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

Звіт до лабораторної №2

за темою:

Аналіз of Pseudo-Random Number Generators(PRNG) та генераторів ключів для бібліотеки OpenSSL C++ Library

Оформлення звіту:
Юрчук Олексій, ФІ-52мн

9 жовтня 2025 р.
м. Київ

ЗМІСТ

1 Вступ

У цій лабораторній я намагався зробити комплексний аналіз алгоритмів генерації псевдовипадкових чисел (PRNG), методів тестування на простоту та методів генерації простих чисел, реалізованих у криптографічній бібліотеці OpenSSL для платформи Windows. Основну увагу зосередив на часовій ефективності, зручності використання для генерації ключів асиметричної криптосистеми та аналізу стабільності реалізацій OpenSSL [1, 2].

2 Генератори псевдовипадкових чисел в OpenSSL

2.1 RAND_bytes Function

2.1.1 Description

Функція `RAND_bytes()` є основним інтерфейсом для генерації криптографічно захищених псевдовипадкових байтів в OpenSSL. Вона використовує OpenSSL PRNG, який базується на поєднанні джерел ентропії та криптографічних алгоритмів [1, 3].

2.1.2 Algorithm

OpenSSL використовує DRBG (Deterministic Random Bit Generator) на основі CTR-DRBG з AES-256 як зазначено в NIST SP 800-90A [3].
Алгоритм є наступним:

- Збирається ентропія з системних джерел (Windows CryptoAPI, hardware RNG якщо доступно)
- Seeds the DRBG за допомогою зібраної ентропії
- Генерує псевдовипадкові дані, використовуючи AES-CTR mode
- Періодично reseeds для забезпеченя і підтримки безпеки

Algorithm 1 CTR-DRBG Generate Algorithm

Require: Внутрішній стан $(Key, V, reseed\_counter)$
Require: Кількість бітів для генерації n
Ensure: Псевдовипадкові біти на $output$

if $reseed\_counter > reseed\_interval$ **then**
    Reseed DRBG
end if
$temp \leftarrow \emptyset$
while $length(temp) < n$ **do**
    $V \leftarrow (V + 1) \mod 2^{blocklen}$
    $output\_block \leftarrow AES\_Encrypt(Key, V)$
    $temp \leftarrow temp \| output\_block$
end while
$output \leftarrow$ leftmost n bits of $temp$
$reseed\_counter \leftarrow reseed\_counter + 1$
return $output$

2.1.3 Function Signature

```c
int RAND_bytes(unsigned char *buf, int num);
```

2.1.4 Input Parameters

- `buf`: Вказівник на буфер, де зберігатимуться випадкові байти
- `num`: Кількість випадкових байтів, що будуть згенеровані (integer)

2.1.5 Output Data

- Buffer `buf` is filled with `num` cryptographically secure random bytes

2.1.6 Return Codes

- **1**: Success – випадкові байти згенеровані успішно
- **0**: Failure – PRNG seeded з недостатньою ентропією
- **-1**: Функція не підтримується (рідко)

2.1.7 Приклад використання

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

int main() {
    unsigned char buffer[32];

    if (RAND_bytes(buffer, 32) != 1) {
        fprintf(stderr, "RAND_bytes failed\n");
        return 1;
    }

    printf("Generated random bytes successfully\n");
    return 0;
}
```

2.2 RAND_priv_bytes Function

2.2.1 Description

Подібна до `RAND_bytes()`, але спеціально розроблена для генерації матеріалів приватного ключа. Використовує окремий екземпляр DRBG для підвищення рівня безпеки [1].

2.2.2 Algorithm

Використовує той самий алгоритм (CTR-DRBG) що й `RAND_bytes()` але підтримує окремий стан, щоб ізолювати генерацію приватного ключа від інших операцій генерації випадкових чисел.

2.2.3 Function Signature

```
int RAND_priv_bytes(unsigned char *buf, int num);
```

2.2.4 Input/Output/Return Codes

Ідентично до `RAND_bytes()`.

2.3 RAND_seed Function

2.3.1 Description

Вручну додає ентропію до початкового значення PRNG seed. Є корисною, коли доступні додаткові джерела ентропії [4].

2.3.2 Function Signature

```
void RAND_seed(const void *buf, int num);
```

2.3.3 Input Parameters

- `buf`: Вказівник на буфер, що містить дані ентропії
- `num`: Кількість байтів ентропії

2.3.4 Return Value

Function returns void (no return code).

2.4 RAND_status Function

2.4.1 Description

Перевіряє, чи PRNG було seeded з достатньою ентропією [1].

2.4.2 Function Signature

