:

$$M_{\text{padded}} = 00 \parallel 02 \parallel \text{Random} \parallel 00 \parallel \text{Message}$$

The Oracle: Server returns different error messages:

- "Padding Error" — if decrypted message doesn't start with $00 \parallel 02$
- "Decryption Success" — if padding is valid

The Attack (Adaptive Chosen Ciphertext):

1. Intercept ciphertext C
2. Choose multiplier s and send: $C' \equiv C \cdot s^e \pmod{N}$
3. Server decrypts: $M' \equiv C'^d \equiv M \cdot s \pmod{N}$
4. Padding check reveals information about $M \cdot s$
5. Binary search on possible values of M

Complexity: $\approx 2^{20}$ oracle queries (feasible!)
Impact: DROWN attack (2016) exploited this in SSLv2 to break TLS connections.
Fix: Use OAEP padding + constant-time comparison.

RSA Attack 4: ROCA (CVE-2017-15361)

Return of Coppersmith's Attack — Infineon Chip Vulnerability
Background: Infineon's TPM chips (smart cards, HSMs, laptops) generated RSA keys with flawed primes.
The Flaw: To speed up prime generation, primes had special form:

$$p = k \cdot M + (65537^a \bmod M)$$

where $M = \prod(\text{small primes})$ and a is small.
Consequence: Drastically reduced prime space from 2^{1024} to $\approx 2^{40}$!
The Attack: Coppersmith's theorem finds small roots of polynomial equations modulo N using lattice reduction (LLL algorithm).
Timeline:

- **1996:** Coppersmith develops theorem
- **2012-2016:** Infineon ships vulnerable chips
- **2017:** Discovered by Czech researchers

Impact: Estonia's 750,000 national ID cards, millions of TPMs compromised.
Lesson: Optimizing cryptographic primitives is extremely dangerous.

Why Elliptic Curves?

The Efficiency Advantage:

Security Level	RSA Key Size	ECC Key Size
112-bit	2048 bits	224 bits
128-bit	3072 bits	256 bits
192-bit	7680 bits	384 bits
256-bit	15360 bits	521 bits

Benefits:

- Smaller keys $\rightarrow$ Less bandwidth
- Faster operations $\rightarrow$ Better for mobile/IoT
- Same security with $\frac{1}{10}$ the key size

Trade-off: More complex mathematics, implementation vulnerabilities.

ECC: The Weierstrass Equation

Elliptic Curve over Finite Field $\mathbb{F}_p$:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

Constraint: $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$ (non-singular curve)

Point Addition (Group Law):

Geometrically: Draw line through points P and Q. Line intersects curve at third point. Reflect across x-axis to get $R = P + Q$.

Algebraic Formulas:

For $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ with $P \neq Q$:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \pmod{p}$$

$$x_3 = \lambda^2 - x_1 - x_2 \pmod{p}, \quad y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p}$$

For point doubling ($P = Q$):

$$\lambda = \frac{3x_1^2 + a}{2y_1} \pmod{p}$$

Point at Infinity: Identity element $\mathcal{O}$ (like zero in addition).

ECC: Scalar Multiplication & ECDLP

Scalar Multiplication: $Q = k \cdot P = \underbrace{P + P + \cdots + P}_{k \text{ times}}$

Computed efficiently using "double-and-add" algorithm: $O(\log k)$ operations

Elliptic Curve Discrete Log Problem (ECDLP):

Forward (Easy): Given P and k, compute $Q = k \cdot P$

Reverse (Hard): Given P and Q, find k such that $Q = k \cdot P$

Best Known Attack: Pollard's rho algorithm — $O(\sqrt{n})$ where n is the curve order

For 256-bit curve ($n \approx 2^{256}$): $O(2^{128})$ operations — infeasible!

