Lab_1 MRKM

Порівняння бібліотек OpenSSL, Crypto++, cryptlib та PyCrypto для розробки гібридної криптосистеми на платформі Windows

Гібридна криптосистема зазвичай поєднує асиметричні алгоритми (наприклад, RSA або ECDSA для обміну ключами) з симетричними (наприклад, AES для шифрування даних) та хеш-функціями (наприклад, SHA-256). Нижче наведено порівняння зазначених бібліотек з точки зору ефективності за часом (швидкість виконання операцій) та пам'яттю (використання ресурсів). Дані базуються на бенчмарках з відкритих джерел (2008–2022 роки), включаючи тести на симетричні/асиметричні примітиви, SSL/TLS операції та загальну продуктивність. Тести проводилися на різних платформах, включаючи Windows, де результати подібні до Linux через крос-платформовість бібліотек.

Бібліотеки тестувалися на апаратному прискоренні (наприклад, AES-NI на Intel/AMD), яке доступне на Windows. PyCrypto є застарілою (остання версія 2013 року) — рекомендується заміна на PyCryptodome або cryptography.io для сучасних проектів.

Загальні характеристики бібліотек

Бібліотека	Мова програмування	Ліцензія	Підтримка алгоритмів (ключові для гібридної системи)	Апаратне прискорення	Розмір коду (SLOC)
OpenSSL	C	Apache 2.0	RSA, ECDSA, AES, SHA-2/3, HMAC, GCM/CCM; повна підтримка гібридних схем (PKCS#1/8/12)	AES-NI, AVX/AVX2, RDRAND, ARM/Power ISA	472k
Crypto++	C++	Boost (публічний домен)	RSA, ECDSA, AES, SHA-2/3, HMAC, GCM/EAX; модульна підтримка ECC (NIST, Curve25519)	AES-NI, AVX/AVX2, RDRAND, ARM/Power ISA	115k
cryptlib	C	Sleepycat (або комерційна)	RSA, DSA, AES, SHA-2/3,	AES-NI, AVX/AVX2,	241k

Бібліотека	Мова програмування	Ліцензія	Підтримка алгоритмів (ключові для гібридної системи)	Апаратне прискорення	Розмір коду (SLOC)
			HMAC, GCM; базова підтримка PKCS#1/8/12	RDRAND	
PyCrypto	Python	Публічний домен	RSA, AES, SHA-1/2, HMAC; обмежена підтримка ECC	Залежить від Python (частково через C-розширення)	~50k (але з Python-оверхедом)

Ефективність за часом (швидкість)

- Бенчмарки фокусуються на ключових примітивах: симетричне шифрування (AES/Blowfish), асиметричне (RSA/ECDSA), хешування (SHA) та SSL/TLS-операції (для гібридних сценаріїв).
- Одиниці: KB/s для симетричних/хешів, операцій/с для асиметричних. Тести на Intel Pentium/Core (2008–2022), де OpenSSL часто лідирує завдяки оптимізованому асемблеру.

Бібліотека	Симетричне шифрування (наприклад, AES-128, KB/s)	Асиметричне (наприклад, RSA-2048 sign/verify, оп/с)	Хешування (наприклад, SHA-256, KB/s)	Загальна продуктивність у SSL/TLS (KB/s)	Примітки
OpenSSL	Висока: 45,461 (AES), 56,510 (Blowfish)	Висока: ~1350 sign/s, ~26,760 verify/s	Висока: ~15,711	Найвища: ~33,359	Лідер у швидкості; ~0.6 cpb для AES. Оптимізовано для Windows з AES-NI.
Crypto++	Середня: 27,111 (AES), 50,407 (Blowfish)	Середня: Повільніше OpenSSL на 20–30%	Середня: Близько до OpenSSL, але ~5.7 cpb для RNG	Середня: Повільніше OpenSSL на 15–37%	Конкурентна з Botan; добре для C++-проектів, але загальний дизайн уповільнює.

Бібліотека	Симетричне шифрування (наприклад, AES-128, KB/s)	Асиметричне (наприклад, RSA-2048 sign/verify, оп/с)	Хешування (наприклад, SHA-256, KB/s)	Загальна продуктивність у SSL/TLS (KB/s)	Примітки
cryptlib	Низька: ~9,671 (AES-GCM/ChaCha)	Низька: ~1493 sign/s, ~27,901 verify/s	Низька: ~14,376	Найнижча: ~9,671	Підходить для легких завдань; низька швидкість у гібридних операціях.
PyCrypto	Низька: Швидше для малих даних (~1.5x vs cryptography для 32B AES), повільніше для великих (~0.1x для 1MiB)	Низька: Значно повільніше C/C++ (фактор 10x+)	Низька: Швидше для малих (~7x vs cryptography для SHA), повільніше для великих (~0.1x)	Не оптимізована для SSL; повільна	Інтерпретована мова уповільнює; добре для прототипів, але не для продакшну.

- **Висновки за часом**: OpenSSL — найкращий вибір для високопродуктивних гібридних систем (наприклад, серверів на Windows). Crypto++ підходить для об'єктно-орієнтованих проектів. cryptlib та PyCrypto повільніші, особливо для великих даних або інтенсивних операцій.

Ефективність за пам'яттю (використання ресурсів)

- Одиниці: Runtime memory (MB/KB на сесію/операцію). Дані з бенчмарків та документації; варіюється залежно від конфігурації (наприклад, включені модулі).
- OpenSSL та Crypto++ мають більший оверхед через багатство функцій; cryptlib — компактніша; PyCrypto страждає від Python-runtime (~50–100MB базово).

