НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ СІКОРСЬКОГО»

ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Лабораторна робота 3

Реалізація основних асиметричних криптосистем

Підгрупа 2А

Виконали:

Галіца О.О.

Паршин О.Ю.

Литвиненко Ю.С.

ФІ-22мн

Перевірила:

Байденко П.В.

Київ 2022

1 Теоретичні відомості

1.1 Криптосистема шифрування Ель-Гамаля

Розглянемо криптографічну схему шифрування, яка побудована на односторонній функції Діффі-Хеллмана. Цей алгоритм запропоновано в 1985 році, пізніше найбільш відомої та поширеної схеми асиметричної криптографії RSA. Алгоритм шифрування Ель-Гамаля має загальну структуру систем шифрування з відкритими ключами: систему шифрування з відкритим і секретним ключами будує один абонент , секретний ключ знає тільки і саме секретний ключ використовується для розшифрування шифрованого повідомлення, яке зашифровано «загально доступним» ключем будь-яким іншим абонентом.

Побудова схеми шифрування Ель-Гамаля користувачем А.

Для цього абонент A виконує наступні кроки:

1. Обирає велике просте число p та α — примітивний елемент поля F_p.
2. Обирає випадкове число k, $1 < k < p$ — секретний ключ A, який використовується для розшифрування.
3. Обчислює $y = \alpha^k \ mod\ p$. Параметри (p, α, y) складають відкритий ключ, що застосовується для шифрування повідомлень, призначених для абонента A.
4. Забезпечує доступність відкритого ключа для всіх абонентів, які бажають відіслати A зашифроване повідомлення.
   Відкритий ключ може бути з підтвердженням автентичності (зазвичай, центрами сертифікації ключів ЦСК).

Шифрування повідомлення M, 1<M<p в системі шифрування Ель-Гамаля користувачем B для користувача A, який побудував систему і має секретний ключ для розшифрування.

Користувач B, який знає алгоритм шифрування і відкритий ключ хоче переслати користувачу A зашифроване повідомлення, що кодується і є відповідним цілим числом M, $1 < M < p$. Для цього він виконує наступні дії:

1. Вибирає випадкове число x_M, $1 < x_M < p - 1$
2. Обчислює $C_1 = \alpha^{x_M} \ mod\ p$ та $C_2 = y^{x_M} \ mod\ p$
3. Формує шифрований текст, що є впорядкованою парою чисел (складається з двох частин): (C_1,C_2)

Розшифрування зашифрованого повідомлення в системі шифрування Ель-Гамаля користувачем А.

Користувач A обчислює:

$$C_1^{-k}C_2 \ mod\ p = \alpha^{-x_M k} \cdot \alpha^{k x_M} \ mod\ p = M$$

Значення C_1^{-k} можна обчислити як $(C_1^{-1})^k \ mod\ p$, або як $(C_1^k)^{-1} \ mod\ p$, де обернений за модулем швидко знаходиться алгоритмом Евкліда.

Стійкість схеми шифрування Ель-Гамаля обумовлена складністю розв'язання задачі дискретного логарифмування.

Зауваження

Для різних повідомлень M та $M^{'}$ числа X_M та $X_{M'}$ мають бути різними. Таким чином, неякісний генератор випадкових чисел, використаний для вибору чисел x_M, та порушення процедури шифрування може призвести до рівності $x_M = x_{M'}$ і, відповідно, до успішної атаки на систему.

1.2 Цифровий підпис Ель-Гамаля

Побудова криптосистеми ЦП Ель-Гамаля користувачем А.

Користувач A виконує наступні кроки:

1. Обирає велике просте число p та α — примітивний елемент поля F_p.
2. Обирає випадкове число k, $1 < k < p$ — секретний ключ для формування ЦП.
3. Обчислює $y = \alpha^k \ mod\ p$. Параметри (p, α, y) складають відкритий ключ, що застосовується для перевірки електронного підпису користувача A.

Формування ЦП для повідомлення M, 1<M<p в системі, ЦП Ель-Гамаля користувачем А.

1. Вибирає випадкове число x_M, $1 < x_M < p-1$, (x_M,$p-1$) $= 1$
2. Обчислює $r = \alpha^{x_M} \ mod\ p$
3. Розв'язує відносно s рівняння $M \equiv (kr + x_M s) \ mod\ (p-1)$, тобто $s = (M - kr)x_M^{-1} \ mod\ (p-1)$
4. Формує повідомлення з цифровим підписом: (M,r,s), де пара чисел (r,s) є цифровим підписом.

Перевірка цифрового підпису Ель-Гамаля

Користувачем B перевіряється рівність:

$$y^r r^s \ mod \ p = \alpha^M \ mod \ p$$

Якщо ця рівність виконується, то підпис вірний, підтверджена цілісність повідомлення і справжність автора. Інакше мало місце навмисне чи випадкове спотворення повідомлення або підпису.

Зауваження

Для різних $M_1 \neq M_2$ параметри ЦП x_{M_1} і x_{M_2} повинні бути різними. Тобто, x_{M_i} не повинно ніколи повторюватися. Інакше криптоаналітик E знаходить секретний ключ для підписування повідомлень і зламує систему цифрового підпису Ель-Гамаля.

