НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

Звіт до лабораторної №3

за темою:
Реалізація основних асиметричних криптосистем з застосуванням OpenSSL C++ на Windows платформі

Оформлення звіту:
Юрчук Олексій, ФІ-52мн

October 19, 2025
м. Київ

ЗМІСТ

1 Introduction

Public-key cryptography revolutionized secure communications by solving the key distribution problem that plagued symmetric cryptosystems. Among the fundamental asymmetric cryptosystems, the ElGamal cryptosystem, proposed by Taher ElGamal in 1985 [1], stands as one of the most important and widely studied schemes. Unlike RSA, which relies on the integer factorization problem, ElGamal's security is based on the discrete logarithm problem (DLP) in finite fields [2].

This report presents a comprehensive theoretical analysis of the ElGamal cryptosystem, examining its mathematical foundations, security properties, and applications in both general and specialized cryptographic protocols. We focus on the implementation aspects relevant to the OpenSSL library on the Windows platform, providing a thorough understanding necessary for practical deployment.

1.1 Motivation and Objectives

The primary objectives of this research are:

- To provide rigorous mathematical foundations of the ElGamal cryptosystem
- To analyze the security assumptions and their implications
- To examine the applicability of ElGamal in general cryptographic protocols (encrypt, digital signatures)
- To explore specialized protocols built upon ElGamal primitives
- To prepare theoretical groundwork for OpenSSL-based implementation on Windows

1.2 Historical Context

The ElGamal cryptosystem emerged during a period of intense development in public-key cryptography. Following the groundbreaking work of Diffie and Hellman on key exchange [3] and the RSA cryptosystem [4], researchers sought alternative approaches with different security foundations. ElGamal's contribution provided not only an encryption scheme but also a digital signature algorithm, both leveraging the difficulty of computing discrete logarithms [1].

2 Mathematical Foundations

2.1 Algebraic Structures

Означення 2.1 (Cyclic Group)**.** *A group $(G, \cdot)$ is cyclic if there exists an element $g \in G$ such that every element of G can be expressed as g^k for some integer k. The element g is called a generator of G.*

Означення 2.2 (Finite Field)**.** *A finite field $\mathbb{F}_p$ (also denoted $GF(p)$ for prime p) is the set $\{0, 1, 2, \ldots, p-1\}$ with addition and multiplication modulo p. The multiplicative group $\mathbb{F}_p^* = \mathbb{F}_p \setminus \{0\}$ has order $p-1$.*

Теорема 2.1 (Primitive Root Theorem)**.** *For any prime p, the multiplicative group $\mathbb{F}_p^*$ is cyclic and has $\phi(p-1)$ generators, where ϕ is Euler's totient function [5].*

2.2 The Discrete Logarithm Problem

Означення 2.3 (Discrete Logarithm Problem (DLP))**.** *Let G be a cyclic group of order n with generator g. Given $h \in G$, the discrete logarithm problem is to find an integer $x \in \{0, 1, \ldots, n-1\}$ such that $g^x = h$. We write $x = \log_g h$ [2].*

The computational intractability of DLP in appropriately chosen groups forms the security foundation of ElGamal. The best known algorithms for solving DLP in prime fields include:

- **Baby-step Giant-step**: Time complexity $O(\sqrt{n})$, space complexity $O(\sqrt{n})$ [6]
- **Pollard's Rho**: Time complexity $O(\sqrt{n})$, space complexity $O(1)$ [7]
- **Index Calculus**: Subexponential time for prime fields, $L_p[1/3, c]$ where $L_p[\alpha, c] = \exp((c + o(1))(\ln p)^{\alpha}(\ln \ln p)^{1-\alpha})$ [8]
- **Number Field Sieve**: Best known algorithm for large prime fields [9]

2.3 Computational Assumptions

Означення 2.4 (Computational Diffie-Hellman (CDH) Assumption)**.** *Given g, g^a, and g^b in a cyclic group G of order n, it is computationally infeasible to compute g^{ab} [10].*

Означення 2.5 (Decisional Diffie-Hellman (DDH) Assumption)**.** *Given g, g^a, g^b, and g^c in a cyclic group G, it is computationally infeasible to distinguish whether $c = ab \pmod{n}$ or c is random [10].*

The ElGamal cryptosystem's security relies on these assumptions, with semantic security requiring the stronger DDH assumption [11].

3 The ElGamal Encryption Scheme

3.1 Key Generation

Algorithm 1 ElGamal Key Generation

```
Require: Security parameter $\lambda$
Ensure: Public key $(p, g, h)$, private key $x$
1: Select a large prime $p$ (typically $|p| \geq 2048$ bits)
2: Choose a generator $g$ of $\mathbb{F}_p^*$ (or a large subgroup)
3: Select random private key $x \xleftarrow{\$} \mathbb{Z}_{p-1}$
4: Compute public key component $h \leftarrow g^x \bmod p$
5: return Public key: $pk = (p, g, h)$, Private key: $sk = x$
```

3.1.1 Parameter Selection

The security of ElGamal critically depends on proper parameter selection:

- **Prime** p: Should be at least 2048 bits for 112-bit security, 3072 bits for 128-bit security [12]
- **Generator** g: Can be a generator of $\mathbb{F}_p^*$ or a subgroup of prime order q where $q|(p-1)$ and q is large (e.g., 256 bits)
- **Private key** x: Uniformly random from $\{1, 2, \ldots, p-2\}$ (or $\{1, 2, \ldots, q-1\}$ for subgroup)