Бібліотека	Runtime memory (на сесію/операцію)	Бінарний розмір (компіляція)	Примітки
OpenSSL	Висока: 1–10 MB (для TLS-сесії)	3–20 MB	Великий через оптимізації; не для обмежених пристроїв.
Crypto++	Середня: ~500 KB–2 MB	~2–5 MB (модульний)	Модульний дизайн дозволяє зменшити; менше ніж OpenSSL.
cryptlib	Низька: ~500 KB–1 MB	~1–3 MB	Компактна, портативна; низьке використання CPU (~91%).
PyCrypto	Висока: 50–200 MB (з Python-оверхедом)	Залежить від Python (~100 MB)	Великий оверхед інтерпретатора; не ефективна для Windows-додатків з

Бібліотека	Runtime memory (на сесію/операцію)	Бінарний розмір (компіляція)	Примітки
			обмеженою пам'яттю.

- **Висновки за пам'яттю**: cryptlib — найефективніша для обмежених ресурсів. OpenSSL підходить для десктопних Windows-додатків, але не для вбудованих. PyCrypto неефективна через Python.

Приклад реалізації гібридної криптосистеми на Python з використанням PyCryptodome

Зверніть увагу, що PyCrypto є застарілою бібліотекою (остання версія з 2013 року) і не рекомендується для нових проектів через вразливості та відсутність підтримки. Замість неї використовуйте **PyCryptodome**, яка є повноцінним форком PyCrypto з покращеннями, сумісністю та підтримкою сучасних Python-версій (3.7+). Синтаксис майже ідентичний, тому код легко адаптувати.

Гібридна криптосистема:

- Асиметричне шифрування (RSA з PKCS#1 OAEP) для обміну сесійним ключем.
- Симетричне шифрування (AES в режимі EAX для конфіденційності та аутентифікації) для даних.

Щоб запустити код, спочатку встановіть PyCryptodome: pip install pycryptodome.

1. Генерація пари RSA-ключів (2048 біт)

Цей код генерує приватний і публічний ключі та зберігає їх у файлах private.pem (приватний) та receiver.pem (публічний для отримувача).

Python

```
from Crypto.PublicKey import RSA

# Генерація ключів
key = RSA.generate(2048)

# Збереження приватного ключа
private_key = key.export_key()
with open("private.pem", "wb") as f:
    f.write(private_key)

# Збереження публічного ключа
public_key = key.publickey().export_key()
with open("receiver.pem", "wb") as f:
    f.write(public_key)

print("Ключі згенеровано та збережено в файлах private.pem та receiver.pem.")
```

2. Шифрування даних (відправник)

Використовує публічний ключ отримувача для шифрування сесійного ключа AES, а потім шифрує дані. Результат зберігається в encrypted_data.bin.

Python

```
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP

# Дані для шифрування
data = "Секретне повідомлення: Це тест гібридної криптосистеми.".encode("utf-8")

# Імпорт публічного ключа отримувача
recipient_key = RSA.import_key(open("receiver.pem").read())

# Генерація сесійного ключа AES (16 байт для AES-128)
session_key = get_random_bytes(16)

# Шифрування сесійного ключа за допомогою RSA з OAEP
cipher_rsa = PKCS1_OAEP.new(recipient_key)
enc_session_key = cipher_rsa.encrypt(session_key)

# Шифрування даних за допомогою AES в режимі EAX
cipher_aes = AES.new(session_key, AES.MODE_EAX)
ciphertext, tag = cipher_aes.encrypt_and_digest(data)

# Збереження зашифрованих даних у файл
with open("encrypted_data.bin", "wb") as f:
    [f.write(x) for x in (enc_session_key, cipher_aes.nonce, tag, ciphertext)]

print("Дані зашифровано та збережено в encrypted_data.bin.")
```

3. Дешифрування даних (отримувач)

Використовує приватний ключ для дешифрування сесійного ключа, а потім розшифровує дані.

Python

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import AES, PKCS1_OAEP

# Імпорт приватного ключа
private_key = RSA.import_key(open("private.pem").read())

# Читання зашифрованих даних з файлу
with open("encrypted_data.bin", "rb") as f:
    enc_session_key = f.read(private_key.size_in_bytes())
    nonce = f.read(16)
    tag = f.read(16)
    ciphertext = f.read(-1)

# Дешифрування сесійного ключа за допомогою RSA з OAEP
cipher_rsa = PKCS1_OAEP.new(private_key)
session_key = cipher_rsa.decrypt(enc_session_key)

# Дешифрування даних за допомогою AES в режимі EAX
cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce=nonce)
data = cipher_aes.decrypt_and_verify(ciphertext, tag)
```

```
print("Розшифровані дані:", data.decode("utf-8"))
```

Пояснення ефективності

- **Час**: Для малих даних (як у прикладі) операції швидкі (<1 сек на стандартному ПК). RSA повільніше для великих ключів, але використовується тільки для сесійного ключа. AES швидкий завдяки апаратному прискоренню (наприклад, AES-NI на Windows).
- **Пам'ять**: PyCryptodome ефективна — використовує ~50-200 KB на операцію, плюс оверхед Python (~50 MB базово). Для великих даних масштабується добре.
- **Тестування**: Запустіть код послідовно: спочатку генерацію ключів, потім шифрування, дешифрування. Якщо потрібно, додайте бенчмарки з timeit для вимірювання часу.

Цей код демонструє базові криптографічні примітиви для гібридної системи. Якщо потрібно адаптувати під PyCrypto (старіша версія), замініть імпорти на from Crypto... (без 'dome'), але уникайте в продакшні. Якщо є додаткові вимоги (наприклад, хешування або інші алгоритми), дайте знати!