Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна

ПРОГРАМУВАННЯ КРИПТОГРАФІЧНИХ ПРИМІТИВІВ

Методичні вказівки
до лабораторного практикуму

Електронний ресурс

Харків – 2026

УДК 004.75:004.056.55
П 78

Рецензенти:

Л. В. Ковальчук – доктор технічних наук, професор, провідний науковий співробітник відділу математичного і економетричного моделювання Інституту проблем моделювання в енергетиці імені Г. Є. Пухова;

Р. В. Олійников – доктор технічних наук, професор, професор кафедри кібербезпеки інформаційних систем, мереж і технологій Харківського національного університету імені В. Н. Каразіна.

Затверджено до розміщення в мережі Інтернет рішенням Науково-методичної ради Харківського національного університету імені В. Н. Каразіна (протокол № 8 від 21 травня 2026 року)

Програмування криптографічних примітивів : методичні вказівки до лабораторного практикуму [Електронний ресурс] / уклад. М. Ю. Родінко. – Харків : ХНУ імені В. Н. Каразіна, 2026. – (PDF 59 с.)

П 78

Дані методичні вказівки до лабораторного практикуму присвячені практичним аспектам реалізації та аналізу ефективності базових криптографічних примітивів мовою Java. У межах лабораторного практикуму розглядаються підходи до реалізації як симетричних перетворень (блоковий шифр AES у різних режимах роботи, функції гешування SHA-2 та SHA-3), так і асиметричних примітивів (алгоритм направленого шифрування RSA, цифрові підписи DSA й ECDSA), а також розгортання елементів інфраструктури відкритих ключів (PKI) через створення сертифікатів.

УДК 004.75:004.056.55

ЗМІСТ

ВСТУП

Сучасний етап розвитку інженерії програмного забезпечення характеризується стрімким переходом до розподілених, мікросервісних та децентралізованих архітектур. За умов, коли компоненти програмних комплексів взаємодіють через відкриті й незахищені мережеві середовища, безпека функціонування систем стає обов'язковою складовою загального життєвого циклу розробки. Надійність будь-якого високорівневого протоколу безпеки чи інтеграційного API фундаментально залежить від стійкості, швидкодії та коректності програмної реалізації базових криптографічних примітивів.

Для розробника програмного забезпечення криптографія – це насамперед об'єкт проєктування, який функціонує в умовах обмежених обчислювальних ресурсів, процесорного часу та пам'яті. Практична реалізація криптографічного захисту в сучасних системах базується на засадах безпечного програмування (Secure Coding) та використанні перевірених індустріальних стандартів. Спроби створення алгоритмів «з нуля» у реальних проєктах часто призводять до критичних архітектурних помилок. Натомість сучасний інженер має досконало володіти інструментарієм наявних екосистем розробки, розуміючи внутрішню логіку роботи примітивів на рівні окремих бітів та байтів.

Усі лабораторні роботи виконуються з використанням можливостей платформ JCA (Java Cryptography Architecture) та JCE (Java Cryptography Extension).

Методичні вказівки розроблено таким чином, щоб трансформувати теоретичні знання студентів у стійкі навички побудови захищеного програмного забезпечення, здатного протидіяти актуальним загрозам у розподіленому цифровому просторі.

Практикум містить завдання для виконання лабораторних робіт з дисципліни «Основи кібербезпеки та захисту інформації», яка викладається у рамках підготовки бакалаврів за спеціальністю F3 «Комп'ютерні науки».

ЛАБОРАТОРНА РОБОТА №1. Програмна реалізація та дослідження симетричних перетворень типу гамування, перестановка і підстановка. Формування ключових даних

1.1 Мета роботи

Ознайомитись з такими криптографічними перетвореннями як перестановка, підстановка, гамування та реалізувати відповідні процедури зашифрування/розшифрування для кожного типу перетворення із використанням мови програмування Java.

1.2 Теоретичні відомості

Класифікація та архітектурні принципи симетричних перетворень. Симетрична криптографія базується на використанні єдиного секретного ключа для шифрування та розшифрування даних. Відповідно до класичних криптографічних принципів, сформульованих Клодом Шенноном [1], надійність симетричного шифру забезпечується двома основними властивостями:

- перемішування (Confusion) – приховування статистичного зв’язку між відкритим текстом, шифротекстом і ключем. Зазвичай реалізується через операції підстановки;
- розсіювання (Diffusion) – розподіл статистичних особливостей відкритого тексту по всій довжині шифротексту. Зазвичай реалізується через операції перестановки.

Сучасні симетричні криптосистеми поділяються на два великі класи: блокові шифри (обробляють дані фіксованими блоками) та потокові шифри (обробляють дані побітово або побайтово у вигляді безперервного потоку).

