Secure File Exchange System: A Hybrid Cryptosystem

Final Project Presentation

Authentication

Rabin Signature Scheme

Confidentiality

Blowfish (CFB Mode)

Key Exchange

Elliptic Curve ElGamal

Yuval Lerfeld
Avishag Levi
Einav Momi Ben Shushan

GitHub

The Core Security Challenge: Balancing Speed and Trust

The Goal: The CIA Triad

Confidentiality

Ensuring data is accessible only to authorized individuals.

Integrity

Protecting data from unauthorized modification.

Authenticity

Verifying the identity of the sender and the origin of the data.

The Engineering Dilemma

Symmetric Encryption (e.g., Blowfish, AES)

Pro: Extremely fast and efficient for bulk data.

Con: Key distribution is a major security risk. How do you share the secret key securely in the first place?

Asymmetric Encryption (e.g., RSA, ECC)

Pro: Solves the key distribution problem with a public/private key pair.

Con: As stated in cryptographic literature, "RSA is much slower than symmetric cryptosystems." This makes it impractical for encrypting large files directly.

Key Takeaway: Neither system is a complete solution on its own. The optimal approach must combine the strengths of both.

Our Solution: A Three-Step Hybrid Protocol

SIGN

Alice signs the file's hash using her **Rabin private key** to ensure Authenticity & Integrity.

ENCRYPT

Alice encrypts the file with a random, one-time **Blowfish session key** for Confidentiality.

EXCHANGE KEY

Alice encrypts the Blowfish session key using Bob's **EC-ElGamal public key** for secure key exchange.

[Original File] → [Rabin Signing] → [Blowfish Encryption] → [EC-ElGamal Key Wrap] → [Secure Payload]

Step 1: Authentication via Rabin Signatures

The Mathematics of Provable Keys

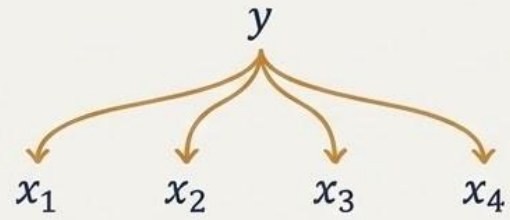
Core Concept:

The security of the Rabin cryptosystem is based on the difficulty of finding square roots modulo a composite number n, which is as hard as factoring n itself.

Blum Integers:

To guarantee the existence of square roots, we use Blum integers for our modulus n, where the prime factors p and q are both congruent to 3 (mod 4).

Key Insight: The "Four Roots" Problem



$y \equiv x^2 \pmod{n}$ in Fira Code

Weakness for Encryption: This ambiguity is a problem for decryption (which of the four plaintexts is correct?).

Strength for Signing: The ability to produce *any* valid square root of a hash $H(m)$ serves as definitive proof that you know the secret factors p and q. The signature is simply one of these roots.

Rabin Implementation: The Search for a Signature

The Challenge: Not every number has a square root modulo n. In fact, only about 25% of possible message hashes will be "quadratic residues" (i.e., have a square root).

The Solution: Probabilistic Padding:

- We cannot change the file hash $H(m)$.
- Instead, we append a small, incremental counter or nonce (i) to the hash before signing: $H(m \,||\, i)$.
- We iterate this counter $i = 0, 1, 2, \ldots$ until we find a hash that is a quadratic residue modulo n.

```
def sign(message, private_key):
    p, q = private_key
    n = p * q

    nonce = 0
    while True:
        # Append nonce and re-hash
        padded_hash = hash_function(message + bytes(nonce))

        # Attempt to find a square root (signature)
        signature = find_sqrt_mod_n(padded_hash, p, q)

        if signature is not None:
            # Success! Return the signature and the nonce used.
            return signature, nonce

        # Failure, increment nonce and try again.
        nonce += 1
```

The signature consists of both the root S and the nonce i that made it possible. Verification requires checking that $S^2 \equiv H(m \,||\, i) \pmod{n}$.