Algorithm 2 ElGamal Encryption

```
Require: Public key $(p, g, h)$, message $m \in \mathbb{F}_p^*$
Ensure: Ciphertext $(c_1, c_2)$
1: Select random ephemeral key $y \xleftarrow{\$} \mathbb{Z}_{p-1}$
2: Compute $c_1 \leftarrow g^y \bmod p$
3: Compute shared secret $s \leftarrow h^y \bmod p$
4: Compute $c_2 \leftarrow m \cdot s \bmod p$
5: return Ciphertext: $C = (c_1, c_2)$
```

3.2 Encryption

3.2.1 Correctness

The correctness of ElGamal encryption follows from the property:

$$\begin{aligned} c_2 \cdot (c_1^x)^{-1} &= m \cdot h^y \cdot (g^y)^{-x} \bmod p & (1) \\ &= m \cdot (g^x)^y \cdot g^{-xy} \bmod p & (2) \\ &= m \cdot g^{xy} \cdot g^{-xy} \bmod p & (3) \\ &= m \bmod p & (4) \end{aligned}$$

3.3 Decryption

Algorithm 3 ElGamal Decryption

```
Require: Private key $x$, ciphertext $(c_1, c_2)$
Ensure: Plaintext $m$
1: Compute shared secret $s \leftarrow c_1^x \bmod p$
2: Compute modular inverse $s^{-1} \leftarrow s^{-1} \bmod p$
3: Recover message $m \leftarrow c_2 \cdot s^{-1} \bmod p$
4: return Plaintext: $m$
```

3.3.1 Computational Efficiency

The computational costs are:

- **Key Generation**: 1 modular exponentiation
- **Encryption**: 2 modular exponentiations, 1 modular multiplication
- **Decryption**: 1 modular exponentiation, 1 modular inversion, 1 modular multiplication

Using fast exponentiation algorithms (square-and-multiply, sliding window) and Montgomery multiplication, these operations are efficient for appropriate parameter sizes [2].

3.4 Message Encoding

A practical challenge with ElGamal is that messages must be elements of $\mathbb{F}_p^*$. Several approaches exist:

1. **Direct Encoding**: For $m < p$, use m directly (requires large p)

2. **Hybrid Encryption**: Use ElGamal to encrypt a symmetric key, then use symmetric encryption (e.g., AES) for the actual message [13]
3. **Block Encoding**: Split message into blocks, each less than p

4 Security Analysis

4.1 Security Definitions

Означення 4.1 (IND-CPA Security). *A cryptosystem is indistinguishable under chosen-plaintext attack (IND-CPA) if no polynomial-time adversary can distinguish between encryptions of two chosen plaintexts with probability significantly better than $1/2$ [14].*

Теорема 4.1 (ElGamal IND-CPA Security). *The ElGamal encryption scheme is IND-CPA secure under the DDH assumption in the underlying group [11].*

Proof Sketch. The security reduction proceeds by showing that if an adversary can break IND-CPA security of ElGamal, then we can construct an algorithm to solve the DDH problem. Given a DDH challenge (g, g^a, g^b, g^c), we set the public key as $h = g^a$ and create a challenge ciphertext as $(g^b, m_i \cdot g^c)$ for randomly chosen $i \in \{0, 1\}$. If $c = ab$, this is a valid ElGamal encryption; otherwise, it's random. The adversary's ability to distinguish these cases directly translates to solving DDH [11]. $\square$

4.2 Malleability and IND-CCA2

A critical weakness of basic ElGamal is its **malleability**: given a ciphertext (c_1, c_2) encrypting message m, anyone can create $(c_1, c_2 \cdot r)$ which decrypts to $m \cdot r$ for any $r \in \mathbb{F}_p^*$.

Припущення 4.2. *Basic ElGamal encryption is not IND-CCA2 (indistinguishable under adaptive chosen-ciphertext attack) secure.*

To achieve IND-CCA2 security, ElGamal can be enhanced using:

- **Fujisaki-Okamoto transformation** [15]
- **OAEP padding** (Optimal Asymmetric Encryption Padding) [16]
- **Cramer-Shoup cryptosystem** (an ElGamal variant) [17]

4.3 Semantic Security

Теорема 4.3 (Semantic Security). *ElGamal encryption provides semantic security under the DDH assumption. Specifically, ciphertexts reveal no information about plaintexts beyond their length [14].*

The randomization in encryption (through ephemeral key y) ensures that encrypting the same message twice produces different ciphertexts, a crucial property for semantic security.

Algorithm 4 ElGamal Signature Generation

```
Require: Private key $x$, message $m$, public parameters $(p, g)$
Ensure: Signature $(r, s)$
1: Select random ephemeral key $k \xleftarrow{\$} \mathbb{Z}_{p-1}^*$ where $\gcd(k, p-1) = 1$
2: Compute $r \leftarrow g^k \bmod p$
3: Compute hash $h \leftarrow H(m)$ where $H$ is a cryptographic hash function
4: Compute $s \leftarrow k^{-1}(h - xr) \bmod (p-1)$
5: return Signature: $\sigma = (r, s)$
```

Algorithm 5 ElGamal Signature Verification

```
Require: Public key $(p, g, h)$, message $m$, signature $(r, s)$
Ensure: valid or invalid
1: Check $0 < r < p$ and $0 < s < p-1$
2: Compute hash $h \leftarrow H(m)$
3: Verify: $g^h \overset{?}{\equiv} h^r \cdot r^s \pmod p$
4: if equation holds then
5:     return valid
6: else
7:     return invalid
8: end if
```

5 ElGamal Digital Signature Scheme

5.1 Signature Generation and Verification

5.2 Signature Correctness

Correctness follows from:

$$\begin{aligned} h^r \cdot r^s &\equiv (g^x)^r \cdot (g^k)^s \pmod p && (5)\\ &\equiv g^{xr+ks} \pmod p && (6)\\ &\equiv g^{xr+k\cdot k^{-1}(h-xr)} \pmod p && (7)\\ &\equiv g^{xr+h-xr} \pmod p && (8)\\ &\equiv g^h \pmod p && (9) \end{aligned}$$