Операція підстановки (Substitution / S-Box). Підстановка – це метод шифрування, за якого елементи відкритого тексту (біти, байти, символи) замінюються іншими елементами відповідно до визначеного правила або таблиці замін (S-Box – Substitution Box).

Математично підстановку в скінченному алфавіті *A* можна описати як бієктивне відображення (перестановку множини символів):

$$\pi: A \rightarrow A.$$

Класичний приклад: шифр Цезаря (моноалфавітна підстановка).

Кожен символ зміщується на фіксовану кількість позицій *k* в алфавіті потужністю *N*:

- шифрування: $c_i = (m_i + k) \bmod N$,
- розшифрування: $m_i = (c_i - k) \bmod N$.

У сучасних блокових шифрах використовуються нелінійні табличні підстановки (S-блоки), які оперують байтами (елементами кінцевих полів Галуа $GF(2^8)$). Нелінійність S-блоків є критично важливою для протидії диференціальному та лінійному криптоаналізу.

Операція перестановки (Permutation / P-Box). Перестановка – це метод криптографічного перетворення, за якого елементи відкритого тексту міняються місцями (перевпорядковуються), але самі символи чи біти залишаються незмінними.

Математично перестановка на множині з *n* елементів задається функцією:

$$\sigma: \{1, 2, ..., n\} \rightarrow \{1, 2, ..., n\},$$

Відкритий текст: $M = (m_1, m_2, ..., m_n)$,

Шифротекст: $C = (m_{\sigma(1)}, m_{\sigma(2)}, ..., m_{\sigma(n)})$.

У комп’ютерній реалізації перестановок на рівні байтів використовуються індекси масивів, а на рівні бітів – побітові операції зсуву (<<, >>) та логічного «І» (&) для виділення потрібних розрядів. Перестановки забезпечують ефект розсіювання: зміна одного біта на вході призводить до зміни багатьох бітів на виході після кількох раундів шифрування.

Гамування (Stream Ciphering / XOR-перетворення). Гамування – це процес накладання на відкритий текст криптографічної гами, яка є послідовністю випадкових або псевдовипадкових елементів (бітів/байтів).

Шифрування виконується шляхом побітового або побайтового додавання за модулем 2 (операція XOR) елементів відкритого тексту та гами.

- Шифрування: $c_i = m_i \oplus g_i$
- Розшифрування: $m_i = c_i \oplus g_i$.

(Оскільки $x \oplus y \oplus y = x$, процедури шифрування та розшифрування є абсолютно ідентичними).

Формування та генерація ключових даних. Оскільки зберігати та передавати випадкову гаму гігабайтного розміру практично неможливо, у розподілених системах використовують генератори псевдовипадкових послідовностей (ГПВП / PRNG).

ГПВП – це детермінований алгоритм, який приймає на вхід короткий секретний ключ фіксованої довжини (мале число, яке називається **Seed** або початкове значення), і генерує на його основі довгу псевдовипадкову послідовність (гаму), що за своїми статистичними властивостями наближається до випадкової.

Критерії якості ключових даних:

- період генератора – кількість тактів, після яких послідовність гами починає циклічно повторюватися. Період має бути максимально великим.
- статистична випадковість – згенерована гама повинна проходити тести на випадковість (наприклад, NIST STS або Dieharder), тобто мати рівномірний розподіл нулів та одиниць, відсутність автокореляції тощо;
- криптографічна стійкість (CSPRNG) – неможливість передбачити наступний біт гами за попередніми відомими бітами з ймовірністю, більшою за 0.5, навіть якщо структура алгоритму повністю відома зловмиснику.

Необхідні теоретичні відомості наведені на слайдах лекцій і в додаткових матеріалах.

1.3 Хід роботи

Завдання № 1. Реалізувати шифр перестановки мовою програмування Java двома способами згідно із описаними вимогами.

Спосіб 1

– обраний алфавіт – англійський;

– відкритий текст та ключ задано константно (у вигляді масиву символів);

– реалізовано окремі методи для зашифрування та розшифрування;

– виконано зашифрування відкритого тексту згідно з обраним ключем;

– виконано розшифрування шифртексту згідно з обраним ключем;

– в консоль виведено відкритий текст, шифртекст та розшифрований текст; перевірено, що текст після розшифрування співпадає з відкритим текстом.

Спосіб 2

– обраний алфавіт – англійський;

– відкритий текст довільної довжини, що підлягатиме зашифруванню, вводиться користувачем з клавіатури;

– ключ шифрування формується програмно псевдовипадковим чином за допомогою Collections.shuffle;

– реалізовано окремі методи для генерації ключа, зашифрування та розшифрування;

– виконано зашифрування відкритого тексту згідно зі згенерованим ключем;

– виконано розшифрування шифртексту згідно зі згенерованим ключем;

– в консоль виведено ключ, шифртекст та розшифрований текст; перевірено, що текст після розшифрування співпадає з відкритим текстом.

