UFUG2106 Crypto Project

Zebin Chen
HKUST-GZ
zchen892@connect.hkust-gz.edu.cn

Zishen Lai
HKUST-GZ
zlai012@connect.hkust-gz.edu.cn

Tianyu Guo
HKUST-GZ
tguo409@connect.hkust-gz.edu.cn

Xinhao Zhao
HKUST-GZ
xzhao112@connect.hkust-gz.edu.cn

1 Motivation

This report's goal is straightforward: to implement, from scratch in Python, two classic public-key encryption schemes—RSA and ElGamal—and to perform a simple comparison of their correctness, efficiency, and security intuitions. Our motivations are:

1. **Turn abstract formulas into working code.** By writing our own prime-generation routines and modular exponentiation functions, we reinforce the foundations of big-integer arithmetic and number theory, and see exactly how textbook notation maps to code.
2. **Compare all two schemes on the same experimental platform.** Testing both short and long plaintexts lets us observe firsthand the trade-offs between computation time and ciphertext/key sizes in RSA versus ElGamal.
3. **Lay the groundwork for deeper study.** Gaining practical experience with classic algorithms—learning where random-number quality matters, how parameter choices affect correctness, and what limits plaintext size—prepares us to explore more advanced encryption techniques or contribute to real-world security protocols.

In summary, by hands-on implementation, lightweight benchmarking, and concise analysis, this project not only satisfies the course requirement to "implement and compare RSA/ElGamal" but also gives us an accessible introduction to ECC-ElGamal, establishing a solid, practical starting point for further cryptographic study.

2 Related Work

In this project, we've encapsulated three public-key encryption schemes—RSA, ElGamal over prime fields, and ECC-ElGamal—into a single Python package that exposes a uniform API for key generation, encryption, and decryption. It can be checked on Crypto-Project on GitHub.We also introduce the system design 3 of our project.

Beyond simply coding up each algorithm, we documented the underlying number-theoretic and algebraic principles—integer factorization for RSA, discrete logarithms in $\mathbb{F}_p^\times$ for ElGamal, and the elliptic-curve discrete logarithm for ECC-ElGamal. To evaluate practical performance, we benchmarked key generation, encryption, and decryption across a range of key sizes, using Python's

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

timeit module to measure average runtimes. And security was assessed by estimating equivalent security levels. 4

Finally, we identified several avenues for future enhancement. 5.2

3 System Design

Architecture 1

- The repository is divided into three layers to enforce clear separation of concerns. 2
- **Interface Layer** (`README.md`, `benchmarks/`): exposes installation instructions, usage examples and performance scripts; decouples user entry points from implementation details.
- **Core Logic Layer** (`crypto/`): contains all cryptographic algorithms and shared utilities; each file handles a single responsibility, making extension and maintenance straightforward.
- **Quality Assurance Layer** (`tests/`): holds pytest modules that seed their RNGs for reproducible checks; verifies correctness without running heavy benchmarks in CI.

```
crypto
   __init__.py
   elgamal.py
   ecc_elgamal.py
   primes.py
   rsa.py
   utils.py
tests
   __init__.py
   test_elgamal.py
   test_ecc_elgamal.py
   test_rsa.py
benchmarks
    benchmark.py
README.md
requirements.txt
```

Figure 1: Project Directory Structure

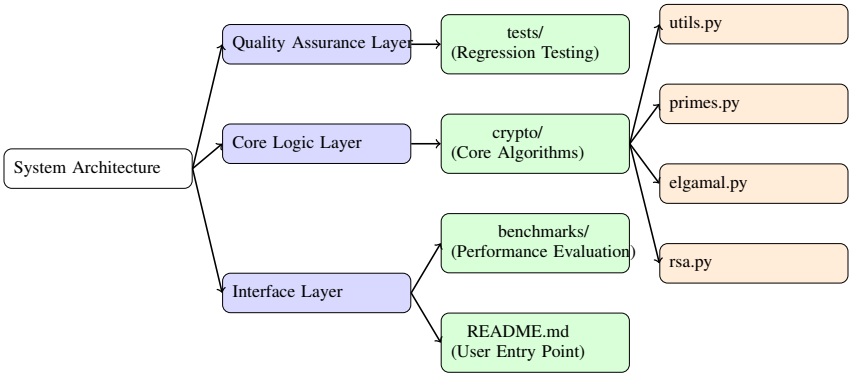

Figure 2: System architecture of the cryptographic library

Modules

- `rsa.py`: parameterized `keygen()`, `encrypt()`, `decrypt()` with optional OAEP/PKCS#1 padding.
- `elgamal.py`: analogous API using a fresh random k per encryption and pluggable group parameters.
- `primes.py`: Miller–Rabin primality test and safe-prime generation, decoupled from cipher logic.
- `utils.py`: number-theoretic helpers (modular inverse, GCD) and integer-to-byte encoding utilities.