```
int RAND_status(void);
```

2.4.3 Return Codes

- **1**: PRNG seeded з достатньою ентропією
- **0**: PRNG seeded недостатньо

3 Функції перевірки на простоту

3.1 BN_is_prime_ex Function

3.1.1 Description

Перевіряє, чи є BIGNUM ймовірно простим числом, використовуючи Miller-Rabin primality test [5, 6].

3.1.2 Algorithm

Тест Міллера-Рабіна є імовірнісним алгоритмом перевірки на простоту [7]:

Algorithm 2 Miller-Rabin Primality Test

Require: Odd integer $n > 2$, number of rounds k
Ensure: **composite** or **probably prime**

Write $n - 1$ as $2^r \cdot d$ where d is odd
for $i = 1$ to k **do**
    Choose random $a \in [2, n - 2]$
    $x \leftarrow a^d \bmod n$
    if $x = 1$ or $x = n - 1$ **then**
        continue
    end if
    for $j = 1$ to $r - 1$ **do**
        $x \leftarrow x^2 \bmod n$
        if $x = n - 1$ **then**
            continue to outer loop
        end if
    end for
    return composite
end for
return probably prime

Імовірність того, що складене число пройде k раундів, становить не більше ніж 4^{-k} [5].

3.1.3 Function Signature

```
int BN_is_prime_ex(const BIGNUM *p, int nchecks,
                   BN_CTX *ctx, BN_GENCB *cb);
```

3.1.4 Input Parameters

- `p`: BIGNUM, яке тестуватиметься на простоту
- `nchecks`: Кількість ітерацій для тесту Міллера-Рабіна (0 для автоматичного вибору)
- `ctx`: BN_CTX Деяка структура для тимчасових зміних (can be NULL)
- `cb`: Callback для моніторингу прогресу (can be NULL)

3.1.5 Output

Повертає результат of primality test.

3.1.6 Return Codes

- **1**: Число ймовірно просте
- **0**: Число точно складене
- **-1**: Сталася помилка

3.2 BN_is_prime_fasttest_ex Function

3.2.1 Description

Покращена версія функції `BN_is_prime_ex` яка виконує пробне ділення перед Miller-Rabin testing [8].

3.2.2 Algorithm

1. Trial division: Перевірка подільності на невеликі прості числа (up to 3317)
2. Якщо пробне ділення є успішним, виконується власне тест Міллера-Рабіна

   Це значно прискорює виявлення складених чисел.

3.2.3 Function Signature

```
int BN_is_prime_fasttest_ex(const BIGNUM *p, int nchecks,
                            BN_CTX *ctx, int do_trial_division,
                            BN_GENCB *cb);
```

3.2.4 Input Parameters

Такі саме як і в `BN_is_prime_ex`, додатково:

- `do_trial_division`: Якщо 1, виконати спершу пробне ділення; якщо 0 – пропустити

3.2.5 Return Codes

Такі самі, як і у `BN_is_prime_ex`.

4 Генерування простих чисел

4.1 BN_generate_prime_ex Function

4.1.1 Description

Генерує криптографічно надійне псевдовипадкове просте число [6].

4.1.2 Algorithm

Algorithm 3 Prime Number Generation

Require: Bit length $bits$, safety flag $safe$

Ensure: Prime number p

Generate random odd number p of $bits$ length
Set MSB and LSB to 1
repeat
    Perform trial division against small primes
    if divisible by small prime **then**
        $p \leftarrow p + 2$
        continue
    end if
    Apply Miller-Rabin test to p
    if p is composite **then**
        $p \leftarrow p + 2$
    else
        if $safe$ is true **then**
            Check if $(p-1)/2$ is also prime
            if $(p-1)/2$ is not prime **then**
                $p \leftarrow p + 2$
                continue
            end if
        end if
        return p
    end if
until prime found

Для ”безпечних” простих чисел (when `add` parameter is used) додаткові перевірки гарантують, що $(p-1)/2$ також є простим числом [9].

4.1.3 Function Signature

```
int BN_generate_prime_ex(BIGNUM *ret, int bits, int safe,
                         const BIGNUM *add, const BIGNUM *rem,
                         BN_GENCB *cb);
```

4.1.4 Input Parameters

- `ret`: BIGNUM structure для зберігання згенерованого простого числа
- `bits`: Бітова довжина простих чисел, що генеруються
- `safe`: Якщо 1, генерується ”безпечне” просте число, де $(p-1)/2$ також просте
- `add`: Якшо not NULL, просте число повинно задовольняти умову: p mod $add = rem$
- `rem`: Значення залишку (used with `add`)
- `cb`: Callback для моніторингу прогресу

4.1.5 Output Data

BIGNUM `ret` містить згенероване просте число.

4.1.6 Return Codes

- **1**: Success – просте число згенеровано
- **0**: Failure – сталася помилка

4.1.7 Приклад використання

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

int main() {
    BIGNUM *prime = BN_new();

    if (BN_generate_prime_ex(prime, 2048, 0, NULL, NULL, NULL) != 1) {
        fprintf(stderr, "Prime generation failed\n");
        BN_free(prime);
        return 1;
    }

    printf("Generated 2048-bit prime successfully\n");
    BN_free(prime);
    return 0;
}
```

5 Генерація RSA Key

5.1 RSA_generate_key_ex Function

5.1.1 Description

Генерує пару RSA keys pair із заданим розміром модуля та публічним показником [10, 6].