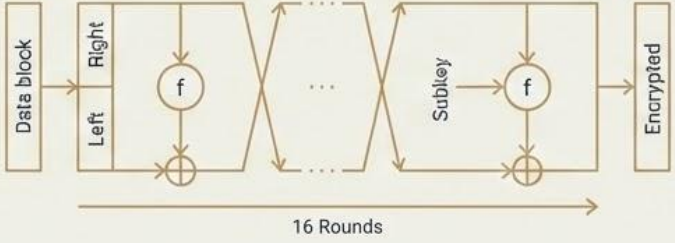
Step 2: Confidentiality via Blowfish Encryption

What is Blowfish?

Blowfish is a **64-bit block cipher** that uses 16 rounds of encryption. It processes data by splitting it into two halves. Unlike older algorithms (like DES), it uses **dynamic S-Boxes** generated from the key, making it highly resistant to attacks.

Core Architecture

- **Structure:** 16-round Feistel Network. The Feistel structure is a classic, well-analyzed design template for block ciphers.
- **Block Size:** 64-bit blocks.
- **Key Size:** Variable, from 32 bits up to 448 bits.



Blowfish Advantages

- **Speed** Designed for high speed on standard microprocessors. It outperforms legacy algorithms (like DES) without requiring specialized hardware.
- **Compactness** Features a minimal memory footprint (requires less than 5KB), making it highly suitable for resource-constrained environments.
- **Security** Utilizes Key-Dependent Dynamic S-Boxes and digits of π ("Nothing up my sleeve" numbers). This structure eliminates backdoors and prevents pre-computation attacks.

Blowfish: The Process

Blowfish is a **64-bit Block Cipher.** It divides data into fixed 64-bit blocks and encrypts each one individually.

The Process:

1. Splitting:

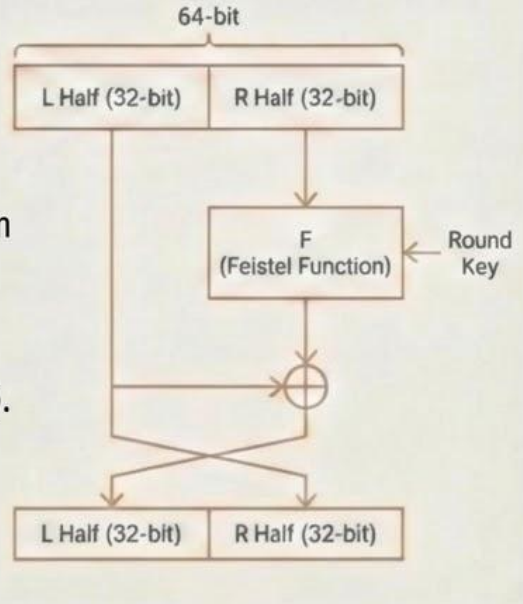
- The 64-bit input block is split into two equal 32-bit halves.

2. The Core (16 Feistel Rounds):

- **The F-Function:** Splits the 32 bits into 4 bytes, applies S-Box lookups, and mixes them using alternating XOR and Addition (+).
- **XOR & Swap:** The result is XORed with the other half, and then the two halves are swapped.
- Result: Even the smallest change in input impacts the entire block ("Avalanche Effect").

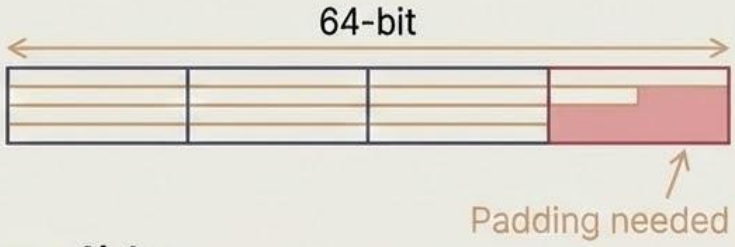
3. The Brain (Dynamic S-Boxes):

- The mixing logic is managed by the **Substitution Boxes.**
- They start as mathematical constants but are **mixed** with the **User's Key.**
- *Conclusion:* Every key creates a unique encryption machine.



