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

Звіт до лабораторної №4

за темою:
Особливості існуючих прикладних програм, що використовують криптографічні механізми для захисту інформації

Оформлення звіту:
Юрчук Олексій, ФІ-52мн

October 21, 2025
м. Київ

1 Introduction

Information security has become a critical concern in modern computing systems, where sensitive data transmission and storage require robust protection mechanisms. Cryptographic systems form the foundation of information security, providing confidentiality, integrity, authentication, and non-repudiation services [1]. This report examines the features of existing software systems that implement cryptographic mechanisms, with particular emphasis on asymmetric cryptography, standardized interfaces, and vulnerability assessment.

The evolution of cryptographic software has led to the development of standardized application programming interfaces (APIs) such as Microsoft's CryptoAPI and the Public-Key Cryptography Standards (PKCS) developed by RSA Laboratories. These standards facilitate interoperability between different cryptographic implementations and ensure consistent security properties across diverse platforms [2, 3].

This theoretical foundation establishes the context for practical implementation of cryptographic systems, specifically the ElGamal cryptosystem using the OpenSSL library, and investigation of potential vulnerabilities in cryptographic providers that may arise from imperfections in operating system protection mechanisms.

2 Fundamentals of Cryptographic Systems

2.1 Classification of Cryptographic Algorithms

Cryptographic algorithms are broadly classified into two categories: symmetric and asymmetric cryptography. Symmetric algorithms, such as AES and DES, use the same key for both encryption and decryption operations. While computationally efficient, symmetric systems face the challenge of secure key distribution [4].

Asymmetric or public-key cryptography addresses the key distribution problem by using mathematically related key pairs: a public key for encryption and a private key for decryption. The security of asymmetric systems relies on computational hardness assumptions, such as the difficulty of factoring large integers (RSA) or computing discrete logarithms (ElGamal, DSA) [5].

2.2 The ElGamal Cryptosystem

The ElGamal cryptosystem, proposed by Taher ElGamal in 1985, is based on the discrete logarithm problem in finite fields [6]. The system operates in a multiplicative group of integers modulo a large prime p, where the discrete logarithm problem is believed to be computationally intractable.

Key Generation:

1. Select a large prime p and a generator g of the multiplicative group $\mathbb{Z}_p^*$
2. Choose a random private key x where $1 \leq x \leq p-2$
3. Compute the public key $y = g^x \mod p$
4. Public key: (p, g, y); Private key: x

Encryption: To encrypt a message m where $0 \leq m \leq p-1$:

1. Choose a random ephemeral key k where $1 \leq k \leq p-2$
2. Compute $c_1 = g^k \mod p$
3. Compute $c_2 = m \cdot y^k \mod p$
4. Ciphertext: (c_1, c_2)

Decryption: To decrypt ciphertext (c_1, c_2):

$$m = c_2 \cdot (c_1^x)^{-1} \mod p \tag{1}$$

The security of ElGamal relies on the computational Diffie-Hellman assumption, making it resistant to known cryptanalytic attacks when properly implemented with sufficiently large parameters [6].

2.3 Cryptographic Primitives and Services

Modern cryptographic systems provide four fundamental security services:

Confidentiality ensures that information is accessible only to authorized parties through encryption mechanisms. Both symmetric and asymmetric algorithms can provide confidentiality, though asymmetric systems are typically used for key exchange due to computational overhead.

Integrity guarantees that information has not been altered during transmission or storage. Hash functions and message authentication codes (MACs) provide integrity verification without preventing modification attempts.

Authentication verifies the identity of communicating parties. Digital signatures, based on asymmetric cryptography, provide strong authentication by demonstrating possession of a private key corresponding to a known public key.

Non-repudiation prevents parties from denying their actions. Digital signatures provide non-repudiation by creating verifiable evidence that a specific party generated a message [1].

3 Standardized Cryptographic Interfaces

3.1 Microsoft CryptoAPI

Microsoft's Cryptographic Application Programming Interface (CryptoAPI) provides a framework for integrating cryptographic functionality into Windows applications [3]. The architecture consists of several layers:

Application Layer: Applications interact with CryptoAPI through high-level functions that abstract cryptographic operations.

CryptoAPI Layer: This layer provides standardized interfaces for cryptographic operations including hashing, encryption, digital signatures, and certificate management.