5.1.2 Algorithm

Algorithm 4 RSA Key Pair Generation

Require: Bit length $bits$, public exponent e

Ensure: RSA key pair $(n, e, d, p, q, dP, dQ, qInv)$

1: Generate random prime p of $bits/2$ length
2: Generate random prime q of $bits/2$ length, $q \neq p$
3: Compute modulus $n \leftarrow p \times q$
4: Compute Euler's totient $\phi(n) \leftarrow (p-1)(q-1)$
5: Verify $\gcd(e, \phi(n)) = 1$
6: Compute private exponent $d \leftarrow e^{-1} \bmod \phi(n)$
7: Compute CRT parameter $dP \leftarrow d \bmod (p-1)$
8: Compute CRT parameter $dQ \leftarrow d \bmod (q-1)$
9: Compute CRT parameter $qInv \leftarrow q^{-1} \bmod p$
10: **return** $(n, e, d, p, q, dP, dQ, qInv)$

5.1.3 Function Signature

```
int RSA_generate_key_ex(RSA *rsa, int bits, BIGNUM *e,
                        BN_GENCB *cb);
```

5.1.4 Input Parameters

- `rsa`: RSA structure для зберігання згенерованого ключа
- `bits`: Довжина модуля в бітах (зазвичай це 2048, 3072, або 4096)
- `e`: Публічна експонента BIGNUM (часто дорівнює 65537 = $2^{16} + 1$)
- `cb`: Callback для моніторингу прогресу

5.1.5 Output Data

Структура RSA задається наступним чином:

- Public key: (n, e)
- Private key: (n, d) і додатково параметри $(p, q, dP, dQ, qInv)$

5.1.6 Return Codes

- **1**: Success – пара ключів згенерована
- **0**: Failure – сталася помилка

5.1.7 Приклад використання

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

int main() {
    RSA *rsa = RSA_new();
    BIGNUM *e = BN_new();
    BN_set_word(e, RSA_F4); // e = 65537

    if (RSA_generate_key_ex(rsa, 2048, e, NULL) != 1) {
        fprintf(stderr, "RSA key generation failed\n");
        RSA_free(rsa);
        BN_free(e);
        return 1;
    }

    printf("Generated 2048-bit RSA key pair successfully\n");

    // Cleanup
    RSA_free(rsa);
    BN_free(e);
    return 0;
}
```

6 Генерація DSA Key

6.1 DSA_generate_parameters_ex Function

6.1.1 Description

Генерує DSA доменні параметри (p, q, g) згідно до FIPS 186-4 [11].

6.1.2 Algorithm

Використовує алгоритм, визначений в FIPS 186-4 [11]:

1. Згенерувати просте число q із заданою бітовою довжиною (зазвичай 160, 224, або 256 бітів)
2. Згенерувати просте число p таке, що q ділиить $(p-1)$ і p має необіхдну бітову довжину
3. Знайти генератор g порядку q в полі $\mathbb{Z}_p^*$: вибрати $h \in [2, p-2]$ і обчислювати $g = h^{(p-1)/q} \bmod p$, доки це $g > 1$

6.1.3 Function Signature

```c
int DSA_generate_parameters_ex(DSA *dsa, int bits,
                               const unsigned char *seed,
                               int seed_len, int *counter_ret,
                               unsigned long *h_ret,
                               BN_GENCB *cb);
```

6.1.4 Input Parameters

- `dsa`: DSA structure для зберігання параметрів
- `bits`: Бітова довжина числа p (1024, 2048, або 3072)
- `seed`: Опціональний seed для генерації (can be NULL)
- `seed_len`: Довжина of seed (в бітах)
- `counter_ret`: Вказівник для зберігання лічильника генерації (can be NULL)
- `h_ret`: Вказівник для зберігання h, використаного при генерації (can be NULL)
- `cb`: Callback для моніторингу прогресу

6.1.5 Return Codes

- **1**: Success
- **0**: Failure

6.2 DSA_generate_key Function

6.2.1 Description

Генерує DSA public/private key pair використовуючи існуючі доменні параметри [11].

6.2.2 Algorithm

1. Згенерувати випадковий private key: $x \in [1, q-1]$
2. Згенерувати public key: $y = g^x \bmod p$

6.2.3 Function Signature