- `__init__.py`: re-exports RSA and ElGamal classes under a unified public API, hiding internal complexity.
- `benchmark.py`: accepts CLI parameters (e.g. `-algo rsa -size 2048`) to output throughput and latency metrics.
- `test_rsa.py`, `test_elgamal.py`: seed RNGs for reproducibility, check encryption/decryption correctness and optional homomorphic properties.

Key Design Choices

- **Dedicated utility modules**: factoring out prime generation and math routines minimizes duplication and enables future acceleration with libraries like `gmpy2` or `numba`.
- **Separation of tests and benchmarks**: ensures CI runs fast correctness checks while performance evaluations can run offline or on dedicated hardware.
- **Parameterization**: key sizes, group parameters and RNG sources are configurable to support research experiments and compliance scenarios (e.g. FIPS-approved RNGs).

4 Method Comparison and Results

4.1 RSA

1. **Key Generation**
   - Choose two large primes p and q.
   - Compute
   $$n = p \times q$$
   and
   $$\varphi(n) = (p-1), (q-1), .$$
   - Pick an integer e with $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
   - Compute d such that
   $$e, d \equiv 1 \pmod{\varphi(n)}, .$$
   - Public key: (n, e). Private key: (n, d).
2. **Encryption** For message $M < n$, compute
   $$C = M^e \bmod n, .$$
3. **Decryption** Given ciphertext C, compute
   $$M = C^d \bmod n$$
   to recover the plaintext.
4. **Security** Based on the difficulty of factoring the large integer n.

4.2 ElGamal

1. **Parameter and Key Generation**
   - Choose a large prime p and a generator g of the multiplicative group $\mathbb{Z}_p^\times$.
   - Select a private key
   $$x \xleftarrow{\$} \{1, 2, \ldots, p-2\}.$$
   - Compute the public component
   $$y \equiv g^x \pmod{p}.$$
   - The public key is (p, g, y); the private key is x.
2. **Encryption**
   To encrypt a message M with $0 \leq M < p$:
   $$k \xleftarrow{\$} \{1, 2, \ldots, p-2\}, \quad a \equiv g^k \pmod{p}, \quad b \equiv M \cdot y^k \pmod{p}.$$
   Output the ciphertext (a, b).

3. **Decryption**
   Given ciphertext (a, b), recover

$$M \equiv b \cdot a^{-x} \pmod{p}.$$

   (Note $a^{-x} \equiv a^{p-1-x} \pmod{p}$ by Fermat's little theorem.)

4. **Security**
   Relies on the hardness of the discrete logarithm (or Diffie–Hellman) problem in $\mathbb{Z}_p^\times$.

4.3 Elliptic-Curve ElGamal

1. **Parameter and Key Generation**
   - Choose an elliptic curve

$$E : y^2 = x^3 + a\,x + b$$

     defined over the finite field $\mathbb{F}_q$, and fix a base point $G \in E(\mathbb{F}_q)$ of prime order n.
     (Our implementation employs the standardized elliptic curve **secp256k1**, defined over a 256-bit prime field. Despite its compact key size, **secp256k1** provides strong cryptographic security and is widely used in real-world applications such as cryptocurrencies.)
   - Select a private key

$$d \xleftarrow{\$} \{1, 2, \ldots, n-1\}.$$

   - Compute the public key

$$Q = d\,G,$$

     where $d\,G$ denotes multiplication defined for elliptic curve.
   - The public key is $(E,\ G,\ Q)$; the private key is d.

2. **Encryption**
   To encrypt a binary message $m \in \{0,1\}^*$:

$$k \xleftarrow{\$} \{1, 2, \ldots, n-1\}, \quad R = k\,G, \quad S = k\,Q, \quad K = \mathrm{KDF}(S_x), \quad c = m \oplus K.$$

   Output the ciphertext (R, c).

3. **Decryption**
   Given (R, c), recover the message by:

$$S = d\,R, \quad K = \mathrm{KDF}(S_x), \quad m = c \oplus K,$$

   since $S = d\,R = d\,(kG) = k\,(dG) = k\,Q$.

4. **Security**
   The security level of elliptic-curve ElGamal cryptographic schemes primarily depends on the choice of the curve and the size of the keys. Owing to the computational hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP), elliptic-curve-based algorithms can achieve equivalent security to classical ElGamal encryption while requiring significantly smaller key sizes. In particular, the curve used in our implementation—**secp256k1**—provides a security level comparable to that of a 3072-bit key in conventional ElGamal, using only a 256-bit key.