Завдання № 2. Реалізувати шифр підстановки мовою програмування Java двома способами згідно із описаними вимогами.

Спосіб 1

– обраний алфавіт – англійський;

– відкритий текст та ключ (таблицю заміни) задано константно;

– реалізовано окремі методи для зашифрування та розшифрування;

– виконано зашифрування відкритого тексту згідно з обраним ключем (таблицею заміни);

– виконано розшифрування шифртексту згідно з обраним ключем;

– в консоль виведено відкритий текст, шифртекст та розшифрований текст;

– перевірено, що текст після розшифрування співпадає з відкритим текстом.

Спосіб 2

– обраний алфавіт – англійський;

– відкритий текст довільної довжини, що підлягатиме зашифруванню, вводиться користувачем з клавіатури;

– ключ шифрування (таблиця заміни) формується програмно псевдовипадковим чином;

– реалізовано окремі методи для генерації ключа, зашифрування та розшифрування;

– виконано зашифрування відкритого тексту згідно зі згенерованим ключем (таблицею заміни);

– виконано розшифрування шифртексту згідно зі згенерованим ключем;

– в консоль виведено ключ (таблицю заміни/підстановки), шифртекст та розшифрований текст; перевірено, що текст після розшифрування співпадає з відкритим текстом.

Завдання № 3. Реалізувати шифр гамування мовою програмування Java двома способами згідно наступних вимог.

Спосіб 1

– генерується відкритий текст у вигляді двійкової послідовності фіксованої довжини;

– генерується ключ у вигляді двійкової послідовності фіксованої довжини (що співпадає з довжиною вхідного тексту);

– реалізовано окремі методи для генерації відкритого тексту і ключа, зашифрування, розшифрування в режимі гамування;

– виконано зашифрування відкритого тексту згідно зі згенерованим ключем;

– виконано розшифрування шифртексту згідно зі згенерованим ключем;

– в консоль виведено відкритий текст, ключ, шифртекст та розшифрований текст у форматі HEX; перевірено, що текст після розшифрування співпадає з відкритим текстом.

Спосіб 2*

Алгоритм дій схожий зі способом 1. Відмінності:

– відкритий текст задається у вигляді рядка символів;

– для того, щоб реалізувати процедуру зашифрування, текст необхідно перетворити у двійковий формат;

– після розшифрування отриманий текст у вигляді двійкової послідовності перетворити на рядок символів. Порівняти вхідний та отриманий тексти.

1.4 Контрольні питання

1. Дайте визначення симетричної криптосистеми. Які її основні переваги та недоліки порівняно з асиметричною криптографією?

2. Поясніть суть та архітектурну різницю між поняттями «перемішування» (confusion) та «розсіювання» (diffusion). За допомогою яких саме операцій вони реалізуються у сучасних шифрах?

3. Опишіть математичний та алгоритмічний механізм операції перестановки. Яким чином перестановка бітів впливає на поширення змін (ефект лавини) у структурі шифротексту?

4. Чому операція додавання за модулем 2 (XOR) є базовою для потокового шифрування? Доведіть тотожність процедур шифрування та розшифрування за умови використання ідентичної гами.

5. Чим відрізняються криптографічно стійкі генератори (CSPRNG) від звичайних статистичних генераторів псевдовипадкових чисел?

ЛАБОРАТОРНА РОБОТА №2. Програмна реалізація та дослідження продуктивності блокового шифру AES у різних режимах роботи із використанням методів бібліотеки Java

1.1 Мета роботи

Ознайомитись з можливостями Java Cryptography API в частині реалізації шифрування за допомогою блокового шифру AES у різних режимах роботи та дослідити швидкодію функцій зашифрування та розшифрування.

1.2 Теоретичні відомості

Архітектурні особливості блокового шифру AES. Advanced Encryption Standard (AES) [2] базується на симетричному алгоритмі Rijndael, який став світовим стандартом завдяки своїй високій криптостійкості та швидкодії. На відміну від застарілого стандарту DES, який мав обмеження щодо довжини ключа та розміру блока, AES оперує фіксованим розміром блока даних, що становить 128 біт (або 16 байт). Процес шифрування побудований на архітектурі підстановочно-перестановочної мережі (SPN) та включає серію математичних трансформацій над внутрішньою матрицею станів розміром 4x4 байти.