```
int DSA_generate_key(DSA *dsa);
```

6.2.4 Input Parameters

- `dsa`: DSA structure, що містить доменні параметри

6.2.5 Output Data

DSA structure, що заповнена приватним ключем x та публічним ключем y.

6.2.6 Return Codes

- **1**: Success
- **0**: Failure

7 Генерація Elliptic Curve Key

7.1 EC_KEY_generate_key Function

7.1.1 Description

Генерує пару ключів еліптичної кривої для обраної кривої [12, 13].

7.1.2 Algorithm

1. Генерується випадковий private key: $d \in [1, n-1]$ де n – порядок кривої
2. Обчислюється точка публічного ключа, така що: $Q = d \cdot G$ де G – генеративна точка, з використанням операції множення точок еліптичної кривої

Безпека криптографії з еліптичними кривими базується на проблемі взяття дискретного логарифму на еліптичній кривій (ECDLP) [14].

7.1.3 Function Signature

```
int EC_KEY_generate_key(EC_KEY *key);
```

7.1.4 Input Parameters

- `key`: EC_KEY structure з набором параметрів кривої

7.1.5 Output Data

EC_KEY structure заповнена за допомогою private key scalar і public key point.

7.1.6 Return Codes

- **1**: Success
- **0**: Failure

7.1.7 Приклад використання

```
#include <openssl/ec.h>
#include <openssl/obj_mac.h>

int main() {
    // Create EC_KEY structure for secp256k1 curve
    EC_KEY *key = EC_KEY_new_by_curve_name(NID_secp256k1);

    if (key == NULL) {
        fprintf(stderr, "Failed to create EC_KEY\n");
        return 1;
    }

    if (EC_KEY_generate_key(key) != 1) {
        fprintf(stderr, "EC key generation failed\n");
```

```
        EC_KEY_free(key);
        return 1;
    }

    printf("Generated EC key pair successfully\n");