Standard Curves:

- **secp256k1:** Bitcoin, Ethereum
- **Curve25519:** Modern protocols (Signal, TLS 1.3)
- **P-256 (secp256r1):** NIST standard, widely deployed

ECDSA: Elliptic Curve Digital Signature Algorithm

Key Generation:

- Private key: random $d \in [1, n-1]$
- Public key: $Q = d \cdot G$ where G is the generator point

Signing Message m:

1. Hash message: $z = H(m)$ (e.g., SHA-256)
2. Choose random nonce: $k \in [1, n-1]$ (MUST be random and unique!)
3. Compute: $R = k \cdot G = (x_R, y_R)$
4. Set: $r = x_R \bmod n$ (if $r = 0$, retry with new k)
5. Compute: $s = k^{-1}(z + r \cdot d) \bmod n$ (if $s = 0$, retry)
6. Signature: (r, s)

Verification:

1. Compute: $w = s^{-1} \bmod n$
2. Compute: $u_1 = zw \bmod n, \quad u_2 = rw \bmod n$
3. Compute: $R' = u_1 \cdot G + u_2 \cdot Q$
4. Accept if $x_{R'} \equiv r \pmod{n}$

The PlayStation 3 Hack (2010)

Background: PS3 used ECDSA to sign game executables. Private key held by Sony.
Sony's Fatal Mistake: Used a **static** (constant) nonce k for every signature!
The Mathematics:
Two signatures of different messages:

$$s_1 = k^{-1}(z_1 + r \cdot d) \bmod n$$
$$s_2 = k^{-1}(z_2 + r \cdot d) \bmod n$$

Note: r is the same (since k is the same)!
Subtract the equations:

$$s_1 - s_2 = k^{-1}(z_1 - z_2) \bmod n$$

Solve for k:

$$k = \frac{z_1 - z_2}{s_1 - s_2} \bmod n$$

Once k is recovered, extract private key d from either signature:

$$d = r^{-1}(s \cdot k - z) \bmod n$$

Impact: Hackers could sign their own code as Sony. Complete security breach.

ECC Implementation Attacks

Side-Channel Attacks:

- **Timing Attacks:** Measure time taken for operations to deduce secret key bits
- **Power Analysis:** Monitor power consumption during scalar multiplication
- **EM Analysis:** Electromagnetic radiation leaks computation details

Invalid Curve Attacks:

Attacker provides point P that's *not* on the specified curve but on a weaker curve with smaller order. If implementation doesn't validate that P is on the curve, ECDLP becomes easier!

Mitigation Strategies:

- Constant-time implementations (no data-dependent branches)
- Point validation (always check $y^2 = x^3 + ax + b$)
- Use Montgomery ladder for scalar multiplication
- Blinding techniques (randomize internal computations)

Lesson: Mathematically secure $\neq$ Implementation secure!

Digital Signatures: Concept & Purpose

Goals:

- **Authentication:** Prove message came from claimed sender
- **Integrity:** Prove message hasn't been tampered with
- **Non-repudiation:** Signer cannot deny signing

RSA Signatures (Textbook):

- Sign: $S = H(M)^d \bmod N$ (encrypt hash with private key)
- Verify: Check if $S^e \equiv H(M) \pmod N$ (decrypt with public key)

Why Hash First?

- Message may be larger than modulus N
- Signing full message is slow
- Hash provides fixed-size digest (256-512 bits)

Common Schemes:

- RSA-PSS (Probabilistic Signature Scheme) — secure RSA signatures
- ECDSA — smaller signatures, faster verification
- EdDSA (Ed25519) — modern, deterministic, immune to nonce reuse

JWT (JSON Web Tokens) & Security

Structure: `Base64(Header).Base64(Payload).Base64(Signature)`
Example Header:

```
{"alg": "HS256", "typ": "JWT"}
```

Example Payload:

```
{"sub": "user123", "role": "admin", "exp": 1735689600}
```

Vulnerability 1: The 'none' Algorithm

1. Attacker decodes JWT, changes payload to `"role": "admin"`
2. Changes header: `{"alg": "none"}`
3. Removes signature entirely
4. Flawed backend code: `if (alg == "none") { return true; }`

Result: Complete authentication bypass!
Vulnerability 2: Algorithm Confusion (HS256 vs RS256)
Server expects RS256 (public key verification), but accepts HS256 (symmetric HMAC). Attacker signs with public key as HMAC secret!
Mitigation: Explicitly whitelist allowed algorithms, reject "none".