4.4 Performance Comparison

In this section, we compare the computation time of the algorithm we implement. To evaluate the computational efficiency of RSA, ElGamal, and ECC-ElGamal algorithms, we conducted five repeated experiments for each algorithm under different key sizes, testing their time expense on key generation, encryption, and decryption. Additionally, to isolate the impact of prime generation on algorithm complexity, we performed a fixed-prime experiment for ElGamal, removing the random prime generation step.

Algorithm	Bit Size	Key Generation Time (s)	Encryption Time (s)	Decryption Time (s)
RSA	512	3.332740	0.0107277	0.233534
RSA	768	11.5873	0.00974636	0.370135
RSA	1024	41.0410	0.0133873	0.524911
ElGamal	512	17.5110	0.00312467	0.00707569
ElGamal	768	152.145	0.00739932	0.0152948
ElGamal	1024	288.341	0.0163398	0.0188498

Table 1: Performance comparison of RSA and ElGamal at different key sizes

4.4.1 RSA vs. ElGamal

We compared the computation time of RSA algorithm and ElGamal algorithm under 3 different bit sizes (512, 768, 1024), the results are shown in Table 1

A significant portion of the overall computation time is attributed to the generation of large random prime numbers. This process typically involves repeatedly generating random odd integers of the specified bit length and testing their primality until a suitable prime is found. In the case of RSA, two primes are required, each approximately half the total key size, which results in relatively faster key generation compared to ElGamal. However, RSA tends to exhibit slower performance during encryption and decryption operations.

It is also worth noting that ElGamal allows the reuse of its key pair across multiple encryption sessions, whereas RSA generally requires a new key pair for each secure interaction when ephemeral security is desired. As a result, in scenarios involving frequent message encryption and decryption, ElGamal may achieve significantly better overall computational efficiency.

4.4.2 ElGamal vs. ECC-ElGamal

As ECC-ElGamal use a published elliptical curve parameter (it's too hard to generate one by ourselves), in this section we only compare the computation performance between conventional ElGamal algorithm (using a published 3072-bits prime and the corresponding generator, from RFC 3526 Section 6- Group 15) and ECC-ElGamal (using elliptical curve "secp256k1"). The results are shown in Table 2:

Algorithm	Bit Size	Key Generation Time (s)	Encryption Time (s)	Decryption Time (s)
ECC	256	0.0990836	0.183243	0.107889
ElGamal	3072	0.2891500	0.292014	0.287438

Table 2: Performance comparison between ECC and traditional ElGamal

Due to the smaller bit size when achieving the similar security level, the computation time of ECC-ElGamal is significantly less than conventional ElGamal

4.5 Security Comparison

RSA: RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem whose security is based on the difficulty of factoring large integers. Public-key operations (encryption and signature verification) are relatively fast, while private-key operations (decryption and signing) are comparatively slower.

ElGamal: The ElGamal encryption scheme operates over the multiplicative group of a prime field $\mathbb{F}_p^\times$, and its security relies on the discrete logarithm problem in that group. Unlike RSA, ElGamal is inherently randomized: encrypting the same plaintext twice yields different ciphertexts, providing native IND-CPA security. Because both keys and ciphertexts consist of group elements, their sizes are roughly twice those of RSA at equivalent security levels.

ECC-ElGamal ECC-ElGamal adapts ElGamal encryption to an elliptic-curve group, with security based on the elliptic-curve discrete logarithm problem (ECDLP). Elliptic curves achieve comparable security with much shorter key lengths, making ECC-ElGamal well suited for resource-constrained or bandwidth-limited environments.

Comparison

Item	RSA	ElGamal (Finite Field)	ECC-ElGamal
Security Assumption	Integer factorization problem	Discrete logarithm problem	Elliptic-curve discrete logarithm problem
Symmetric Security Strength	1024 bit $\approx$ 80 bit 2048 bit $\approx$ 112 bit 3072 bit $\approx$ 128 bit	1024 bit $\approx$ 80 bit 2048 bit $\approx$ 112 bit 3072 bit $\approx$ 128 bit	160 bit $\approx$ 80 bit 224 bit $\approx$ 112 bit 256 bit $\approx$ 128 bit
Typical Recommended Key Length	3072 bit (128-bit security)	3072 bit (128-bit security)	256 bit (128-bit security)
Quantum Resistance	Not quantum-safe (Shor's algorithm breaks)	Not quantum-safe (Shor's algorithm breaks)	Not quantum-safe (Shor's algorithm breaks)
Algorithmic Complexity	Keygen: $O(k^4)$ Encrypt/Decrypt: $O(k^3)$	Keygen: $O(k^4)$ Encrypt/Decrypt: $O(k^3)$	Keygen/Enc/Dec: $O(k^2)$
Performance	Fast encrypt/verify, slow decrypt/sign; heavy big-int ops	Two exponentiations per encrypt/decrypt, slightly slower	ECC point-multiply ops—faster, lower resource consumption
Key/Public-Key Size	Private $\approx$ 3072 bit Public $\approx$ 3072 bit	Private $\approx$ 3072 bit Public $\approx$ 3072 bit	Private $\approx$ 256 bit Public $\approx$ 512 bit (two coords)
Ciphertext/Signature Size	$\approx$ 3072 bit	$\approx 2 \times 3072$ bit	$\approx 2 \times 256$ bit
Implementation Maturity	Extremely mature, widely supported	Mature but less common than RSA	Mainstream in resource-constrained contexts (IoT, mobile)

Table 3: Comparison of RSA, ElGamal, and ECC-ElGamal

We make a analysis table in detailed 4.5 and here is some takeaways.