Кількість раундів шифрування безпосередньо залежить від обраної довжини ключа. Для ключа розміром 128 біт виконується 10 раундів, для 192 біт – 12 раундів, а для найстійкішого варіанта у 256 біт – 14 раундів. Кожен раунд, окрім фінального, складається з чотирьох послідовних етапів: субституції байтів (SubBytes), зсуву рядків (ShiftRows), перемішування стовпців (MixColumns) та додавання раундового ключа за операцією XOR (AddRoundKey). Збільшення довжини ключа пропорційно підвищує стійкість системи до атак методом грубої сили, але водночас створює додаткове обчислювальне навантаження, що є ключовим чинником при дослідженні продуктивності у межах цієї лабораторної роботи.

Режими роботи блокових шифрів та їхній вплив на продуктивність. Оскільки реальні масиви даних майже завжди перевищують стандартний 16-байтний блок, для їх обробки застосовують спеціальні режими роботи. Вони визначають, яким чином математичний апарат алгоритму AES масштабується на послідовність блоків, і безпосередньо впливають як на безпеку, так і на швидкість виконання операцій.

Найпростішим і водночас найменш безпечним є режим електронної кодової книги (ECB). У цьому режимі кожен блок відкритого тексту шифрується абсолютно незалежно від інших з використанням одного й того самого секретного ключа. Головна перевага ECB полягає у максимальній швидкості виконання та можливості повної паралелізації обчислень на багатоядерних процесорах. Проте його фундаментальний недолік – збереження шаблонів даних: однакові блоки вхідного тексту завжди генерують ідентичні блоки шифртексту, що дозволяє зловмиснику бачити структуру повідомлення без знання ключа.

Для усунення цієї вразливості застосовують режим зчеплення блоків шифртексту (CBC). Перед шифруванням кожного поточного блока до нього застосовується операція XOR з результатом шифрування попереднього блока. Для старту процесу та забезпечення унікальності першого блока використовується вектор ініціалізації (IV). Це повністю приховує структуру даних, але створює суттєвий бар'єр для продуктивності: процес зашифрування є суворо послідовним і його неможливо розпаралелити, оскільки кожна наступна ітерація чекає на завершення попередньої. При цьому процес розшифрування в режимі CBC позбавлений цього обмеження і може виконуватися паралельно, оскільки всі блоки шифртексту вже відомі заздалегідь. Крім того, режим CBC теоретично вразливий до атак на основі оракула доповнення (Padding Oracle Attacks), що вимагає додаткових заходів безпеки на рівні протоколу.

Альтернативний підхід пропонує режим лічильника (CTR), який фактично перетворює блоковий шифр на потоковий. Алгоритм AES шифрує не саме повідомлення, а унікальне значення спеціального лічильника, яке змінюється для кожного наступного блока. Отриманий результат побайтово поєднується з

відкритим текстом за допомогою операції XOR. Режим CTR демонструє виняткову продуктивність, оскільки операції зашифрування та розшифрування є абсолютно незалежними для кожного блока, що дозволяє задіяти всі доступні обчислювальні ядра процесора одночасно.

Сучасним індустріальним стандартом є режим Галуа/лічильника (GCM), який належить до класу автентифікованого шифрування (AEAD). Він поєднує в собі швидкість та паралелізм режиму CTR із вбудованою функцією контролю цілісності інформації. Під час обробки даних режим GCM не лише шифрує повідомлення, а й паралельно обчислює унікальний цифровий тег автентифікації за допомогою арифметики в кінцевих полях Галуа. Це дозволяє одержувачу миттєво виявити будь-які спроби модифікації або підміни шифртексту. Обчислення тегу створює невелике додаткове навантаження на процесор, проте завдяки архітектурі лічильника цей режим залишається одним із найшвидших серед безпечних криптографічних рішень.

Механізми доповнення блоків (Padding). Оскільки режими ECB та CBC вимагають, щоб вхідне повідомлення було чітко кратним розміру блока, виникає потреба у використанні алгоритмів доповнення даних. Стандарт PKCS5Padding вирішує цю проблему шляхом додавання до кінця повідомлення певної кількості байтів, де значення кожного доданого байта точно дорівнює їхній загальній кількості. Наприклад, якщо до повного блока бракує трьох байт, у кінець буде записано три байти зі значенням 0x03.

Навіть якщо довжина початкових даних є ідеально кратною 16 байтам, алгоритм все одно змушений додати один повний фіктивний блок із 16 байт зі значенням 0x10, щоб уникнути неоднозначності під час розшифрування. Натомість режими CTR та GCM повністю позбавлені цих накладних витрат, оскільки вони працюють як потокові шифри і дозволяють обробляти дані довільної довжини без жодних змін структури, що позитивно впливає на загальну швидкість обробки файлів.