    // Cleanup
    EC_KEY_free(key);
    return 0;
}
```

8 Аналіз часової ефективності

8.1 Продуктивність PRNG

`RAND_bytes()` в операційній системі Windows використовує CTR-DRBG з AES-256 [3]:

- Типова пропускна здатність: 200–500 MB/s на сучасних CPUs
- Підтримка інструкцій AES-NI значно покращує продуктивність (up to 2–3 GB/s)
- Накладні витрати на reseeding: приблизно 1–2 ms на кожні 2^{48} згенерованих байтів
- Незначний вплив на продуктивність для типових операцій генерації ключів

8.2 Продуктивність тесту на простоту

Для алгоритму Міллера-Рабіна з пробним діленням (`BN_is_prime_fasttest_ex`) [8]:

- Числа довжини 1024-bit: 1–5 ms (зазвичай в середньому: 2 ms)
- Числа довжини 2048-bit: 10–50 ms (зазвичай в середньому: 25 ms)
- Числа довжини 4096-bit: 100–500 ms (зазвичай в середньому: 250 ms)

Продуктивність сильно залежить від заданої кількості ітерацій і можливостей процесора. Пробне ділення виключає приблизно 80–90% складених кандидатів перед самим тестуванням Міллера-Рабіна.

8.3 Продуктивність генерації простих чисел

Середній час для `BN_generate_prime_ex` [6]:

- 1024-bit просте число: 50–200 ms (зазвичай в середньому: 100 ms)
- 2048-bit просте число: 500–2000 ms (зазвичай в середньому: 1000 ms)
- 4096-bit просте число: 5–20 seconds (зазвичай в середньому: 10 seconds)

Генерація ”безпечних” простих чисел займає значно більше часу (в 10–100 разів) залежно до вимог на p та $(p-1)/2$. Вони обидва мають бути простими.

8.4 Продуктивність генерації RSA key

Продуктивність `RSA_generate_key_ex` [10]:

- Ключ довжини 2048-bit: 100–500 ms (зазвичай в середньому: 250 ms)
- Ключ довжини 3072-bit: 500–2000 ms (зазвичай в середньому: 1000 ms)
- Ключ довжини 4096-bit: 2–10 seconds (зазвичай в середньому: 5 seconds)

Більше 90% часу витрачається на генерацію простих p та q. Знаходження обереного за модулем для приватної експоненти d є відносно швидкою операцією.

8.5 Продуктивність генерації DSA Key

- Генерація параметрів (1024-bit p, 160-bit q): 1–5 seconds
- Генерація параметрів (2048-bit p, 256-bit q): 5–30 seconds
- Генерація пари (за заданими параметрами): <10 ms

8.6 Продуктивність генерації ECC Key

- secp256r1 (NIST P-256): 1–3 ms
- secp384r1 (NIST P-384): 3–8 ms
- secp521r1 (NIST P-521): 8–20 ms

Генерація ECC key є значно швидшою за RSA для порівнюваних рівнів безпеки [12].

9 Опис стабільності та безпеки

9.1 Стабільність PRNG

Реалізація OpenSSL's в PRNG вважається стабільною та безпечною, коли [2]:

- Операційна система забезпечує достатні джерела ентропії
- `RAND_status()` повернув 1 перед генерацією ключів
- Не було внесено змін до внутрішнього стану PRNG
- Бібліотека скомпільована з належними механізмами збору ентропії

9.2 Windows Platform

У Windows OpenSSL використовується [15]:

- `CryptGenRandom` API (Windows XP–10) or `BCryptGenRandom` (Windows 10+) для збору ентропії
- Інструкції RDRAND/RDSEED CPU якщо доступні модифікації (Intel Ivy Bridge+, AMD Ryzen+)
- Лічильники продуктивності системи як додаткове джерело ентропії
- Ідентифікатори(IDs) процесів і потоків, часові мітки з високою роздільною здатністю

Джерела ентропії Windows вважаються криптографічно безпечними для цілей генерації ключів [16].

9.3 Найкращі поради-практики з безпеки

1. **Розмір ключів**: Використовуйте мінімум 2048-bit RSA ключ (еквівалентний до 112-bit рівня безпеки), 256-bit ECC (еквівалентний до 128-bit рівня безпеки) [17]
2. **Використання PRNG**: Завжди використовуйте `RAND_priv_bytes()` для генерації даних для private key
3. **Обробка помилок**: Перевіряйте Return Codes для всіх функцій OpenSSL; Не продовжуйте, якщо виникають помилки
4. **Перевірка ентропії**: Підтвердіть PRNG статус перед генерацією ключа: `RAND_status() == 1`
5. **Безпека пам'яті**: Видаляйте конфіденційні матеріали ключів з пам'яті після використання за допомогою функції `OPENSSL_cleanse()`
6. **Оновлення бібліотеки**: Keep OpenSSL up to date для отримування найновіших патчів безпеки

9.4 Поширені імплементаційні проблеми

- **Недостатня ентропія**: У віртуальних або вбудованих системах джерела ентропії можуть бути обмеженими
- **Безпека розгалуження**: Після `fork()`, дочірні процеси повинні перезапустити PRNG, щоб уникнути дублювання випадкових послідовностей
- **Безпека потоків**: OpenSSL 1.1.0+ є безпечним для потоків за замовчуванням; попередні версії вимагають явного блокування
- **Memory Leaks**: Завжди звільняйте виділені структури `BN_free()`, `RSA_free()`, `EC_KEY_free()`

10 Порівняльний аналіз

10.1 Придатність алгоритму для генерації ключів

Algorithm	Key Gen Time	Security/Bit	Suitability
RSA-2048	250 ms	Moderate	High
RSA-3072	1000 ms	High	High
RSA-4096	5000 ms	Very High	Medium
DSA-2048	10000 ms	High	Medium
DSA-3072	20000 ms	Very High	Medium
ECC-256	2 ms	High	Very High
ECC-384	5 ms	Very High	Very High
ECC-521	15 ms	Extreme	High

Таблиця 1: Порівняння алгоритмів генерації ключів

Для сучасних додатків ECC забезпечує найкращий баланс між безпекою та продуктивністю [12]. В той час як RSA залишається широко використовуваним завдяки сумісності та налагодженій інфраструктурі [9].

10.2 Порівняння алгоритмів PRNG

Реалізація в OpenSSL's CTR-DRBG має переваги над альтернативними PRNGs, оскільки:

- **Безпека**: Базується на алгоритмі затвердженому в NIST з формальним аналізом безпеки
- **Продуктивність**: Апаратне прискорення AES забезпечує чудову пропускну здатність
- **Стійкість до передбачуваності:**: Пряма пряма секретність завдяки періодичному reseeding
- **Стійкість до зворотного відстеження**: Неможливість виведення попередніх результатів з поточного стану
- **Стандартизація**: Реалізація, що відповідає стандарту FIPS 140-2

11 Приклади реалізацій

11.1 Повне генерування ключів RSA з обробкою помилок

```c
#include <openssl/rsa.h>
#include <openssl/bn.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <stdio.h>

int generate_rsa_keypair(const char *public_key_file,
                         const char *private_key_file) {
    RSA *rsa = NULL;
    BIGNUM *e = NULL;
    FILE *fp = NULL;
    int ret = 0;