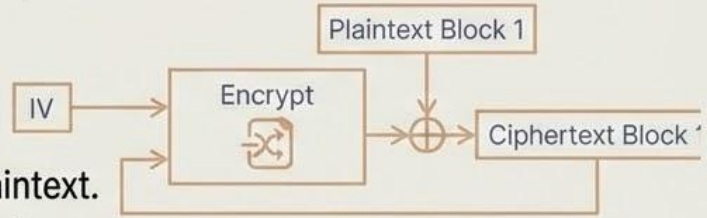
Mode of Operation: Cipher Feedback (CFB) for File Streaming

The Challenge:



- Blowfish operates on fixed 64-bit blocks.
- **Problem:** Most files don't fit perfectly into 64-bit chunks. Standard algorithms require **Padding** (adding dummy data), which is inefficient.
- **Solution:** We use **CFB Mode** to transform Blowfish into a **Stream Cipher.**

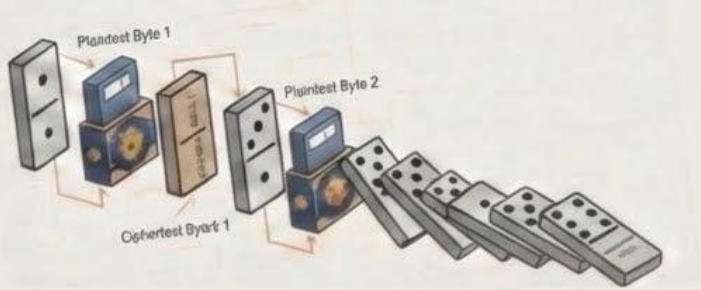
The Process (Step-by-step):



1. The Initialization Vector **(IV)** is encrypted by the Blowfish engine.
2. We take the first byte of the result **XOR** with the first byte of the plaintext.
3. The result encrypted byte is shifted **into the IV**, pushing old data out.

Repeat: This updated IV is used to encrypt the next byte.

The Result: "The Domino Effect"



- Each encrypted byte depends on the previous one.
- Changes in one byte propagate through the entire chain.

Blowfish Implementation: The CFB Encryption Loop

While standard algorithms usually require a specific *Decryption* Function to reverse the process mathematically.

With the CFB method we do not use a decrypt function. Instead, we use the *Encrypt* Function again.

The "Cool Trick" (XOR Symmetry):

- **The Logic:** XOR is its own inverse ($A \oplus B = C \rightarrow C \oplus B = A$).
- **The Goal:** We don't need to "unlock" the Blowfish engine we just need to recreate the exact same "Noise" (Keystream).

The Process:

- We run the *Encrypt* function on the IV to regenerate the noise.
- We XOR the noise with the Ciphertext.
- **Result:** The original *Plaintext* is revealed.

```python
# The high-level library call:
def encrypt_cfb(plaintext, key, iv):
    blowfish_cipher = blowfish.now(key, Blowfish.MODE_CFB, iv)
    ciphertext = blowfish_cipher.encrypt(plaintext)
    return ciphertext

# Under the hood, the library performs this logic:
def conceptual_cfb_loop(plaintext_bytes, key, iv):
    ciphertext_bytes = bytearray()
    previous_block = iv

    for block in chunk(plaintext_bytes, 8): # Process in 8-byte chunks
        # 1. Encrypt the previous block (or IV)
        keystream = blowfish_encrypt_block(previous_block, key)

        # 2. XOR with the current plaintext block
        encrypted_block = xor(block, keystream)

        ciphertext_bytes.extend(encrypted_block)

        # 3. The new ciphertext becomes the next input
        previous_block = encrypted_block

    return bytes(ciphertext_bytes)
```