Апаратне прискорення та особливості Java Virtual Machine. Важливим чинником, який необхідно враховувати під час дослідження продуктивності, є

наявність апаратної підтримки шифрування на рівні центрального процесора. Сучасні архітектури процесорів містять спеціальний набір інструкцій AES-NI, призначений для виконання раундів шифрування безпосередньо в апаратних модулях кристала.

Віртуальна машина Java (JVM) за замовчуванням намагається використовувати ці інструкції через механізм внутрішніх функцій (intrinsics). Це призводить до того, що швидкість роботи алгоритму AES на практиці може виявитися значно вищою, ніж очікується від чисто програмної реалізації, а різниця в продуктивності між різними режимами може частково нівелюватися потужністю самого процесора. Для точного аналізу швидкодії в Java слід враховувати ефекти JIT-компіляції (прогріву віртуальної машини) та роботу збирача сміття (Garbage Collector), які можуть спотворювати результати перших ітерацій вимірювань.

Конфігурування та параметризація екземпляру класу Cipher. Створення та налаштування об’єкта класу Cipher є першим і ключовим кроком для виконання будь-якої криптографічної операції в архітектурі JCA. Замість використання стандартних конструкторів через ключове слово new, розробники застосовують фабричний метод Cipher.getInstance(). Головним аргументом цього методу є так званий рядок трансформації (transformation string), який визначає точну конфігурацію криптографічного конвеєра. Цей рядок має суворо визначену структуру, де через косу риску вказуються три параметри: базовий алгоритм, режим роботи та схема доповнення.

Перший параметр рядка вказує на базовий симетричний шифр (AES) і визначає, що під капотом будуть використовуватися математичні трансформації над 128-бітними блоками даних. Другий параметр задає режим зв'язування блоків (наприклад, CBC). Цей вибір критично впливає на подальший код: якщо обрано режим CBC або GCM, віртуальна машина під час ініціалізації обов'язково вимагатиме передачі додаткового об’єкта з параметрами, наприклад IvParameterSpec для вектора ініціалізації або GCMParameterSpec для налаштування довжини автентифікаційного тегу. Третій параметр визначає

схему доповнення даних (PKCS5Padding). Його використання обов'язкове для блокових режимів типу ECB та CBC, тоді як для потокових режимів CTR або GCM слід вказувати значення NoPadding.

Оскільки всі параметри до методу getInstance() передаються у вигляді текстових рядків, компілятор позбавлений можливості перевірити їхню коректність на етапі збирання. Будь-яка помилка або спроба викликати непідтримувану комбінацію параметрів виявиться лише під час виконання через механізм перевіряємих винятків (checked exceptions), таких як NoSuchAlgorithmException, NoSuchPaddingException або NoSuchProviderException.

Особливості життєвого циклу об’єкта Cipher в Java Cryptography API. Практична реалізація криптографічних операцій у середовищі Java [3] підпорядкована чітким правилам архітектури JCA/JCE. Процес починається з генерації надійних ключів та векторів ініціалізації. Замість стандартного генератора псевдовипадкових чисел використовується клас SecureRandom, який є криптографічно стійким. Створення секретного ключа для симетричного шифрування зазвичай відбувається за допомогою класу KeyGenerator.

Безпосереднє управління процесом шифрування здійснюється через екземпляр класу Cipher, життєвий цикл якого складається з кількох чітких послідовних фаз. Для кращого розуміння цього процесу розглянемо типовий програмний код конфігурації та виконання шифрування:

```
// 1. Підготовка ключа та вектора ініціалізації
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(128, new SecureRandom());
SecretKey secretKey = keyGen.generateKey();

byte[] iv = new byte[16];
new SecureRandom().nextBytes(iv);
IvParameterSpec ivSpec = new IvParameterSpec(iv);
```

```
// 2. Створення екземпляру Cipher
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

// 3. Ініціалізація об'єкта
cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);

// 4. Безпосереднє шифрування (обробка даних)
byte[] plaintext = "Секретне повідомлення для лабораторної".getBytes();
byte[] ciphertext = cipher.doFinal(plaintext);
```

Перша активна фаза роботи самого об’єкта реалізується через ініціалізацію методом init, де вказується режим роботи (зашифрування Cipher.ENCRYPT_MODE чи розшифрування Cipher.DECRYPT_MODE), передається об'єкт ключа та параметри алгоритму (в даному випадку IvParameterSpec). Ця фаза є дуже ресурсомісткою, оскільки під час неї відбувається розгортання ключа – обчислення підключів для кожного окремого раунду AES. Якщо у лабораторній роботі ініціалізацію об'єкта помістити всередину циклу вимірювання швидкодії, результати будуть суттєво викривлені.