Heartbleed (CVE-2014-0160)

Context: OpenSSL's implementation of TLS Heartbeat extension.
Normal Operation:

- Client: "Echo this 3-byte payload: 'HAT'"
- Server: "HAT" (keeps connection alive)

The Attack:

- Client: "Echo this 64KB payload: 'A'" (declares 64KB but sends 1 byte!)
- Server: Doesn't validate claimed length vs. actual payload size

Vulnerable Code (simplified):

```
unsigned int payload_length = *(unsigned int*)request;
unsigned char *payload = request + 4;
unsigned char *response = malloc(payload_length);
memcpy(response, payload, payload_length); // NO BOUNDS CHECK!
send_to_client(response, payload_length);
```

Result: Server reads 64KB from heap memory (keys, session tokens, passwords, user data)!
Impact: Affected 17% of all HTTPS servers (2014). Yahoo, Amazon, many others vulnerable.
Fix: Check: `if (payload_length > actual_length) reject;`

POODLE: Padding Oracle On Downgraded Legacy

Target: SSLv3's CBC mode (2014)
The Flaw: SSLv3 doesn't verify padding bytes—only checks last byte is correct.
Attack Steps:

1. Downgrade TLS connection to SSLv3 (via protocol negotiation manipulation)
2. Intercept encrypted cookie/session
3. Use padding oracle to decrypt byte-by-byte
4. Each byte requires ≈ 256 attempts

Example: Steal session cookies from HTTPS connections.
Impact: SSLv3 was deprecated worldwide after disclosure.
Lesson: Never support outdated protocols "for compatibility"—attackers will force downgrades!

TLS 1.3: Modern Security

Key Improvements over TLS 1.2:

- **Removed weak algorithms:** RSA key exchange, SHA-1, RC4, 3DES, CBC mode
- **Forward secrecy mandatory:** Uses (EC)DHE for all connections
- **Faster handshake:** 1-RTT (round-trip time) instead of 2-RTT
- **0-RTT resumption:** Instant resumed connections (with replay protection caveats)
- **Encrypted handshake:** Hides certificate and extensions from eavesdroppers

Cipher Suites (TLS 1.3):

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256

All provide AEAD (Authenticated Encryption with Associated Data).
Key Exchange: ECDHE with X25519 or P-256 curves.

Zero-Knowledge Proofs (ZKP)

Goal: Prove knowledge of secret x without revealing any information about x.
Properties Required:

1. **Completeness:** If statement is true, honest verifier will be convinced
2. **Soundness:** If statement is false, cheating prover cannot convince verifier (except with negligible probability)
3. **Zero-Knowledge:** Verifier learns nothing except that statement is true

Example: Ali Baba's Cave

- Cave has two paths (A and B) meeting at magic door requiring password
- Prover enters cave via random path (A or B) out of verifier's sight
- Verifier randomly demands: "Exit via path A!" or "Exit via path B!"
- If prover knows password, can always comply (open door if needed)
- If prover doesn't know password, succeeds only 50% of time
- After 20 rounds: Probability of fake $= (1/2)^{20} \approx 10^{-6}$

Applications: Anonymous credentials (ZCash), authentication without passwords, private smart contracts.

ZKP: Schnorr Protocol (Discrete Log Proof)

Setup: Group G with generator g of order q. Public: $h = g^x$ where x is prover's secret.
Protocol: Prover convinces verifier they know x without revealing it.

1. **Commitment:** Prover picks random $r \in \mathbb{Z}_q$, computes $t = g^r$, sends t to verifier
2. **Challenge:** Verifier picks random $c \in \mathbb{Z}_q$, sends c to prover
3. **Response:** Prover computes $s = r + cx \bmod q$, sends s to verifier
4. **Verification:** Verifier checks if $g^s = t \cdot h^c$

Why it works:

$$g^s = g^{r+cx} = g^r \cdot g^{cx} = t \cdot (g^x)^c = t \cdot h^c \quad \checkmark$$

Zero-Knowledge: The transcript (t, c, s) can be simulated without knowing x (pick c, s first, compute $t = g^s h^{-c}$).
Soundness: Prover who doesn't know x cannot answer random challenges correctly.

Homomorphic Encryption

Concept: Perform computations on encrypted data without decrypting!

Partially Homomorphic Encryption (PHE):

- **RSA (multiplicative):** $E(m_1) \cdot E(m_2) = E(m_1 \cdot m_2)$
- **Paillier (additive):** $E(m_1) \cdot E(m_2) = E(m_1 + m_2)$
- Used for electronic voting (add up encrypted votes)

Fully Homomorphic Encryption (FHE):

- Discovered by Craig Gentry (2009) based on ideal lattices
- Supports *both* addition and multiplication
- Allows computing *any* function on encrypted data
- **Current State:** Computationally very expensive (1000x-10000x slower), but improving rapidly

Use Case: Send encrypted medical data to cloud, cloud computes diagnosis statistics without ever seeing the raw data, returns encrypted result.