Step 3: Key Exchange via Elliptic Curve ElGamal

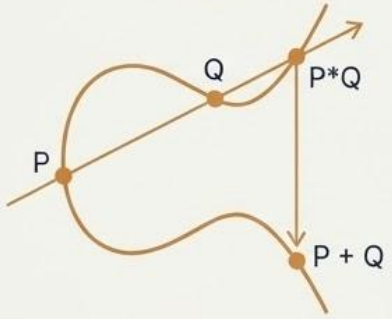
The Geometric Foundation of Modern Cryptography

- **The Curve:** We use **secp256k1**, the curve famous for its use in Bitcoin.

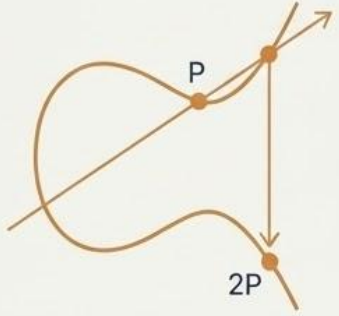
$$y^2 \equiv x^3 + 7 \pmod{p}$$

- **The 'Hard Problem':** The security of ECC is based on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP). Given points P and Q, it is computationally infeasible to find the integer k such that $Q = k * P$.

Visualizing the Math



(Point Addition)



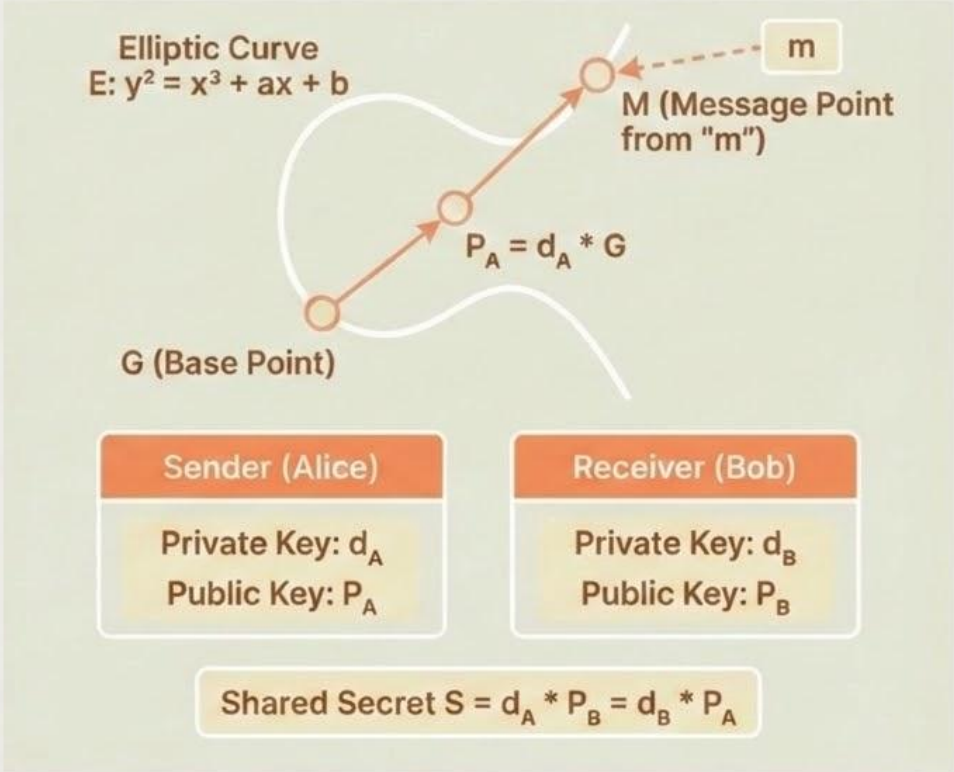
(Point Doubling)

EC Implementation: Point Operations in Code

From Geometry to Algebra: The geometric rules for point addition are translated into algebraic formulas. The core of this is calculating the slope *m* of the line.