5.3 Security Considerations for Signatures

Critical Requirements:

1. **Ephemeral key uniqueness**: Never reuse k for different messages
2. **Ephemeral key secrecy**: If k is revealed, private key x can be computed: $x \equiv r^{-1}(ks - h) \pmod{p-1}$
3. **Hash function**: Must use cryptographic hash to prevent existential forgery

Припущення 5.1 (Signature Key Recovery Attack)**.** *If two signatures (r, s_1) and (r, s_2) are generated with the same ephemeral key k for messages m_1 and m_2, the private key can be recovered as:*

$$x \equiv r^{-1} \left(\frac{H(m_1) - H(m_2)}{s_1 - s_2} \bmod (p-1) \right) \pmod{p-1}$$

6 Variants and Extensions

6.1 ElGamal in Subgroups

For efficiency, ElGamal can operate in a subgroup of $\mathbb{F}_p^*$ of prime order q where $q|(p-1)$.

Advantages:

- Shorter signatures and ciphertexts (size $|q|$ instead of $|p|$)
- Computationally more efficient
- Matches DSA parameter structure

Parameter Selection:

- Choose safe prime $p = 2q + 1$ where both p and q are prime
- Or use NIST-style parameters: p of 2048-3072 bits, q of 224-256 bits

6.2 Elliptic Curve ElGamal

ElGamal naturally extends to elliptic curve groups, offering significantly smaller key sizes for equivalent security [18, 19].

Algorithm 6 EC-ElGamal Encryption

```
Require: EC public key $(E, G, Q)$, message point $M \in E$
Ensure: Ciphertext $(C_1, C_2)$
1: Select random $k \xleftarrow{\$} \mathbb{Z}_n$ where $n = \text{order}(G)$
2: Compute $C_1 \leftarrow kG$
3: Compute $C_2 \leftarrow M + kQ$
4: return $(C_1, C_2)$
```

Security Equivalence: EC-ElGamal security reduces to ECDLP (Elliptic Curve Discrete Logarithm Problem) and EC-DDH [20].

6.3 Signed ElGamal

To achieve IND-CCA2 security, Signed ElGamal combines encryption with authentication:

1. Encrypt message: $(c_1, c_2) \leftarrow \text{ElGamal.Enc}(pk, m)$
2. Generate signature: $\sigma \leftarrow \text{Sign}(sk_{\text{sig}}, (c_1, c_2))$
3. Output: (c_1, c_2, σ)

This construction achieves IND-CCA2 in the random oracle model [21].

7 General Cryptographic Protocols Using ElGamal

7.1 Threshold Cryptography

ElGamal's structure naturally supports threshold decryption schemes where t out of n parties must cooperate to decrypt [22].

7.1.1 Threshold ElGamal Decryption

Setup:

1. Private key x is shared using Shamir's secret sharing: $x = \sum_{i=1}^{t} x_i \lambda_i$ where λ_i are Lagrange coefficients
2. Each party i holds share x_i
3. Public key: $h = g^x$

Decryption:

1. Given ciphertext (c_1, c_2), each party i computes partial decryption: $d_i = c_1^{x_i}$
2. Combiner reconstructs: $s = \prod_{i=1}^{t} d_i^{\lambda_i} = c_1^{\sum_{i=1}^{t} x_i \lambda_i} = c_1^x$
3. Recover message: $m = c_2/s$

7.2 Re-encryption and Mix Networks

ElGamal's multiplicative homomorphic property enables re-encryption without knowledge of the message [23].

Припущення 7.1 (ElGamal Re-encryption)**.** *Given ciphertext (c_1, c_2), anyone can create a fresh encryption of the same message:*

$$(c_1', c_2') = (c_1 \cdot g^r, c_2 \cdot h^r) \text{ for random } r$$

This decrypts to the same message m.

Application – Mix Networks: Servers can shuffle and re-encrypt votes/messages without learning contents, providing anonymity [23].

7.3 Verifiable Encryption

ElGamal supports zero-knowledge proofs of plaintext properties without revealing the plaintext [24].

Example: Prove that ciphertext (c_1, c_2) encrypts a value in range $[0, 2^n - 1]$ without revealing the value.

8 Specialized Cryptographic Protocols

8.1 Homomorphic Properties

ElGamal exhibits **multiplicative homomorphism**:

Теорема 8.1 (Multiplicative Homomorphism)**.** *Given two ElGamal ciphertexts (c_1, c_2) encrypting m_1 and (c_1', c_2') encrypting m_2:*

$$(c_1 \cdot c_1', c_2 \cdot c_2') \text{ is a valid encryption of } m_1 \cdot m_2$$

Applications:

- Secure voting: Multiply encrypted votes to tally without decrypting individual votes
- Secure auctions: Compute statistics on encrypted bids
- Privacy-preserving data mining

8.2 Electronic Voting Protocols

ElGamal is extensively used in e-voting due to its homomorphic property and verifiability [25].

8.2.1 Basic Voting Protocol

1. **Setup**: Election authority generates ElGamal public key (p, g, h)
2. **Voting**: Voter encodes vote as $m \in \{g^0, g^1\}$ for no/yes, encrypts: $(c_1, c_2) = (g^r, m \cdot h^r)$
3. **Proof**: Voter provides zero-knowledge proof that $m \in \{1, g\}$
4. **Tallying**: Multiply all ciphertexts: $(C_1, C_2) = (\prod c_{1,i}, \prod c_{2,i})$
5. **Result**: Decrypt (C_1, C_2) to get g^{total}, compute discrete log to find total

8.3 Secure Multi-Party Computation

ElGamal serves as a building block for secure multi-party computation (MPC) protocols [26].