The Quantum Threat

Shor's Algorithm (1994): A quantum algorithm that solves Integer Factorization and Discrete Logarithm in polynomial time.

The Apocalypse

If a sufficiently large Quantum Computer (CRQC) is built:

- **RSA:** Broken (Factoring is easy)
- **Diffie-Hellman / ECC:** Broken (DLP is easy)
- **Digital Signatures:** Forged instantly
- **HTTPS/TLS:** All historical traffic recorded today can be decrypted tomorrow ("Harvest Now, Decrypt Later")

Grover's Algorithm: Speeds up searching unstructured databases (brute force).

- **Impact on AES/SHA:** Effective key strength halved.
- **Mitigation:** Use AES-256 instead of AES-128. SHA-384 instead of SHA-256.

Post-Quantum Cryptography (PQC)

NIST standardization process (2016-2024) selected algorithms resistant to quantum attacks.
Main Approaches:

1. **Lattice-Based Cryptography:**
   - Based on "Learning With Errors" (LWE) or "Shortest Vector Problem" (SVP) in high-dimensional lattices.
   - **Selected Standards:** CRYSTALS-Kyber (KEM), CRYSTALS-Dilithium (Signatures).
2. **Hash-Based Signatures:**
   - Based on Merkle Trees and hash functions. Very conservative security.
   - **Selected Standard:** SPHINCS+.
3. **Code-Based Cryptography:**
   - Based on error-correcting codes (McEliece). Fast but huge keys.

The Future: "Hybrid" modes (Classical + PQC) will be used for decades to ensure security against both classical and potential quantum adversaries.

CTF Toolkit: The Essentials

When you encounter a crypto challenge, identifying the type is 90% of the battle.

Indicator	Probable Cipher	Tool / Library
Simple shifts, limited charset	Caesar / Vigenère	`dcode.fr`, `quipqiup.com`
Hex/Base64, repeating patterns	XOR	`xortool`, CyberChef
Small $e = 3$, huge N, "pubkey"	RSA	`RsaCtfTool`, `gmpy2`
"BEGIN PRIVATE KEY", signatures	ECC / DSA	`SageMath`, `openssl`
Padding errors, Oracle	AES-CBC / PKCS#7	`PadBuster`, `poracle`
Bitwise shifts, custom logic	Custom Stream	`Z3 Solver`
Random numbers predict	LCG / Mersenne	`randcrack` (Python)

Solving Custom Crypto with Z3

Many CTFs feature "home-rolled" crypto involving bit-shifts, XORs, and modular arithmetic. Reversing it manually is painful. Use a SMT Solver.

```python
from z3 import *

# 1. Initialize Solver
s = Solver()

# 2. Define Variables (e.g. 4 bytes of flag)
flag = [BitVec(f'b{i}', 8) for i in range(4)]

# 3. Add Constraints (The logic from the challenge)
# "The first byte XORed with 0x55 is 0x12"
s.add(flag[0] ^ 0x55 == 0x12)
# "The sum of all bytes is 400"
s.add(Sum(flag) == 400)
# "The third byte is double the first"
s.add(flag[2] == flag[0] << 1)

# 4. Check & Print
if s.check() == sat:
    m = s.model()
    # Convert result back to characters
    print("".join([chr(m[b].as_long()) for b in flag]))
```

Concept: You describe *what* the answer looks like mathematically, Z3 figures out *how* to find it.

Final Takeaways & Resources

The Golden Rules

1. **Don't Roll Your Own Crypto.** Use libsodium, Tink, or standard OS libraries.
2. **Math $\neq$ Code.** Implementation side-channels (timing, power, memory) kill secure math.
3. **Randomness is Everything.** Bad RNG = No Security.

Where to Learn More:

- **Practice:** CryptoHack.org (Highly Recommended), CryptoPals.com
- **Read:** "Serious Cryptography" (Aumasson), "Real-World Cryptography" (Wong)
- **Tools:** CyberChef, SageMath

Questions?

`flag{crypto_is_math_with_consequences}`