The Challenge: Division in a Finite Field:

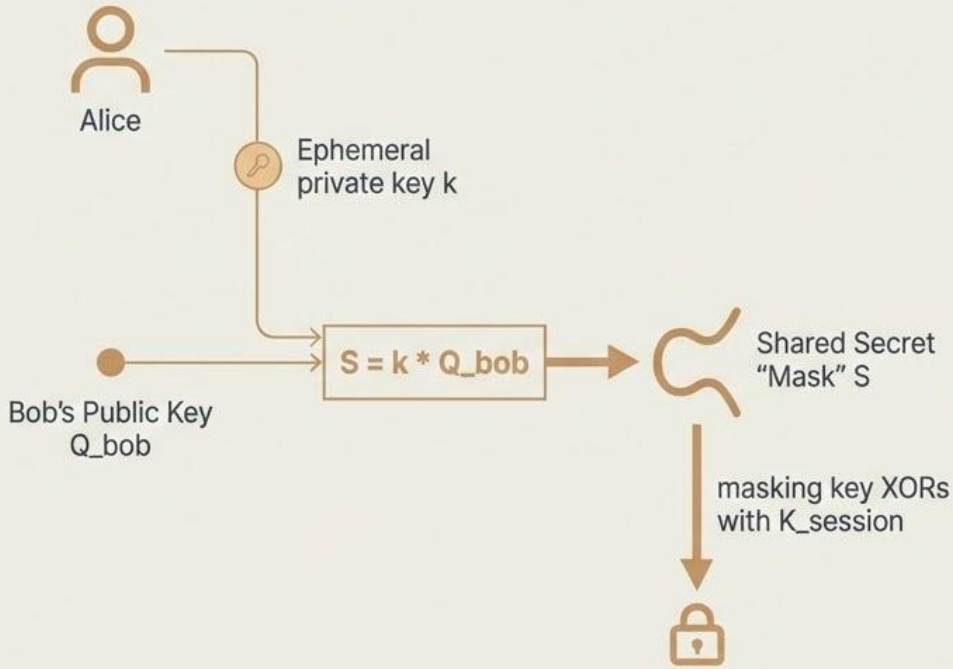
- The formula for the slope *m* involves division (e.g., m = (y2 – y1) / (x2 – x1)).
- In a finite field (modulo *p*), division is not directly defined.
- We achieve division by multiplying by the modular **multiplicative inverse**. That is, a / b becomes a * mod_inverse(b, p).



The EC-ElGamal Protocol: Masking the Secret Key

Goal: To encrypt the short Blowfish session key (K_session) using Bob's public key (Q_bob) so that only Bob can decrypt it.

1. **Generate Ephemeral Key**
   Alice creates a temporary, single-use random number k.
2. **Compute Public 'Hint' (R)**
   Alice computes R = k * G, where G is the public base point. She sends R to Bob.
3. **Compute Shared Secret 'Mask' (S)**
   Alice computes S = k * Q_bob, a point on the curve that only she and Bob can compute.
   - Bob re-computes it as S = d_bob * R.
4. **Encrypt**
   Alice uses the x-coordinate of S to derive a masking key and XORs it with K_session.



EL-GAMAL Decryption

Bob needs to recover the Symmetric Session Key to decrypt the actual message.

The Inputs:

1. **Bob's Private Key (d):** Kept secret (known only to Bob).
2. **Received Ephemeral Point (R):** Publicly sent by Alice.

The Mathematical Magic:

- Bob computes the shared secret **(S)** using the formula:

$$S = d \times R$$

Why it works:

- Alice calculated: **S = k X Q** (Random k X Bob's Public Key).
- Bob calculates: **S = d X R** (Bob's Private Key X Alice's Random Point).
- Result: Due to Elliptic Curve properties, both result in the exact same point **S.**

Final Step: Bob uses point **S** to "unmask" the encrypted session key, revealing the original Blowfish Key.