Cryptographic Service Provider (CSP) Layer: CSPs implement the actual cryptographic algorithms. Multiple CSPs can coexist on a system, allowing applications to select appropriate providers based on security requirements or regulatory compliance.

The CryptoAPI architecture supports both software-based and hardware-based cryptographic implementations. Hardware security modules (HSMs) can be integrated as CSPs, providing enhanced key protection through dedicated cryptographic hardware.

Key features of CryptoAPI include:

- Algorithm independence through provider abstraction
- Support for cryptographic hardware tokens
- Certificate-based public key infrastructure (PKI)
- Secure key storage and management
- Cryptographic random number generation

3.2 Public-Key Cryptography Standards (PKCS)

The PKCS family of standards, developed by RSA Laboratories, defines interoperable cryptographic data formats and protocols [2]. These standards ensure compatibility between different cryptographic implementations and facilitate secure data exchange across heterogeneous systems.

PKCS #1 specifies RSA encryption and signature schemes, including padding methods (PKCS#1 v1.5 and OAEP) that prevent various cryptanalytic attacks. The standard defines data formats for RSA public and private keys, encrypted messages, and digital signatures.

PKCS #3 describes the Diffie-Hellman key agreement protocol, enabling two parties to establish a shared secret over an insecure channel. This standard specifies domain parameters and key derivation procedures.

PKCS #5 defines password-based encryption schemes, including key derivation functions (PBKDF2) that transform passwords into cryptographic keys with appropriate entropy. These functions incorporate iteration counts and salt values to resist dictionary attacks.

PKCS #7 (now superseded by RFC 5652 Cryptographic Message Syntax) specifies formats for digitally signed, encrypted, or authenticated data. This standard supports multiple signature and encryption algorithms, enabling flexible security policies.

PKCS #11 (Cryptoki) defines a platform-independent API for cryptographic tokens such as smart cards and HSMs. This standard enables applications to access cryptographic services without device-specific code.

PKCS #12 specifies a portable format for storing and transporting private keys, certificates, and other cryptographic data, typically protected by password-based encryption.

3.3 OpenSSL Library Architecture

OpenSSL is a robust, open-source implementation of SSL/TLS protocols and a general-purpose cryptography library [7]. The library provides comprehensive cryptographic functionality including:

Symmetric Cryptography: Implementation of algorithms including AES, DES, 3DES, Blowfish, RC4, and ChaCha20, supporting multiple modes of operation (ECB, CBC, CFB, OFB, CTR, GCM).

Asymmetric Cryptography: Support for RSA, DSA, Diffie-Hellman, and elliptic curve cryptography (ECC). While OpenSSL does not include native ElGamal encryption, the library provides the necessary primitives (modular exponentiation, prime generation, random number generation) for implementing ElGamal.

Hash Functions: Implementation of MD5, SHA-1, SHA-2 family (SHA-224, SHA-256, SHA-384, SHA-512), and SHA-3, along with message authentication codes (HMAC).

Certificate Management: X.509 certificate parsing, generation, and validation, supporting public key infrastructure operations.

Random Number Generation: Cryptographically secure pseudorandom number generator (CSPRNG) essential for key generation and nonce creation.

The OpenSSL architecture separates cryptographic primitives from protocol implementations, enabling developers to utilize low-level cryptographic functions for custom applications such as implementing the ElGamal cryptosystem.

4 Implementation Considerations for Cryptographic Software

4.1 Security Requirements

Implementing cryptographic systems requires careful attention to security requirements beyond algorithmic correctness:

Key Management: Secure generation, storage, distribution, and destruction of cryptographic keys constitute critical security concerns. Keys must be generated using cryptographically secure random number generators with sufficient entropy. Private keys require protection against unauthorized access through encryption, access controls, or hardware security modules [4].

Random Number Generation: Cryptographic operations depend on high-quality randomness. Predictable random numbers compromise security by enabling key recovery or signature forgery attacks. Operating systems provide entropy sources from hardware interrupts, user input, and environmental noise, which cryptographic libraries must properly collect and process.

Side-Channel Resistance: Implementation vulnerabilities may leak secret information through timing variations, power consumption, electromagnetic radiation, or cache access patterns. Constant-time algorithms and secure coding practices mitigate timing attacks [8].

Fault Attacks: Hardware faults induced through environmental manipulation (voltage glitches, temperature variations, radiation) can cause cryptographic implementations to reveal secret keys. Robust error detection and validation mechanisms provide defense against fault injection attacks [9].