Наступна фаза відповідає за обробку масивів даних. У наведеному прикладі для короткого повідомлення одразу викликається doFinal. Однак для великих файлів раціонально використовувати циклічний виклик методу update. Цей метод дозволяє порційно завантажувати байти в буфер, мінімізуючи використання оперативної пам’яті.

Фінальний етап завжди реалізується через метод doFinal, який завершує обчислення, накладає або перевіряє паддінг, видає останні байти шифртексту та автоматично повертає об’єкт Cipher у початковий стан, роблячи його готовим для повторного використання з тим самим ключем. Обробка великих файлів виключно за один виклик doFinal без використання update є грубою

архітектурною помилкою, що може призвести до вичерпання оперативної пам’яті.

1.3 Хід роботи

Завдання 1. Реалізація зашифрування/розшифрування рядків/файлів за допомогою шифру AES в різних режимах роботи (CBC, OFB, CTR)

Необхідно створити консольне меню, де користувач може обрати бажаний режим роботи алгоритму AES (CBC/OFB/CTR) та тип даних, який він хоче зашифрувати: рядок або текстовий файл. Після цього користувач вводить бажаний рядок або ім’я файлу для зашифрування.

– у разі вводу рядка, в результаті роботи програми у консоль повинні бути виведені: згенерований ключ (hex), вектор ініціалізації (hex), шифртекст (hex) та результат розшифрування (String);

– у разі вводу імені файлу, в результаті роботи програми повинні бути створені два нові файли: зашифрований і розшифрований. У консоль повинні бути виведені: *згенерований ключ (hex), вектор ініціалізації (hex), результат розшифрування (вміст розшифрованого файлу, саме розшифрованого, а не початкового, запропонованого користувачем для шифрування)*.

Основні етапи реалізації шифрування рядків

– ознайомитись з документацією [4] та підключити необхідні класи, наприклад:

```
import javax.crypto.Cipher;
```

– реалізувати метод для генерації ключових даних за допомогою класу KeyGenerator:

```
KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
keyGenerator.init(n);
```

SecretKey key = keyGenerator.generateKey();

або SecretKeyFactory (розгортання ключа з пароля).

– створити вектор ініціалізації (клас SecureRandom):

```
public static IvParameterSpec generateIv() {
        byte[] iv = new byte[16];
        new SecureRandom().nextBytes(iv);
        return new IvParameterSpec(iv);
}
```

– реалізувати метод зашифрування рядка:

```
Cipher cipher = Cipher.getInstance(algorithm);
cipher.init(Cipher.ENCRYPT_MODE, key, iv);
```

використати doFinal() для зашифрування. . .

– реалізувати метод розшифрування рядка:

```
Cipher cipher = Cipher.getInstance(algorithm);
cipher.init(Cipher.DECRYPT_MODE, key, iv);
...
```

– здійснити тестування реалізованих методів із заданням необхідних параметрів:

```
String input = "Cybersecurity";
SecretKey key = AESUtil.generateKey(128);
IvParameterSpec ivParameterSpec = AESUtil.generateIv();
String algorithm = "AES/CBC/PKCS5Padding";
```

Основні етапи реалізації шифрування файлів

Етапи, що стосуються генерації ключа та вектора ініціалізації, а також ініціалізації самого алгоритму шифрування є ідентичними до попереднього завдання з шифруванням рядків.

Для роботи з файлами нам знадобляться класи, що працюють з вхідними та вихідними потоками (IO classes).

Варіант 1. Це можуть бути стандартні FileInputStream та FileOutputStream. Для реалізації зашифрування необхідно прочитати файл за допомогою FileInputStream і записати інформацію у масив байтів, над яким далі здійснити шифрування. Зашифрований результат записати у новий вихідний файл за допомогою FileOutputStream.

Зверніть увагу, що файл може бути достатньо великим, тому краще розбити його на частини (зчитати частинами) та здійснити шифрування над кожним масивом байтів окремо. Для цього окрім методу doFinal() вам знадобиться метод update(). Якщо вам потрібно зашифрувати або розшифрувати великий файл, розбитий на кілька блоків, метод update() викликається один раз для кожного блоку даних і завершується викликом методу doFinal() з останнім блоком даних.

Наприклад в разі зашифрування:

```
. . .
FileInputStream inputStream = new FileInputStream(inputFile);
FileOutputStream outputStream = new FileOutputStream(outputFile);
byte[] buffer = new byte[64];
int bytesRead;
while ((bytesRead = inputStream.read(buffer)) != -1) {
        byte[] output = cipher.update(buffer, 0, bytesRead);
        if (output != null) {
                outputStream.write(output);
        }
```

```
}
byte[] outputBytes = cipher.doFinal();
if (outputBytes != null) {
        outputStream.write(outputBytes);
}
inputStream.close();
outputStream.close();
```

Варіант 2. Можна також скористатися спеціальними класами CipherInputStream та CipherOutputStream, приклади дивіться у відповідному розділі документації [5].