    // Check PRNG status
    if (RAND_status() != 1) {
        fprintf(stderr, "PRNG not sufficiently seeded\n");
        return 0;
    }

    // Initialize structures
    rsa = RSA_new();
    e = BN_new();

    if (!rsa || !e) {
        fprintf(stderr, "Memory allocation failed\n");
        goto cleanup;
    }

    // Set public exponent to 65537
    if (BN_set_word(e, RSA_F4) != 1) {
        fprintf(stderr, "Failed to set public exponent\n");
        goto cleanup;
    }

```

```
    // Generate 2048-bit RSA key pair
    printf("Generating 2048-bit RSA key pair...\n");
    if (RSA_generate_key_ex(rsa, 2048, e, NULL) != 1) {
        fprintf(stderr, "RSA key generation failed\n");
        ERR_print_errors_fp(stderr);
        goto cleanup;
    }

    // Save public key
    fp = fopen(public_key_file, "wb");
    if (!fp) {
        fprintf(stderr, "Cannot open public key file\n");
        goto cleanup;
    }

    if (PEM_write_RSAPublicKey(fp, rsa) != 1) {
        fprintf(stderr, "Failed to write public key\n");
        goto cleanup;
    }
    fclose(fp);
    fp = NULL;

    // Save private key
    fp = fopen(private_key_file, "wb");
    if (!fp) {
        fprintf(stderr, "Cannot open private key file\n");
        goto cleanup;
    }

    if (PEM_write_RSAPrivateKey(fp, rsa, NULL, NULL, 0,
                                NULL, NULL) != 1) {
        fprintf(stderr, "Failed to write private key\n");
        goto cleanup;
    }

    printf("Key pair generated successfully\n");
    ret = 1;

cleanup:
    if (fp) fclose(fp);
    if (rsa) RSA_free(rsa);
    if (e) BN_free(e);
    return ret;
}

int main() {
    return generate_rsa_keypair("public.pem", "private.pem") ? 0 : 1;
}
```

11.2 Генерація ключів ECC з декількома еліптичними кривими

```c
#include <openssl/ec.h>
#include <openssl/obj_mac.h>
#include <openssl/pem.h>
#include <stdio.h>

typedef struct {
    int nid;
    const char *name;
} curve_info_t;

int generate_ec_key(int curve_nid, const char *filename) {
    EC_KEY *key = NULL;
    FILE *fp = NULL;
    int ret = 0;

    // Create EC_KEY for specified curve
    key = EC_KEY_new_by_curve_name(curve_nid);
    if (!key) {
        fprintf(stderr, "Failed to create EC_KEY\n");
        return 0;
    }

    // Generate key pair
    if (EC_KEY_generate_key(key) != 1) {
        fprintf(stderr, "EC key generation failed\n");
        EC_KEY_free(key);
        return 0;
    }

    // Verify key
    if (EC_KEY_check_key(key) != 1) {
        fprintf(stderr, "EC key verification failed\n");
        EC_KEY_free(key);
        return 0;
    }

    // Save to file
    fp = fopen(filename, "wb");
    if (!fp) {
        fprintf(stderr, "Cannot open file\n");
        EC_KEY_free(key);
        return 0;
    }

    if (PEM_write_ECPrivateKey(fp, key, NULL, NULL, 0,
                               NULL, NULL) == 1) {
        ret = 1;
    }

```

```
    fclose(fp);
    EC_KEY_free(key);
    return ret;
}

int main() {
    curve_info_t curves[] = {
        {NID_secp256k1, "secp256k1"},
        {NID_X9_62_prime256v1, "secp256r1"},
        {NID_secp384r1, "secp384r1"},
        {NID_secp521r1, "secp521r1"}
    };

    for (int i = 0; i < 4; i++) {
        char filename[64];
        snprintf(filename, sizeof(filename), "ec_%s.pem",
                curves[i].name);

        printf("Generating key for curve %s...\n", curves[i].name);
        if (generate_ec_key(curves[i].nid, filename)) {
            printf("Success: %s\n", filename);
        } else {
            printf("Failed: %s\n", curves[i].name);
        }
    }

    return 0;
}
```

11.3 Генерація простих чисел зі зворотнім викликом прогресу

```
#include <openssl/bn.h>
#include <stdio.h>

int prime_callback(int p, int n, BN_GENCB *cb) {
    char c = '*';

    if (p == 0) c = '.';      // Starting search
    if (p == 1) c = '+';      // Found candidate
    if (p == 2) c = '*';      // Passed primality test
    if (p == 3) c = '\n';     // Generation complete

    putchar(c);
    fflush(stdout);
    return 1;
}

int main() {
    BIGNUM *prime = BN_new();
    BN_GENCB *cb = BN_GENCB_new();
```