4.2 Parameter Selection

Cryptographic security depends critically on parameter choices. For ElGamal implementation:

Prime Selection: The prime p should be at least 2048 bits for contemporary security, with 3072 or 4096 bits recommended for long-term protection. The prime should be a safe prime ($p = 2q + 1$ where q is also prime) to ensure a large prime-order subgroup.

Generator Selection: The generator g must have large order in the multiplicative group. For safe primes, selecting $g = 2$ or verifying that $g^q \mod p \neq 1$ ensures appropriate order.

Random Number Quality: The private key x and ephemeral keys k must be generated using cryptographically secure random number generators. Reusing ephemeral keys or using predictable values catastrophically compromises security.

4.3 Standards Compliance

Compliance with established standards ensures interoperability and adherence to security best practices:

FIPS 140-2/140-3: Federal Information Processing Standards specify security requirements for cryptographic modules, including physical security, key management, and self-tests. FIPS-validated implementations undergo rigorous testing and certification [10].

Common Criteria: International standard (ISO/IEC 15408) for evaluating information technology security. Cryptographic modules can be evaluated against protection profiles specifying security requirements and assurance levels.

Algorithm Standards: NIST Special Publications and FIPS documents specify approved algorithms, key sizes, and operational modes. Compliance demonstrates adherence to government-approved cryptographic practices.

5 Vulnerabilities in Cryptographic Implementations

5.1 Operating System Protection Mechanisms

Cryptographic software depends on operating system security services including memory protection, access control, and process isolation. Imperfections in these mechanisms create vulnerabilities:

Memory Disclosure: Page file swapping may write cryptographic keys to disk in plaintext. Memory dumps generated during system crashes or debugging may expose sensitive data. Secure memory allocation functions prevent swapping and ensure memory zeroing after use.

Side-Channel Leakage: Operating system scheduling and shared resource management create timing channels. Cache timing attacks exploit shared CPU caches to infer cryptographic keys through access pattern analysis. Speculative execution vulnerabilities (Spectre, Meltdown) bypass memory isolation, potentially exposing cryptographic material.

API Vulnerabilities: Cryptographic APIs may contain design flaws enabling misuse. Improper error handling may leak information about private keys through error messages or timing differences between valid and invalid operations.

5.2 Implementation Attacks

Beyond algorithmic security, implementation vulnerabilities threaten cryptographic systems:

Timing Attacks: Variations in execution time based on secret data enable key recovery. Conditional branches and data-dependent memory accesses create timing channels. Constant-time implementations eliminate data-dependent timing variations [8].

Buffer Overflow: Memory safety violations enable code injection or arbitrary memory access. Cryptographic libraries must employ bounds checking and safe memory management.

Side-Channel Analysis: Power analysis and electromagnetic emission analysis recover keys from physical devices. Software side channels include cache timing, branch prediction, and speculative execution.

Fault Injection: Inducing computational errors during cryptographic operations may reveal secret keys. Redundant computation and error detection provide defense against fault attacks [9].

5.3 Attack Resistance Evaluation

Evaluating cryptographic provider resistance requires systematic testing:

Timing Analysis: Statistical analysis of operation timing across various inputs identifies timing channels. Constant-time verification ensures execution time independence from secret data.

Memory Safety: Fuzzing and static analysis detect buffer overflows, use-after-free vulnerabilities, and other memory corruption issues.

Information Leakage: Monitoring system logs, error messages, and debugging output identifies unintentional information disclosure.

Privilege Escalation: Testing cryptographic provider behavior under various privilege levels and access control configurations reveals protection mechanism weaknesses.

6 Practical Implementation Framework

6.1 ElGamal Implementation Using OpenSSL

Implementing the ElGamal cryptosystem using OpenSSL requires utilizing the library's big number arithmetic and random number generation facilities. The implementation process involves:

Parameter Generation: Using OpenSSL's prime generation functions to create safe primes of appropriate size. The `BN_generate_prime_ex()` function generates probable primes with configurable size and properties.

Key Pair Generation: Selecting appropriate generators and computing public keys through modular exponentiation using `BN_mod_exp()`. Private keys are generated using the cryptographically secure random number generator.

Encryption Implementation: Generating ephemeral keys for each encryption operation and performing modular arithmetic to compute ciphertext components. Message encoding may require padding schemes to handle messages shorter than the modulus.