Завдання 2. Вимірювання швидкодії. Реалізуйте окремо тестування швидкодії методів зашифрування/розшифрування файлів у різних режимах роботи шифру. Для вимірювання часу скористатись методом System.nanoTime (). Для отримання більш точного результату, вимірювання для одного й того самого файлу (бажано, достатньо великого розміру) необхідно провести певну кількість разів (наприклад, 100), а потім знайти середнє значення. Знаючи розмір файлу та час виконання, ви можете обчислити швидкість зашифрування/розшифрування у Мбіт/сек, наприклад. Обов'язково вкажіть у звіті, на якій платформі проводились тестування, процесор з якими характеристиками використовувався.

1.4 Контрольні запитання

1. У чому полягає фундаментальна різниця між блоковими (наприклад, CBC) та потоковими (наприклад, CTR, OFB) режимами роботи шифру?
2. Що таке вектор ініціалізації (IV) і яке його головне криптографічне призначення?
3. Яка принципова різниця між методами update() та doFinal() класу Cipher?

4. Чому під час дослідження швидкодії ініціалізацію об'єкта (cipher.init()) необхідно виносити за межі циклу, в якому вимірюється час безпосереднього шифрування файлу?

5. З якою метою вимірювання часу обробки одного файлу проводиться багаторазово (наприклад, 100 разів) із подальшим обчисленням середнього значення?

ЛАБОРАТОРНА РОБОТА № 3. Програмна реалізація та дослідження властивостей функцій гешування SHA-2 і SHA-3 із використанням методів бібліотеки Java

1.1 Мета роботи

Ознайомитись з можливостями Java Cryptography API в частині реалізації гешування за допомогою алгоритмів SHA-2 та SHA-3 та дослідити їх лавинні та швидкісні властивості

1.2 Теоретичні відомості

Концепція криптографічного гешування та його властивості. Криптографічна функція гешування – це математичний алгоритм, який перетворює вхідний масив даних довільної довжини у фіксований рядок байтів, що називається дайджестом повідомлення, геш-значенням або контрольною сумою. Головною особливістю таких функцій є їхня незворотність (односпрямованість), що означає неможливість математичного відновлення початкового тексту за його гешем [6].

Криптографічно стійка функція гешування повинна відповідати трьом фундаментальним вимогам: стійкості до пошуку першого прообразу (неможливості підбору вхідних даних для відомого гешу), стійкості до пошуку другого прообразу (неможливості підбору інших вхідних даних, які дадуть такий самий геш, як і заданий початковий текст) та стійкості до колізій. Колізією називається ситуація, коли два абсолютно різних вхідних набори даних дають ідентичний геш. Оскільки простір можливих вхідних повідомлень є нескінченним, а простір вихідних гешів фіксований, колізії теоретично існують для будь-якої функції, проте для сучасних алгоритмів їх знаходження є обчислювально неможливим завданням.

Архітектура сімейства геш-функцій SHA-2. Сімейство алгоритмів SHA-2 (Secure Hash Algorithm 2) [7] було розроблене Агентством національної

безпеки США (NSA) і опубліковане як стандарт у 2001 році. До цього сімейства належать такі популярні функції, як SHA-224, SHA-256, SHA-384 та SHA-512, де числове значення вказує на довжину фінального дайджесту в бітах. В основі SHA-2 лежить класична структура Меркла-Дамґорда, яка передбачає розбиття вхідного повідомлення на блоки фіксованого розміру (наприклад, 512 біт для SHA-256) та їх послідовну обробку односпрямованою функцією стиснення.

Математичний апарат SHA-2 базується на логічних побітових операціях, таких як AND, OR, XOR, побітових зсувах та циклічних зсувах (ROT), а також на модульному додаванні. Незважаючи на високу швидкість та широке індустріальне поширення, архітектура Меркла-Дамґорда має теоретичну вразливість до атак подовження повідомлення (length-extension attacks). Цей недолік змусив криптографічну спільноту ініціювати розробку нового стандарту, який мав би принципово інший внутрішній дизайн.

Нове покоління геш-функцій та архітектура SHA-3. Стандарт SHA-3 [8] був офіційно прийнятий NIST у 2015 році в результаті відкритого конкурсу, де перемогу здобув алгоритм Keccak. SHA-3 не замінює SHA-2, а існує як альтернативний стандарт із кардинально іншим внутрішнім устроєм. На відміну від структури Меркла-Дамґарда, SHA-3 базується на архітектурі криптографічної «губки» (sponge construction), яка складається з двох послідовних фаз: всмоктування та витискання.