The Full Protocol, Phase 1: Key Generation & Setup

Alice (Sender)

Action: Generates a Rabin key pair.

Private Key: Two large Blum primes, p and q. (Secret)

Public Key: The modulus n = p * q. (Shared)

Bob (Receiver)

Action: Generates an Elliptic Curve key pair.

Private Key: A random integer d_bob. (Secret)
Public Key: A point on the curve Q_bob = d_bob * G. (Shared)

State at End of Phase: Both parties have established their public identities and are ready to communicate.

The Full Protocol, Phase 2: Sender Workflow (Alice)

Input: A file to be sent and a generated symmetric key (Blowfish_Key).

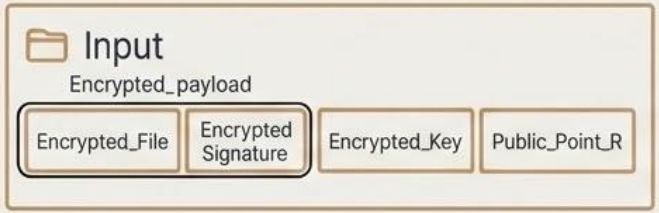
Step-by-Step Actions:

1. **Hash & Sign:**
   - Alice cocryptes H = Hash(File || nonce).
   - She finds a square root of H using her private p and q.
   - **Output:** Signature = (**S**, nonce)
2. **Encrypt File:**
   - Alice encrypts the package (File || **S** || nonce) using the Blowfish_Key in CFB mode.
   - **Output: Encrypted_package**
3. **Encapsulate Key:**
   - Alice uses Bob's public key Q_bob to encrypt her Blowfish_Key.
   - Output: **Encrypted_Key** and **Public_Point_R**

Delivery package (To Bob):

1. BlowFish output: Encrypted_package
2. ELGamal output: Encrypted_Key & Public_Point_R
3. IV vector.

The Full Protocol, Phase 3: Receiver Workflow (Bob)



Step-by-Step Actions (in reverse order of encryption):

1. **"Decapsulate Session Key":** Bob uses his private key d_bob and the public point R to compute the shared secret S = d_bob * R. He derives the same masking key hash(S.x) and XORs it with Encrypted_Key to recover the original Blowfish_Key.
2. **"Decrypt File":** Bob uses the now-recovered Blowfish_Key to decrypt Encrypted_payload back into Original Data.
3. **"Verify Signature"** Bob computes `H = Hash(Plaintext_File || nonce)`. He uses Alice's public key n to check if $s^2 \equiv H \pmod{n}$.

Outcome: If the signature is valid, Bob is assured of **Confidentiality** (only he could decrypt the key), **Integrity** (the hash matches), and **Authenticity** (the signature is valid).