Decryption Implementation: Computing modular inverses using `BN_mod_inverse()` and performing modular multiplication to recover plaintext.

6.2 Security Testing Methodology

Comprehensive security testing evaluates both algorithmic correctness and implementation resistance to attacks:

Functional Testing: Verifying correct encryption and decryption across various message sizes and key parameters. Testing boundary conditions and error handling.

Timing Analysis: Measuring operation timing to detect data-dependent variations. Statistical analysis identifies potential timing channels.

Memory Safety Testing: Employing memory sanitizers (AddressSanitizer, MemorySanitizer) and valgrind to detect memory errors. Fuzzing with random inputs identifies crash conditions.

API Misuse Testing: Testing cryptographic provider behavior under incorrect parameter usage, exploring error conditions and exception handling.

Privilege Testing: Evaluating key protection under different user privilege levels and access control configurations.

7 Conclusion

Cryptographic mechanisms form the foundation of information security in modern software systems. This theoretical examination has explored the fundamental principles of cryptographic systems, with emphasis on asymmetric cryptography and the ElGamal cryptosystem. Standardized interfaces such as CryptoAPI and PKCS facilitate interoperable and consistent cryptographic implementations across diverse platforms.

The security of cryptographic software extends beyond algorithmic strength to encompass implementation details, parameter selection, and resistance to side-channel attacks. Operating system protection mechanisms play crucial roles in securing cryptographic operations, yet imperfections in these mechanisms create potential vulnerabilities. Implementation attacks including timing analysis, fault injection, and memory disclosure threaten even mathematically secure algorithms.

Practical implementation of the ElGamal cryptosystem using the OpenSSL library demonstrates the application of theoretical principles to real-world cryptographic software development. The implementation must address key generation, random number quality, parameter selection, and side-channel resistance while complying with established standards and best practices.

Systematic evaluation of cryptographic provider resistance to attacks exploiting operating system protection mechanism imperfections requires comprehensive testing including timing analysis, memory safety verification, and privilege escalation testing. This multifaceted approach ensures robust cryptographic implementations capable of protecting sensitive information in contemporary computing environments.

The theoretical foundation established in this report provides the necessary context for practical implementation work and security evaluation, enabling the development of secure, standards-compliant cryptographic systems.

References

[1] William Stallings. *Cryptography and Network Security: Principles and Practice*. 7th. Pearson, 2017.

[2] RSA Laboratories. *PKCS #1: RSA Cryptography Standard*. Tech. rep. Version 2.2. RFC 8017. RSA Security Inc., 2012.

[3] *CryptoAPI System Architecture*. Microsoft Developer Network. Microsoft Corporation. 2018.

[4] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, 2010.

[5] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[6] 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.

[7] *OpenSSL Documentation*. OpenSSL Software Foundation. 2023. URL: https://www.openssl.org/docs/.

[8] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Springer, 1996, pp. 104–113.

[9] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults". In: *Advances in Cryptology — EUROCRYPT '97* (1997), pp. 37–51.

[10] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. Tech. rep. FIPS PUB 186-4. NIST, 2013.

A Glossary

ASLR Address Space Layout Randomization - Security technique that randomizes memory addresses

ASN.1 Abstract Syntax Notation One - Standard for describing data structures

CNG Cryptography API: Next Generation - Modern Windows cryptographic API

CRT Chinese Remainder Theorem - Optimization for RSA operations

CSP Cryptographic Service Provider - CryptoAPI implementation module

DEP Data Execution Prevention - Memory protection preventing code execution from data pages

DER Distinguished Encoding Rules - Binary encoding for ASN.1

DPAPI Data Protection API - Windows API for encrypting user data

HSM Hardware Security Module - Dedicated cryptographic hardware

IND-CCA2 Indistinguishability under Adaptive Chosen Ciphertext Attack

IND-CPA Indistinguishability under Chosen Plaintext Attack

KEK Key Encryption Key - Key used to encrypt other keys

KSP Key Storage Provider - CNG key storage implementation

OAEP Optimal Asymmetric Encryption Padding - RSA padding scheme

OID Object Identifier - Unique identifier for algorithms/objects

PEM Privacy Enhanced Mail - Base64 ASCII encoding format

PKCS Public-Key Cryptography Standards - RSA Security standards

TPM Trusted Platform Module - Hardware cryptographic processor

UAC User Account Control - Windows privilege management