RSA, ElGamal and ECC-ElGamal differ in their security foundations, performance and typical use cases. RSA relies on integer factorization, giving fast encryption and verification but slower decryption and signing. ElGamal, based on the finite-field discrete logarithm, needs two exponentiations per operation, making it slightly slower. ECC-ElGamal uses elliptic-curve discrete logs and 256-bit keys; its point-multiplication operations are fast and low-resource, ideal for mobile and IoT.

At a 128-bit security level, RSA and ElGamal require 3072-bit keys, whereas ECC-ElGamal needs only 256 bits, cutting bandwidth and storage. None resist Shor's algorithm, so true quantum safety demands post-quantum schemes. RSA is the most mature with extensive library and hardware support; ElGamal is less common; ECC-ElGamal is increasingly preferred in constrained environments.

In practice, choose RSA for maximum compatibility and minimal migration cost; ElGamal if you need classical discrete-log features; and ECC-ElGamal for new deployments requiring high performance, low bandwidth and small keys.

5 Conclusion and Future Work

5.1 Conclusion

In this project, we implemented and analyzed two public-key encryption schemes: RSA, ElGamal over finite fields, and further evaluate elliptic-curve-based ElGamal (ECC-ElGamal). Through both theoretical comparison and empirical performance testing, we gained a comprehensive understanding of their structural differences, computational characteristics, and practical security levels.

Our implementation of ECC-ElGamal leverages the standardized `secp256k1` curve, demonstrating that elliptic curve cryptography can achieve the same level of security as traditional methods with significantly smaller key sizes and reduced computational overhead. The experimental results validate that ECC-ElGamal outperforms classical ElGamal and RSA in terms of efficiency, especially in key generation and decryption, while maintaining equivalent security guarantees.

While RSA remains the most mature and widely supported scheme, our study confirms that ECC-ElGamal is more suitable for modern applications where bandwidth, storage, and computational resources are limited—such as mobile systems and IoT environments.

Overall, this project not only strengthened our understanding of public-key cryptography but also highlighted the trade-offs between classical and modern approaches. ECC-based schemes are likely to become the preferred choice for future cryptographic deployments in performance-sensitive contexts.

5.2 Future Work

Although our implementations of RSA, ElGamal, and ECC–ElGamal successfully demonstrate basic correctness and performance trade-offs, there remain several avenues for improvement and expansion:

1. **Strengthening Security Measures**
   - Enhance input validation and error handling to detect malformed or malicious inputs, providing clear failure messages rather than silent crashes.
   - Integrate message authentication codes (MACs) or digital signatures to guarantee ciphertext integrity and protect against active tampering.
2. **Supporting Higher Security Parameters and Additional Algorithms**
   - Evaluate and enable larger key sizes and curve parameters to meet evolving security standards.
   - Extend the library to include other cryptographic primitives, such as elliptic-curve digital signature algorithms (ECDSA) or hybrid encryption schemes combining asymmetric key exchange with symmetric data encryption.
3. **Performance Optimization**
   - Profile and optimize the core arithmetic routines (prime generation, modular exponentiation, elliptic-curve scalar multiplication) for large datasets and batch encryption scenarios.
   - Explore parallelization and hardware acceleration (GPU) to reduce latency in high-throughput environments.
4. **Robust Testing and Continuous Integration**
   - Augment unit tests and integration tests to cover edge cases—such as boundary key sizes, invalid curve points, and extreme plaintext lengths—to improve reliability.
   - Set up a continuous integration pipeline that automatically runs tests, enforces style guidelines, and scans for known vulnerabilities on each commit.

Credit

1. **System Design and Documentation**: ZebinChen
2. **RSA Algorithm Module Development and Testing**: TianyuGuo
3. **ElGamal Algorithm Module Development and Testing**: ZebinChen
4. **New Algorithm (ECC–ElGamal) Module Development and Testing**: ZishenLai
5. **Performance Benchmark Analysis**: XinhaoZhao
6. **Security Analysis**: ZebinChen
7. **Report Writing**: ZebinChen, ZishenLai, XinhaoZhao, TianyuGuo
8. **Code Organization and Project Documentation**: ZebinChen