Example Protocol – Millionaires' Problem: Two parties want to determine who is richer without revealing their actual wealth.

1. Alice has wealth w_A, Bob has wealth w_B
2. Alice generates ElGamal keypair, publishes pk
3. Alice encrypts: $E(w_A)$
4. Bob computes: $E(w_A - w_B) = E(w_A) \cdot E(-w_B)$ using homomorphism
5. Using additional cryptographic techniques, they determine sign without decryption

8.4 Key Escrow and Fair Cryptography

ElGamal can be modified to support verifiable escrow, where a trusted third party can decrypt under specific conditions [27].

Verifiable Encryption to Third Party:

1. User encrypts message with recipient's key: (c_1, c_2)
2. User also encrypts ephemeral key y with escrow agent's key: (c_1', c_2')
3. User provides zero-knowledge proof that both ciphertexts use same y
4. Escrow agent can recover y, then decrypt (c_1, c_2) if authorized

9 Implementation Considerations for OpenSSL on Windows

9.1 OpenSSL Architecture for ElGamal

OpenSSL does not provide native ElGamal encryption functions, but offers comprehensive building blocks:

- **BIGNUM library**: Arbitrary precision arithmetic
- **BN_mod_exp**: Modular exponentiation (critical for ElGamal)

- **BN_mod_inverse**: Modular inversion (for decryption)
- **DH (Diffie-Hellman)**: Shares parameter structure with ElGamal
- **DSA**: ElGamal signature variant (compatible parameters)

9.2 Using DSA Infrastructure

OpenSSL's DSA implementation can be adapted for ElGamal signatures:

```c
#include <openssl/dsa.h>
#include <openssl/bn.h>

// Generate DSA parameters (compatible with ElGamal)
DSA *dsa = DSA_new();
DSA_generate_parameters_ex(dsa, 2048, NULL, 0, NULL, NULL, NULL);

// Generate key pair
DSA_generate_key(dsa);

// Access parameters
const BIGNUM *p, *q, *g, *pub_key, *priv_key;
DSA_get0_pqg(dsa, &p, &q, &g);
DSA_get0_key(dsa, &pub_key, &priv_key);
```

9.3 Implementing ElGamal Encryption

```c
#include <openssl/bn.h>
#include <openssl/rand.h>

typedef struct {
    BIGNUM *p;      // Prime modulus
    BIGNUM *g;      // Generator
    BIGNUM *h;      // Public key (g^x mod p)
    BIGNUM *x;      // Private key (only for decryption)
} ElGamal_Key;

typedef struct {
    BIGNUM *c1;     // g^y mod p
    BIGNUM *c2;     // m * h^y mod p
} ElGamal_Ciphertext;

int elgamal_encrypt(ElGamal_Key *pubkey, BIGNUM *message,
                    ElGamal_Ciphertext *ciphertext, BN_CTX *ctx) {
    BIGNUM *y = BN_new();
    BIGNUM *shared_secret = BN_new();

    // Generate random ephemeral key y
    BN_rand_range(y, pubkey->p);

    // Compute c1 = g^y mod p
    BN_mod_exp(ciphertext->c1, pubkey->g, y, pubkey->p, ctx);

    // Compute shared secret = h^y mod p
    BN_mod_exp(shared_secret, pubkey->h, y, pubkey->p, ctx);

    // Compute c2 = m * shared_secret mod p
    BN_mod_mul(ciphertext->c2, message, shared_secret,
               pubkey->p, ctx);

    // Cleanup sensitive data
    BN_clear_free(y);
    BN_clear_free(shared_secret);

    return 1;
}

int elgamal_decrypt(ElGamal_Key *privkey,
                    ElGamal_Ciphertext *ciphertext,
                    BIGNUM *message, BN_CTX *ctx) {
    BIGNUM *shared_secret = BN_new();
    BIGNUM *s_inv = BN_new();

    // Compute shared secret = c1^x mod p
    BN_mod_exp(shared_secret, ciphertext->c1, privkey->x,
               privkey->p, ctx);

    // Compute inverse of shared secret
    BN_mod_inverse(s_inv, shared_secret, privkey->p, ctx);

    // Recover message: m = c2 * s^(-1) mod p
    BN_mod_mul(message, ciphertext->c2, s_inv, privkey->p, ctx);

    // Cleanup
    BN_clear_free(shared_secret);
    BN_clear_free(s_inv);

    return 1;
}
```

9.4 Efficient Modular Exponentiation

OpenSSL provides optimized implementations using:

- **Montgomery multiplication**: Efficient modular arithmetic
- **Windowing methods**: Reduce number of multiplications
- **Chinese Remainder Theorem**: For certain parameter choices

```c
// Using BN_CTX for temporary variables (thread-safe)
BN_CTX *ctx = BN_CTX_new();
BN_CTX_start(ctx);
```

```c

BIGNUM *result = BN_CTX_get(ctx);

// Efficient modular exponentiation
// Uses sliding window method automatically
BN_mod_exp(result, base, exponent, modulus, ctx);

BN_CTX_end(ctx);
BN_CTX_free(ctx);
```

9.5 Windows-Specific Considerations

9.5.1 Entropy Sources

On Windows, OpenSSL uses:

- `BCryptGenRandom` (Windows 10+) or `CryptGenRandom` (legacy)
- RDRAND/RDSEED CPU instructions
- System performance counters

9.5.2 Compilation and Linking

```
# Using MinGW-w64
gcc -o elgamal elgamal.c -I/c/OpenSSL/include ^
    -L/c/OpenSSL/lib -lcrypto -lws2_32
```