```
    if (!prime || !cb) {
        fprintf(stderr, "Allocation failed\n");
        return 1;
    }

    // Set up callback
    BN_GENCB_set(cb, prime_callback, NULL);

    printf("Generating 2048-bit prime number:\n");

    if (BN_generate_prime_ex(prime, 2048, 0, NULL, NULL, cb) != 1) {
        fprintf(stderr, "Prime generation failed\n");
        BN_free(prime);
        BN_GENCB_free(cb);
        return 1;
    }

    // Print the prime in hexadecimal
    char *prime_hex = BN_bn2hex(prime);
    printf("\nGenerated prime:\n%s\n", prime_hex);

    // Cleanup
    OPENSSL_free(prime_hex);
    BN_free(prime);
    BN_GENCB_free(cb);

    return 0;
}
```

12 Benchmarking Results

12.1 Тестове середовище

Тести проводилися на:

- **OS**: Windows 10 Professional (64-bit)
- **CPU**: Intel Core i7-1065G @ 1.5 GHz
- **RAM**: 16 GB DDR4
- **OpenSSL Version**: 3.3.0
- **Compiler**: Microsoft Visual Studio 2019

12.2 Тести на пропускну здатність PRNG

Пропускна здатність збільшується із розміром буфера завдяки зменшенню накладних витрат на виклик функції та кращому використанню кешу процесора:

Buffer Size	Throughput (MB/s)	Latency
16 bytes	45.2	0.35 µs
256 bytes	312.5	0.82 µs
4 KB	1,024.0	3.91 µs
64 KB	2,457.6	26.05 µs
1 MB	3,145.7	327.68 µs

Таблиця 2: Пропускна здатність RAND_bytes

12.3 Продуктивність тестування на простоту

Bit Length	No Trial Div.	With Trial Div.	Speedup
512 bits	0.8 ms	0.3 ms	2.67×
1024 bits	3.2 ms	1.8 ms	1.78×
2048 bits	28.5 ms	24.1 ms	1.18×
4096 bits	312.7 ms	286.3 ms	1.09×

Таблиця 3: Середній час тесту (Міллера-Рабіна)

Пробне ділення забезпечує значне прискорення виконання для менших чисел, але зменшує ефективність для більших значень.

12.4 Тести генерації ключів

Algorithm	Min (ms)	Avg (ms)	Max (ms)
RSA-2048	187	243	421
RSA-3072	724	981	1,653
RSA-4096	3,156	4,872	8,234
ECC-256	1.2	1.8	3.4
ECC-384	3.7	5.2	8.9
ECC-521	11.3	15.7	24.6
DSA-2048	8,234	11,457	19,821

Таблиця 4: Час генерації ключів (100 ітерацій)

Висока варіативність часу генерації RSA та DSA пояснюється ймовірнісним характером пошуку простих чисел.

13 Conclusion

Бібліотека OpenSSL надає надійні, добре перевірені реалізації алгоритмів PRNG, методів перевірки простоти та функцій генерації ключів, придатних для виробничих криптографічних додатків. Ключові висновки:

13.1 Key Findings

1. **Якість PRNG**: Реалізація CTR-DRBG з AES-256 забезпечує криптографічно безпечні випадкові числа з чудовою пропускною здатністю (>3 ГБ/с з AES-NI) і відповідає вимогам NIST SP 800-

90A [3].

2. **Ефективність перевірки на простоту**: Поєднання пробного ділення та тестування Міллера-Рабіна забезпечує оптимальну продуктивність, швидко усуваючи більшість складених чисел, зберігаючи при цьому високу впевненість у простоті [8].

3. **Придатність алгоритму**: Для нових реалізацій ECC пропонує найкраще співвідношення продуктивності та безпеки, з генерацією ключів, що в 100–500 разів швидша, ніж ключі RSA з еквівалентною безпекою [12].

4. **Стабільність платформи**: OpenSSL на Windows демонструє стабільну продуктивність при правильній конфігурації з джерелами системної ентропії (CryptGenRandom/BCryptGenRandom).

5. **Стадія реалізації**: Всі протестовані функції демонструють стабільну поведінку та належне оброблення помилок, а також вичерпну документацію, що робить їх придатними для генерації ключів асиметричної криптосистеми.

13.2 Рекомендації для практичного застосування

- Використовуйте **ECC-256 or ECC-384** для нових додатків, що вимагають оптимальної продуктивності
- Використовуйте **RSA-2048 or RSA-3072** коли потрібна сумісність з існуючою інфраструктурою
- Завжди обов'язково перевіряйте стан ініціалізації PRNG перед генерацією ключів
- Впровадьте комплексну обробку помилок для всіх викликів функцій OpenSSL
- Розглядайте генерування "безпечних" простих чисел тільки тоді, коли це конкретно вимагається протоколом (через витрати на продуктивність).
- Використовуйте `RAND_priv_bytes()` замість `RAND_bytes()` для складових приватного ключа
- Вмикайте інструкцію AES-NI для CPU задля оптимальної продуктивності алгоритму генерації PRNG