Стійкість криптосистеми цифрового підпису Ель-Гамаля як і криптосистеми шифрування обумовлена складністю розв'язання задачі дискретного логарифмування.

2 Результати

В ході лабораторної роботи була реалізована криптосистема Ель-Гамаля з параметризованою довжиною ключа. В якості допоміжної бібліотеки для роботи з довгою арифметикою була використана бібліотека OpenSSL.

Тестове повідомлення: D4D2110984907B5625309D956521BAB4157B8B1ECE04043249A3D379AC112E5B9AF44E721E148D88A942744CF56A06B92D28A0DB950FE4CED2B41A0BD38BCE7D0BE1055CF5DE38F2A588C2C9A79A75011058C320A7B661C6CE1C36C7D870758307E5D2CF07D9B6E8D5299779B6B2910DD17B6766A7EFEE215A98CAC300F2827DB

Рис. 1: Шифрування та розшифрування

Рис. 2: Підписування та перевірка підпису

3 Лістинг програми

3.1 el-gamal.cpp

```cpp
#include "./el-gamal.hpp";


ElGamal::ElGamal(const uint64_t& keyLength)
{
    this->keyLength = keyLength;

    p = rand_prime_big_number(keyLength);
    q = (p - 1) / 2;
    g = BigNumber::sqr_mod(rand_big_number(keyLength - 1, -1), p);
    x = rand_big_number(keyLength - 2, 0);
    y = BigNumber::exp_mod(g, x, p);
}
```

```cpp
std::pair<BigNumber, BigNumber> ElGamal::encrypt(const BigNumber& m)
{
    BigNumber k = rand_big_number(keyLength - 2, 0);

    auto c = std::pair<BigNumber, BigNumber>();
    c.first = BigNumber::exp_mod(g, k, p);
    c.second = (m * BigNumber::exp_mod(y, k, p)) % p;

    return c;
}

BigNumber ElGamal::decrypt(const std::pair<BigNumber, BigNumber>& c)
{
    return (c.second * BigNumber::exp_mod(c.first, -x % (p - 1), p)) % p;
}

std::pair<BigNumber, BigNumber> ElGamal::sign(const BigNumber& m)
{
    BigNumber k = rand_big_number(keyLength - 2, 0);
    auto sign = std::pair<BigNumber, BigNumber>();

    sign.first = BigNumber::exp_mod(g, k, p);
    sign.second = (m - sign.first * x) * BigNumber::exp_mod(k, q - 2, q) % q;

    return sign;
}

bool ElGamal::verify(const BigNumber& m, const std::pair<BigNumber, BigNumber>& sign)
{
    BigNumber leftPart = BigNumber::exp_mod(y, sign.first, p) * BigNumber::exp_mod(
        sign.first,
        sign.second,
        p
    ) % p;

    BigNumber rightPart = BigNumber::exp_mod(g, m, p);

    return leftPart == rightPart;
}
```