9.6 Memory Management and Security

```c
// Secure memory allocation for sensitive data
BIGNUM *private_key = BN_secure_new();

// Clear sensitive data before freeing
BN_clear_free(private_key);

// For regular BIGNUMs
BN_free(public_key);

// Securely wipe memory buffer
OPENSSL_cleanse(buffer, buffer_size);
```

9.7 Error Handling

```c
#include <openssl/err.h>

int elgamal_operation() {
    if (!BN_mod_exp(result, base, exp, mod, ctx)) {
        // Print OpenSSL error stack
        unsigned long err = ERR_get_error();
        char err_buf[256];
        ERR_error_string_n(err, err_buf, sizeof(err_buf));
        fprintf(stderr, "OpenSSL error: %s\n", err_buf);
        return 0;
    }
    return 1;
}
```

10 Performance Analysis

10.1 Computational Complexity

Table 1: ElGamal Computational Costs

Operation	Modular Exp.	Other Operations
Key Generation	1	1 multiplication
Encryption	2	1 multiplication
Decryption	1	1 inversion, 1 multiplication
Sign Generation	1	1 inversion, 2 mult., 1 hash
Sign Verification	2	1 multiplication, 1 hash

For n-bit modulus, modular exponentiation requires $O(n^3)$ bit operations using standard algorithms, or $O(n^2 \log n)$ using FFT-based methods [2].

10.2 Comparison with RSA

Table 2: ElGamal vs RSA Performance Comparison

Operation	ElGamal	RSA	Ratio
Key Generation	Moderate	Slow	ElGamal 5-10× faster
Encryption	Slow	Fast	RSA 10-100× faster
Decryption	Moderate	Slow	Similar
Ciphertext Size	2× size	1× size	ElGamal 2× larger

Key Observations:

- RSA encryption is faster (small public exponent $e = 65537$)
- ElGamal key generation is faster (no prime factorization requirement)
- ElGamal ciphertexts are twice the size (two group elements)
- ElGamal provides semantic security without padding (unlike textbook RSA)

10.3 Elliptic Curve Comparison

EC-ElGamal offers significant key size reduction, leading to faster operations and reduced bandwidth requirements [20].

Table 3: Security Levels: ElGamal vs EC-ElGamal

Security (bits)	ElGamal $\|p\|$	EC-ElGamal	Ratio
80	1024	160	6.4:1
112	2048	224	9.1:1
128	3072	256	12:1
192	7680	384	20:1
256	15360	512	30:1

11 Security Recommendations

11.1 Parameter Selection Guidelines

1. **Modulus Size**: Minimum 2048 bits for current security (2023+), 3072 bits for long-term protection (2030+) [12]

2. **Safe Primes**: Use $p = 2q + 1$ where both p and q are prime, ensuring large subgroup order

3. **Generator Selection**:

    - For full group $\mathbb{F}_p^*$: any generator works
    - For subgroup: verify $g^q \equiv 1 \pmod{p}$

4. **Random Number Generation**: Use cryptographically secure PRNG (OpenSSL's `RAND_bytes`)

5. **Ephemeral Key Management**:

    - Generate fresh k or y for each operation
    - Securely erase after use
    - Never reuse across different messages

11.2 Implementation Security

11.2.1 Side-Channel Attack Mitigation

- **Timing Attacks**: Use constant-time implementations

```
// OpenSSL's BN_mod_exp uses constant-time methods
// when BN_FLG_CONSTTIME is set
BN_set_flags(private_key, BN_FLG_CONSTTIME);
BN_mod_exp(result, base, private_key, modulus, ctx);
```

- **Power Analysis**: Implement blinding techniques

- **Cache-Timing**: Use scatter-gather table lookups

11.2.2 Memory Safety

```
// Always check return values
if (!BN_rand_range(ephemeral_key, modulus)) {
    // Handle error - do not proceed
    return ERROR_CODE;
}

// Secure cleanup
BN_clear_free(ephemeral_key);
OPENSSL_cleanse(buffer, sizeof(buffer));
```

11.3 Protocol-Level Security

1. **Key Freshness**: Rotate keys periodically
2. **Perfect Forward Secrecy**: Use ephemeral keys for each session
3. **Authentication**: Combine encryption with signatures or MACs
4. **Padding**: Use structured padding for CCA2 security
5. **Domain Separation**: Use different keys for different purposes

12 Practical Applications and Case Studies

12.1 PGP and GNU Privacy Guard

While PGP primarily uses RSA and DSA, some implementations support ElGamal:

- **Encryption**: ElGamal for key encapsulation
- **Signatures**: DSA (ElGamal variant)
- **Hybrid Approach**: ElGamal encrypts symmetric key, AES encrypts message

12.2 Cryptocurrency Applications

Several cryptocurrency protocols use ElGamal-based constructions:

12.2.1 Pedersen Commitments

Based on ElGamal, used in Monero and other privacy coins:

$$C(m, r) = g^m h^r$$

Computationally hiding and perfectly binding under DLP [28].

12.2.2 Confidential Transactions

Extension of ElGamal homomorphism for hiding transaction amounts while proving validity [29].

12.3 Secure Voting Systems

Helios Voting System: Open-source e-voting using ElGamal [30]

- Homomorphic tallying without decrypting individual votes
- Public verifiability through zero-knowledge proofs
- Used in university elections, organizational votes

12.4 Cloud Security

Searchable Encryption: ElGamal enables searching encrypted data:

- Encrypt keywords with ElGamal
- Use homomorphic properties for equality testing
- Maintain privacy while enabling cloud search [31]

13 Advanced Topics

13.1 Zero-Knowledge Proofs for ElGamal

13.1.1 Proof of Knowledge of Plaintext

Prove knowledge of m in ciphertext $(c_1, c_2) = (g^y, mh^y)$ without revealing m or y:

Algorithm 7 Schnorr-like ZK Proof for ElGamal Plaintext Knowledge

```
1: **Prover** chooses random $r \overset{\$}{\leftarrow} \mathbb{Z}_p$
2: **Prover** sends commitment: $t = g^r$
3: **Verifier** sends challenge: $c \overset{\$}{\leftarrow} \mathbb{Z}_p$
4: **Prover** computes response: $s = r + cy \mod (p-1)$
5: **Verifier** checks: $g^s \overset{?}{=} t \cdot c_1^c$
```

13.1.2 Proof of Correct Decryption

Prove that decryption was performed correctly without revealing private key [32].