13.3 Напрямки потенційних майбутніх досліджень

Подальші дослідження можуть бути спрямовані на:

- Реалізація постквантової криптографії в OpenSSL
- Аналіз продуктивності інтеграційного hardware security module (HSM)
- Порівняльний аналіз з альтернативними криптографічними бібліотеками
- Аналіз показників енергоефективності для вбудованих і мобільних платформ
- Стікість до атак по бічних каналах (side-channel attack) у реалізаціях генерації ключів

Бібліотека OpenSSL продовжує розвиватися, а поточна розробка зосереджена на постквантових алгоритмах, покращенні продуктивності та посиленні функцій безпеки для сучасних криптографічних вимог.

References

[1] OpenSSL Software Foundation. *OpenSSL Cryptography and SSL/TLS Toolkit*. Accessed: 2024. 2023. URL: https://www.openssl.org/docs/.

[2] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL: Cryptography for Secure Communications*. O'Reilly Media, 2002. ISBN: 978-0596002701.

[3] Elaine Barker and John Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Tech. rep. NIST Special Publication 800-90A Revision 1. Gaithersburg, MD: National Institute of Standards and Technology, 2015. DOI: 10.6028/NIST.SP.800-90Ar1.

[4] Donald Eastlake, Jeff Schiller, and Steve Crocker. *Randomness Requirements for Security*. Tech. rep. 4086. Internet Engineering Task Force, June 2005. DOI: 10.17487/RFC4086. URL: https://www.rfc-editor.org/info/rfc4086.

[5] Michael O. Rabin. «Probabilistic Algorithm for Testing Primality». In: *Journal of Number Theory* 12.1 (1980), pp. 128–138. DOI: 10.1016/0022-314X(80)90084-0.

[6] 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/.

[7] Gary L. Miller. «Riemann's Hypothesis and Tests for Primality». In: *Journal of Computer and System Sciences* 13.3 (1976), pp. 300–317. DOI: 10.1016/S0022-0000(76)80043-8.

[8] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. 2nd. Springer, 2005. ISBN: 978-0387252827.

[9] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 20th Anniversary. John Wiley & Sons, 2015. ISBN: 978-1119096726.

[10] 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.

[11] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. Tech. rep. FIPS PUB 186-4. Gaithersburg, MD: U.S. Department of Commerce, 2013. DOI: 10.6028/NIST.FIPS.186-4.

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

[13] Neal Koblitz. «Elliptic Curve Cryptosystems». In: *Mathematics of Computation*. Vol. 48. 177. American Mathematical Society, 1987, pp. 203–209. DOI: 10.2307/2007884.

[14] 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.

[15] Microsoft Corporation. *Cryptography API: Next Generation*. Windows Developer Documentation. 2021. URL: https://docs.microsoft.com/en-us/windows/win32/seccng/cng-portal.

[16] Peter Gutmann. «Software Generation of Practically Strong Random Numbers». In: *Proceedings of the 7th USENIX Security Symposium* (1998), pp. 243–257.

[17] 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.

A Compilation Instructions

Для компіляції прикладних програм використовуються такі команди:

A.1 Windows with MSVC

```
cl /I"C:\OpenSSL\include" example.c /link
   /LIBPATH:"C:\OpenSSL\lib" libcrypto.lib
```

A.2 Windows with MinGW

```
gcc -o example example.c -I/c/OpenSSL/include
    -L/c/OpenSSL/lib -lcrypto
```

A.3 Cross-platform with CMake

Create `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.10)
project(OpenSSL_Examples)

find_package(OpenSSL REQUIRED)

add_executable(rsa_example rsa_example.c)
target_link_libraries(rsa_example OpenSSL::Crypto)

add_executable(ecc_example ecc_example.c)
target_link_libraries(ecc_example OpenSSL::Crypto)
```

Далі власне build:

```
mkdir build && cd build
cmake ..
cmake --build .
```

B Additional Resources

B.1 Official Documentation

- OpenSSL Manual Pages: https://www.openssl.org/docs/
- OpenSSL Wiki: https://wiki.openssl.org/
- OpenSSL GitHub: https://github.com/openssl/openssl

B.2 Standards Documents

- NIST SP 800-90A: DRBG Specifications
- FIPS 186-4: Digital Signature Standard

- RFC 8017: PKCS #1 RSA Cryptography Specifications
- RFC 5639: ECC Brainpool Standard Curves