3.2 big-number.cpp

```cpp
#include "./big-number.hpp";


BigNumber::BigNumber(): value(initEmpty(), ::BN_free) {}
BigNumber::BigNumber(const int64_t number): value(initDecimal(number), ::BN_free) {}
BigNumber::BigNumber(const std::string& str): value(initHex(str), ::BN_free) {}
BigNumber::BigNumber(const BigNumber& obj): value(copy(obj.value), ::BN_free) {}

BigNumber& BigNumber::operator=(const BigNumber& obj)
{
    if (this != &obj) value.reset(BN_dup(obj.value.get()));

    return *this;
}

BigNumber BigNumber::add(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    BigNumber result;

    BN_add(result.value.get(), firstOperand.value.get(), secondOperand.value.get());

    return result;
}

BigNumber BigNumber::sub(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    BigNumber result;

    BN_sub(result.value.get(), firstOperand.value.get(), secondOperand.value.get());

    return result;
}

BigNumber BigNumber::mod(const BigNumber& number, const BigNumber& modulo)
{
    BigNumber result;
    bigNumberContext ctx(BN_CTX_new(), ::BN_CTX_free);

    BN_nnmod(result.value.get(), number.value.get(), modulo.value.get(), ctx.get());
```

```
    return result;
}

BigNumber BigNumber::add_mod(
    const BigNumber& firstOperand,
    const BigNumber& secondOperand,
    const BigNumber& modulo
)
{
    BigNumber result;
    bigNumberContext ctx(BN_CTX_new(), ::BN_CTX_free);

    BN_mod_add(
        result.value.get(),
        firstOperand.value.get(),
        secondOperand.value.get(),
        modulo.value.get(),
        ctx.get()
    );

    return result;
}

BigNumber BigNumber::mul(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    BigNumber result;
    bigNumberContext ctx(BN_CTX_new(), ::BN_CTX_free);

    BN_mul(
        result.value.get(),
        firstOperand.value.get(),
        secondOperand.value.get(),
        ctx.get()
    );

    return result;
}

BigNumber BigNumber::mul_mod(
```

```
    const BigNumber& firstOperand,
    const BigNumber& secondOperand,
    const BigNumber& modulo
)
{
    BigNumber result;
    bigNumberContext ctx(BN_CTX_new(), ::BN_CTX_free);

    BN_mod_mul(
        result.value.get(),
        firstOperand.value.get(),
        secondOperand.value.get(),
        modulo.value.get(), ctx.get()
    );

    return result;
}

BigNumber BigNumber::div(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    BigNumber result;
    bigNumberContext ctx(BN_CTX_new(), ::BN_CTX_free);

    BN_div(
        result.value.get(),
        nullptr,
        firstOperand.value.get(),
        secondOperand.value.get(),
        ctx.get()
    );

    return result;
}

BigNumber BigNumber::sqr_mod(const BigNumber& number, const BigNumber& modulo)
{
    return BigNumber::mul_mod(number, number, modulo);
}

BigNumber BigNumber::exp_mod(
```

```
    const BigNumber& number,
    const BigNumber& power,
    const BigNumber& modulo
)
{
    BigNumber result;
    bigNumberContext ctx(BN_CTX_new(), ::BN_CTX_free);

    BN_mod_exp(
        result.value.get(),
        number.value.get(),
        power.value.get(),
        modulo.value.get(),
        ctx.get()
    );

    return result;
}

BIGNUM* BigNumber::initEmpty()
{
    auto bigNumber = BN_new();
    BN_zero(bigNumber);

    return bigNumber;
}

BIGNUM* BigNumber::initDecimal(const int64_t& number)
{
    auto bigNumber = BN_new();
    BN_dec2bn(&bigNumber, std::to_string(number).c_str());

    return bigNumber;
}

BIGNUM* BigNumber::initHex(const std::string& hexString)
{
    auto bigNumber = BN_new();
    BN_hex2bn(&bigNumber, hexString.c_str());
```

```
    return bigNumber;
}

BIGNUM* BigNumber::copy(const bigNumberPointer& arg)
{
    return BN_dup(arg.get());
}

BigNumber rand_big_number(const uint64_t& bitsCount, const int8_t& firstSignificantBit)
{
    BigNumber randomNumber;
    BN_rand(randomNumber.value.get(), bitsCount - 1, firstSignificantBit, false);

    return randomNumber;
}

BigNumber rand_prime_big_number(const uint64_t& bitsCount)
{
    BigNumber randomPrime;
    BN_generate_prime_ex(randomPrime.value.get(), bitsCount, 1, NULL, NULL, NULL);

    return randomPrime;
}

BigNumber operator+(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::add(firstOperand, secondOperand);
}

BigNumber operator+(const int64_t& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::add(firstOperand, secondOperand);
}

BigNumber operator+(const BigNumber& firstOperand, const int64_t& secondOperand)
{
    return BigNumber::add(firstOperand, secondOperand);
}

BigNumber operator-(const BigNumber& number)
```

```
{
    return -1 * number;
}

BigNumber operator-(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::sub(firstOperand, secondOperand);
}

BigNumber operator-(const int64_t& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::sub(firstOperand, secondOperand);
}

BigNumber operator-(const BigNumber& firstOperand, const int64_t& secondOperand)
{
    return BigNumber::sub(firstOperand, secondOperand);
}

BigNumber operator*(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::mul(firstOperand, secondOperand);
}

BigNumber operator*(const int64_t& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::mul(firstOperand, secondOperand);
}

BigNumber operator*(const BigNumber& firstOperand, const int64_t& secondOperand)
{
    return BigNumber::mul(firstOperand, secondOperand);
}

BigNumber operator/(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::div(firstOperand, secondOperand);
}

BigNumber operator/(const int64_t& firstOperand, const BigNumber& secondOperand)
```

```
{
    return BigNumber::div(firstOperand, secondOperand);
}

BigNumber operator/(const BigNumber& firstOperand, const int64_t& secondOperand)
{
    return BigNumber::div(firstOperand, secondOperand);
}

BigNumber operator%(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::mod(firstOperand, secondOperand);
}

BigNumber operator%(const int64_t& firstOperand, const BigNumber& secondOperand)
{
    return BigNumber::mod(firstOperand, secondOperand);
}

BigNumber operator%(const BigNumber& firstOperand, const int64_t& secondOperand)
{
    return BigNumber::mod(firstOperand, secondOperand);
}

bool operator==(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BN_cmp(firstOperand.value.get(), secondOperand.value.get()) == 0;
}

bool operator<(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BN_cmp(firstOperand.value.get(), secondOperand.value.get()) == -1;
}

bool operator>(const BigNumber& firstOperand, const BigNumber& secondOperand)
{
    return BN_cmp(firstOperand.value.get(), secondOperand.value.get()) == 1;
}

std::ostream& operator<<(std::ostream& out, const BigNumber& number)
```

```
{
    const long f = out.flags() & std::ios::basefield;
    char* ptr = nullptr;

    ptr = BN_bn2hex(number.value.get());
    out << ptr;

    return out;
}
```