13.2 Multi-Receiver Encryption

ElGamal can be extended for efficient broadcast encryption where one ciphertext decrypts to same message for multiple recipients [33].

Construction:

1. Recipients have public keys $h_1 = g^{x_1}, \ldots, h_n = g^{x_n}$
2. Sender generates single ephemeral key y
3. Ciphertext: $(c_1, c_{2,1}, \ldots, c_{2,n}) = (g^y, mh_1^y, \ldots, mh_n^y)$
4. Each recipient i decrypts: $m = c_{2,i}/c_1^{x_i}$

13.3 Identity-Based ElGamal

Extensions to identity-based encryption using bilinear pairings:

- User's public key derived from identity string
- No need for public key certificates
- Private key issued by trusted authority

14 Comparison with Other Cryptosystems

14.1 Feature Comparison

Table 4: Asymmetric Cryptosystem Comparison

Feature	RSA	ElGamal	ECC	Paillier
Encryption	Yes	Yes	Yes	Yes
Signatures	Yes	Yes	Yes	No
Homomorphic	Partially	Multiplicative	No	Additive
Semantic Security	No*	Yes	Yes	Yes
Ciphertext Size	$1\times$	$2\times$	$2\times$	$1\times$
Key Gen Speed	Slow	Fast	Fast	Slow
Encryption Speed	Fast	Moderate	Moderate	Slow
Decryption Speed	Slow	Moderate	Moderate	Slow
Key Size (128-bit sec)	3072	3072	256	3072
Standardization	High	Moderate	High	Low
Patent Issues	No	No	Some	Some

* Requires padding (OAEP) for semantic security

14.2 Security Foundation Comparison

- **RSA**: Integer factorization problem
- **ElGamal**: Discrete logarithm problem (DLP)
- **ECC**: Elliptic curve discrete logarithm problem (ECDLP)
- **Paillier**: Decisional composite residuosity assumption

All are believed quantum-vulnerable; post-quantum alternatives include lattice-based (NTRU, Kyber) and code-based (McEliece) schemes [34].

15 Future Directions and Open Problems

15.1 Post-Quantum Considerations

Shor's algorithm [35] breaks DLP in polynomial time on quantum computers:

- ElGamal vulnerable to sufficiently large quantum computers
- Estimated timeline: 2030-2040 for cryptographically relevant quantum computers
- Migration strategies needed to post-quantum alternatives

Quantum-Resistant Alternatives:

- **Lattice-based**: NTRU, Kyber (NIST standard)
- **Code-based**: McEliece
- **Multivariate**: Rainbow
- **Hash-based**: SPHINCS+

15.2 Efficiency Improvements

Research Directions:

1. **Batch Processing**: Verify multiple signatures simultaneously
2. **Hardware Acceleration**: GPU/FPGA implementations
3. **Compressed Representations**: Reduce ciphertext/signature size
4. **Threshold Optimizations**: Reduce communication rounds

15.3 Protocol Extensions

Active Research Areas:

- **Functional Encryption**: Decrypt functions of plaintexts
- **Proxy Re-encryption**: Delegate decryption rights
- **Verifiable Computation**: Prove correctness of encrypted computation
- **Homomorphic Signatures**: Sign encrypted data

16 Implementation Roadmap for OpenSSL

16.1 Phase 1: Basic Implementation

Objectives:

1. Implement ElGamal key generation using OpenSSL BIGNUM
2. Implement encryption/decryption functions
3. Implement signature generation/verification
4. Create comprehensive test suite

Key Components:

```c
// Data structures
typedef struct elgamal_key_st ELGAMAL_KEY;
typedef struct elgamal_ctx_st ELGAMAL_CTX;

// API functions
ELGAMAL_KEY* ElGamal_new(void);
void ElGamal_free(ELGAMAL_KEY *key);
int ElGamal_generate_key(ELGAMAL_KEY *key, int bits);
int ElGamal_encrypt(ELGAMAL_CTX *ctx, const BIGNUM *plaintext,
                    BIGNUM *c1, BIGNUM *c2);
int ElGamal_decrypt(ELGAMAL_CTX *ctx, const BIGNUM *c1,
                    const BIGNUM *c2, BIGNUM *plaintext);
```

16.2 Phase 2: Protocol Integration

Objectives:

1. Implement hybrid encryption (ElGamal + AES)
2. Add zero-knowledge proof support
3. Implement threshold cryptography
4. Create protocol demonstration applications

16.3 Phase 3: Optimization and Hardening

Objectives:

1. Implement constant-time operations
2. Add side-channel attack mitigations
3. Optimize for Windows platform
4. Performance benchmarking and tuning
5. Security audit and formal verification

16.4 Development Tools and Testing

Recommended Tools:

- **IDE**: Visual Studio 2022, CLion
- **Build System**: CMake for cross-platform compatibility
- **Testing**: Google Test, OpenSSL test framework
- **Profiling**: VTune, Windows Performance Analyzer
- **Memory Checking**: Dr. Memory, Application Verifier

17 Conclusion

The ElGamal cryptosystem represents a fundamental contribution to public-key cryptography, offering unique properties that distinguish it from other asymmetric schemes. Its foundation on the discrete logarithm problem provides security guarantees different from factorization-based systems like RSA, contributing to cryptographic diversity.