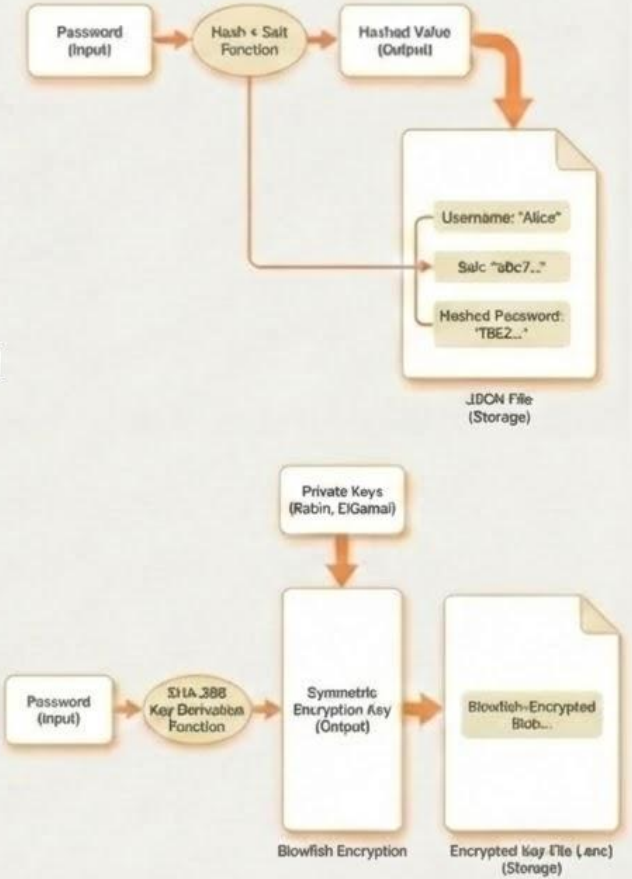
User Authentication & Key Management

1. **System Architecture:** Simulated Client-Server model using local JSON databases for users and messages.

2. **Secure Registration:**
   - Passwords protected via **SHA-256 Hashing + Unique Salt** (defense against Rainbow Tables).
   - Automatic generation of **Rabin** (Signing) and **EC-ElGamal** (Encryption) key pairs.

3. **"Encryption at Rest" (Key Protection):** Private keys are never stored in plaintext. They are encrypted on the disk using **Blowfish**, with a key derived directly from the user's password.

4. **Zero-Knowledge Storage:** The server stores only **Public Keys**. Private keys remain local and encrypted, accessible only during an active session.



Comparative Analysis: Justifying Our Choices

Security Strength per Bit (ECC vs. RSA)

The primary advantage of ECC is providing equivalent security with significantly smaller key sizes.

Symmetric Key Security (bits)	Required RSA Key Size (bits)	Required ECC Key Size (bits)
80	1024	160
192	7680	384
128 (Our Target)	**3072**	**256**
256	15360	512

Conclusion: To achieve 128-bit equivalent security, ECC requires only a 256-bit key, while RSA would need a 3072-bit key. This leads to faster computations and smaller signatures.

Performance of Blowfish

Key Point: While AES is the modern standard, Blowfish's design makes it extremely performant in software on 32-bit and 64-bit processors, a key consideration for this project's implementation.

Conclusion

Core Strategy: We implemented a “Sign then Encrypt” workflow to ensure maximum security.

Step 1: Rabin Signature (Authenticity & Integrity)

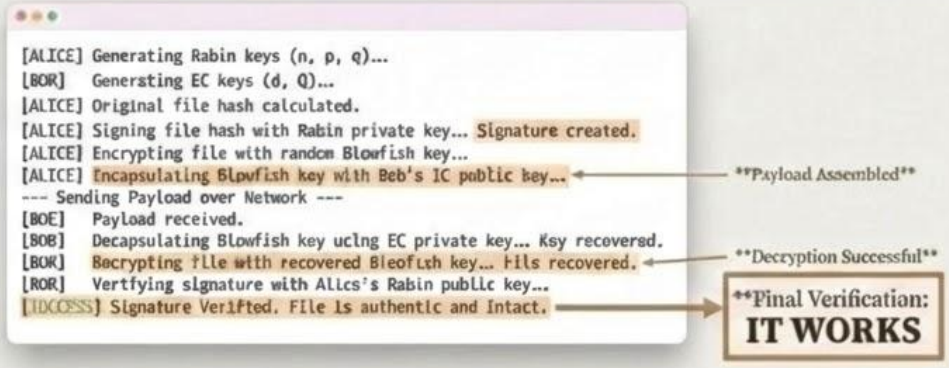
- Action: We first generate a digital signature for the plaintext.
- Goal: Ensures the message originates from the sender and hasn’t been modified.

Step 2: Blowfish (Confidentiality)

- Action: We encrypt both the message and the Signature together.
- Tech: Uses CFB Mode for efficient streaming.

Key Management: EC-ElGamal

- Action: Securely exchanges the session key.
- Tech: Uses Elliptic Curve cryptography to protect the Blowfish key.



Thank you for listening

Any Questions?