17.1 Key Advantages

1. **Semantic Security**: Probabilistic encryption ensures semantic security under DDH assumption
2. **Homomorphic Properties**: Multiplicative homomorphism enables privacy-preserving protocols
3. **Simplicity**: Conceptually elegant with straightforward security proofs
4. **Flexibility**: Easily adaptable to different algebraic structures (subgroups, elliptic curves)
5. **Protocol-Friendly**: Natural fit for threshold cryptography, voting, secure computation

17.2 Practical Considerations

1. **Ciphertext Expansion**: 2× size overhead compared to RSA
2. **Malleability**: Requires additional measures (signatures, CCA2 padding) for some applications
3. **Performance**: Slower encryption than RSA but faster key generation
4. **Standardization**: Less standardized than RSA/ECC in mainstream protocols

17.3 Implementation Outlook

Implementing ElGamal using OpenSSL on Windows is highly feasible:

- OpenSSL provides robust BIGNUM library with all necessary operations
- DSA/DH implementations offer compatible parameter structures
- Windows platform provides adequate entropy sources
- Performance is acceptable for most applications with proper optimization

17.4 Research Significance

ElGamal's importance extends beyond its direct use:

- Foundation for numerous advanced protocols (voting, MPC, homomorphic encryption)
- Theoretical tool for understanding public-key cryptography
- Benchmark for comparing new cryptographic schemes
- Educational value in demonstrating core cryptographic concepts

17.5 Future Work

The proposed implementation will focus on:

1. Creating a robust, secure ElGamal library using OpenSSL
2. Implementing key cryptographic protocols (threshold, voting, re-encryption)
3. Performance analysis and optimization for Windows platform
4. Security evaluation and formal verification
5. Practical demonstrations of ElGamal's unique capabilities

This theoretical foundation provides the necessary background for proceeding with implementation, ensuring that design decisions are grounded in solid cryptographic principles and security analysis.

References

[1] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *IEEE Transactions on Information Theory* 31.4 (1985), pp. 469–472. DOI: 10.1109/TIT.1985.1057074.

[2] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 978-0849385230. URL: http://cacr.uwaterloo.ca/hac/.

[3] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[4] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126. DOI: 10.1145/359340.359342.

[5] Kenneth H. Rosen. *Elementary Number Theory*. 6th. Pearson, 2011. ISBN: 978-0321500311.

[6] Daniel Shanks. "Class Number, a Theory of Factorization, and Genera". In: *Proceedings of Symposia in Pure Mathematics* 20 (1971), pp. 415–440.

[7] John M. Pollard. "Monte Carlo Methods for Index Computation $(\bmod p)$". In: *Mathematics of Computation* 32.143 (1978), pp. 918–924. DOI: 10.2307/2006496.

[8] Daniel M. Gordon. "Discrete Logarithms in $GF(p)$ Using the Number Field Sieve". In: *SIAM Journal on Discrete Mathematics*. Vol. 6. 1. SIAM, 1993, pp. 124–138. DOI: 10.1137/0406010.

[9] Oliver Schirokauer. "Discrete Logarithms and Local Units". In: vol. 345. Royal Society, 1993, pp. 409–423. DOI: 10.1098/rsta.1993.0139.

[10] Dan Boneh. "The Decision Diffie-Hellman Problem". In: *Algorithmic Number Theory: Third International Symposium, ANTS-III*. Springer, 1998, pp. 48–63. DOI: 10.1007/BFb0054851.

[11] Yiannis Tsiounis and Moti Yung. "On the Security of ElGamal Based Encryption". In: *Public Key Cryptography: First International Workshop on Practice and Theory in Public Key Cryptography, PKC'98*. Springer, 1998, pp. 117–134. DOI: 10.1007/BFb0054019.

[12] Elaine Barker. *Recommendation for Key Management: Part 1 – General*. Tech. rep. NIST Special Publication 800-57 Part 1 Revision 5. Gaithersburg, MD: National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.SP.800-57pt1r5.

[13] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. *The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES*. Cryptology ePrint Archive, Paper 2001/108. 2001. URL: https://eprint.iacr.org/2001/108.

[14] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption". In: vol. 28. 2. Elsevier, 1984, pp. 270–299. DOI: 10.1016/0022-0000(84)90070-9.

[15] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology – CRYPTO '99*. Springer, 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34.

[16] Mihir Bellare and Phillip Rogaway. "Optimal Asymmetric Encryption". In: *Advances in Cryptology – EUROCRYPT '94*. Springer, 1994, pp. 92–111. DOI: 10.1007/BFb0053428.

[17] Ronald Cramer and Victor Shoup. "A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack". In: *Advances in Cryptology – CRYPTO '98*. Springer, 1998, pp. 13–25. DOI: 10.1007/BFb0055717.

[18] Neal Koblitz. “Elliptic Curve Cryptosystems”. In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. DOI: 10.2307/2007884.

[19] Victor S. Miller. “Use of Elliptic Curves in Cryptography”. In: *Advances in Cryptology – CRYPTO ’85 Proceedings*. Springer, 1986, pp. 417–426. DOI: 10.1007/3-540-39799-X_31.

[20] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2006. ISBN: 978-0387952734.

[21] Mihir Bellare and Phillip Rogaway. “Random Oracles are Practical: A Paradigm for Designing Efficient Protocols”. In: *1st ACM Conference on Computer and Communications Security*. ACM, 1993, pp. 62–73. DOI: 10.1145/168588.168596.

[22] Yvo Desmedt and Yair Frankel. “Threshold Cryptosystems”. In: *Advances in Cryptology – CRYPTO ’89*. Springer, 1990, pp. 307–315. DOI: 10.1007/0-387-34805-0_28.

[23] David L. Chaum. “Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms”. In: *Communications of the ACM* 24.2 (1981), pp. 84–90. DOI: 10.1145/358549.358563.

[24] Jan Camenisch and Victor Shoup. “Practical Verifiable Encryption and Decryption of Discrete Logarithms”. In: *Advances in Cryptology – CRYPTO 2003*. Springer, 2003, pp. 126–144. DOI: 10.1007/978-3-540-45146-4_8.

[25] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. “A Secure and Optimally Efficient Multi-Authority Election Scheme”. In: *Advances in Cryptology – EUROCRYPT ’97*. Springer, 1997, pp. 103–118. DOI: 10.1007/3-540-69053-0_9.

[26] Oded Goldreich. *Secure Multi-Party Computation*. Cambridge University Press, 1998. URL: http://www.wisdom.weizmann.ac.il/~oded/PSBookFrag/part4N.ps.

[27] Silvio Micali. “Fair Public-Key Cryptosystems”. In: *Advances in Cryptology – CRYPTO ’92*. Springer, 1993, pp. 113–138. DOI: 10.1007/3-540-48071-4_9.

[28] Torben Pryds Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In: *Advances in Cryptology – CRYPTO ’91*. Springer, 1991, pp. 129–140. DOI: 10.1007/3-540-46766-1_9.

[29] Greg Maxwell. *Confidential Transactions*. Technical Report. 2016. URL: https://people.xiph.org/~greg/confidential_values.txt.

[30] Ben Adida. “Helios: Web-based Open-Audit Voting”. In: *17th USENIX Security Symposium*. USENIX Association, 2008, pp. 335–348.

[31] Dan Boneh et al. “Public Key Encryption with Keyword Search”. In: *Advances in Cryptology – EUROCRYPT 2004*. Springer, 2004, pp. 506–522. DOI: 10.1007/978-3-540-24676-3_30.

[32] David Chaum. “Elections with Unconditionally-Secret Ballots and Disruption Equivalent to Breaking RSA”. In: *Advances in Cryptology – EUROCRYPT ’88*. Springer, 1988, pp. 177–182. DOI: 10.1007/3-540-45961-8_15.

[33] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. “Multi-Recipient Encryption Schemes: How to Save on Bandwidth and Computation Without Sacrificing Security”. In: *IEEE Symposium on Security and Privacy*. IEEE, 2000, pp. 97–108. DOI: 10.1109/SECPRI.2000.848450.

[34] Daniel J. Bernstein. “Introduction to Post-Quantum Cryptography”. In: *Post-Quantum Cryptography*. Springer, 2009, pp. 1–14. DOI: 10.1007/978-3-540-88702-7_1.

[35] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: vol. 26. 5. SIAM, 1997. DOI: 10.1137/S0097539795293172.

A Mathematical Notation Reference

Table 5: Notation Used Throughout This Report

Notation	Meaning
$\mathbb{Z}_n$	Integers modulo n: $\{0, 1, \ldots, n-1\}$
$\mathbb{Z}_n^*$	Multiplicative group modulo n (coprime to n)
$\mathbb{F}_p$	Finite field with p elements (prime p)
$\mathbb{F}_p^*$	Multiplicative group of $\mathbb{F}_p$
$g^x \bmod p$	Modular exponentiation
$a \equiv b \pmod{n}$	a is congruent to b modulo n
$\log_g h$	Discrete logarithm of h to base g
$x \xleftarrow{\$} S$	x chosen uniformly at random from set S
$\gcd(a, b)$	Greatest common divisor of a and b
$\phi(n)$	Euler's totient function
$\|x\|$	Bit length of x
$H(\cdot)$	Cryptographic hash function
$\stackrel{?}{=}$	Equality check/verification

B Sample OpenSSL Code: Complete ElGamal Implementation

```c
// elgamal_complete.h
#ifndef ELGAMAL_H
#define ELGAMAL_H

#include <openssl/bn.h>

typedef struct {
    BIGNUM *p;
    BIGNUM *g;
    BIGNUM *h;      // Public key
    BIGNUM *x;      // Private key (NULL for public-only)
} ElGamal_Key;

// Key management
ElGamal_Key* ElGamal_Key_new(void);
void ElGamal_Key_free(ElGamal_Key *key);
int ElGamal_generate_parameters(ElGamal_Key *key, int bits);
int ElGamal_generate_key(ElGamal_Key *key);

// Encryption/Decryption
int ElGamal_encrypt(const ElGamal_Key *pubkey,
                    const BIGNUM *message,
                    BIGNUM *c1, BIGNUM *c2);
int ElGamal_decrypt(const ElGamal_Key *privkey,
                    const BIGNUM *c1, const BIGNUM *c2,
                    BIGNUM *message);
```

```c

// Utility functions
int ElGamal_verify_parameters(const ElGamal_Key *key);
void ElGamal_print_key(const ElGamal_Key *key);

#endif // ELGAMAL_H
```

C Foundational Papers

- ElGamal, T. (1985). ”A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms”
- Diffie, W., Hellman, M. (1976). ”New Directions in Cryptography”
- Goldwasser, S., Micali, S. (1984). ”Probabilistic Encryption”
- Menezes, van Oorschot, Vanstone: ”Handbook of Applied Cryptography”
- Katz, Lindell: ”Introduction to Modern Cryptography”
- Stinson, Paterson: ”Cryptography: Theory and Practice”

D Implementation Guides

- OpenSSL Documentation: https://www.openssl.org/docs/
- ”Network Security with OpenSSL” by Viega, Messier, Chandra
- ”Implementing Cryptography Using Python” by Shannon Bray