Під час фази всмоктування блоки вхідного повідомлення послідовно додаються за операцією XOR до внутрішнього стану «губки», після чого стан перемішується за допомогою багатораундової псевдовипадкової перестановки. Під час фази витискання з отриманого внутрішнього стану зчитується необхідна кількість бітів, що формують фінальний геш. Така конструкція повністю захищає SHA-3 від атак подовження повідомлення. Крім того, завдяки іншій математичній логіці перетворень, SHA-3 демонструє відмінні від SHA-2 показники продуктивності на різних апаратних платформах, що є одним із предметів дослідження у цій лабораторній роботі.

Лавинний ефект як міра криптографічної стійкості. Лавинний ефект (avalanche effect) є однією з найважливіших характеристик будь-якого криптографічного алгоритму, зокрема функцій гешування. Його суть полягає в тому, що мінімальна зміна вхідних даних (наприклад, інверсія всього одного біта, заміна одного символу, додавання пробілу чи зміна регістру літери) повинна призводити до кардинальної, непередбачуваної та лавиноподібної зміни всього фінального геш-значення.

Для ідеальної геш-функції лавинний ефект означає, що при зміні одного біта на вході кожен біт вихідного дайджесту змінює своє значення на протилежне з ймовірністю рівно п’ятдесят відсотків. Візуально у шістнадцятковому форматі (Hex) геші двох майже однакових рядків мають виглядати як абсолютно різні випадкові послідовності, між якими немає жодного видимого взаємозв'язку. Відсутність або слабкість лавинного ефекту свідчить про наявність статистичних закономірностей у криптоалгоритмі, що дозволяє зловмисникам проводити ефективні атаки з криптоаналізу.

Особливості реалізації гешування в Java Cryptography Architecture. В архітектурі Java Cryptography API (JCA) управління процесом обчислення криптографічних дайджестів покладено на клас java.security.MessageDigest. Як і більшість криптографічних класів у Java, він використовує фабричний метод getInstance, що приймає текстову назву алгоритму, наприклад SHA-256 або SHA3-256. Метод обчислення гешу digest() працює виключно з масивами байтів, тому перед передачею текстових рядків їх необхідно обов'язково конвертувати у бінарний вигляд, явно вказуючи кодування (бажано StandardCharsets.UTF_8), щоб уникнути платформно-залежних розбіжностей у результатах.

Для обробки потокових даних та обчислення контрольних сум файлів архітектура Java пропонує спеціалізований клас-фільтр java.security.DigestInputStream, який є обгорткою над стандартними вхідними потоками InputStream. Його унікальна особливість полягає в тому, що він автоматично оновлює внутрішній стан зв'язаного об’єкта MessageDigest у фоновому режимі під час звичайного зчитування даних із файлу через метод

read(). Це дозволяє уникнути ручного керування буферами та викликів методу update(), суттєво спрощуючи архітектуру програми та знижуючи ризик виникнення помилок при роботі з великими об'єктами файлової системи.

Оскільки метод digest() повертає результат у вигляді сирого масиву байтів, для коректного відображення та порівняння отриманих значень у консолі розробники зобов'язані конвертувати його у текстовий шістнадцятковий формат (Hex-string). Спроба перетворити масив байтів гешу безпосередньо у рядок через стандартний конструктор new String() призведе до спотворення даних та появи нечитабельних символів, оскільки байтові значення криптографічного дайджесту не є символами жодного текстового кодування.

Аспекти дослідження продуктивності геш-функцій. При дослідженні швидкодії алгоритмів SHA-2 та SHA-3 студентам слід враховувати апаратні особливості виконання побітових операцій. Наприклад, 512-бітові версії цих алгоритмів (SHA-512 та SHA-3-512) оперують 64-бітними словами, тому вони демонструють вищу ефективність на рідних 64-бітних архітектурах процесорів порівняно з 32-бітовими версіями (такими як SHA-256), які працюють із 32-бітними словами.

Також на загальну пропускну здатність у середовищі Java впливає механізм JIT-компіляції віртуальної машини. Для отримання об’єктивних метрик швидкості (у Мбіт/с) необхідно проводити багаторазові вимірювання (наприклад, 100 ітерацій) на файлах великого розміру, щоб нівелювати час «прогріву» JVM, ефекти відкладеного збирання сміття (Garbage Collection) та затримки дискової підсистеми введення-виведення операційної системи.

Огляд ключових пакетів Java Cryptography API. Криптографічний інтерфейс Java розподілений між кількома основними пакетами, кожен з яких відповідає за певну технологічну область. Базовий пакет java.security [9] містить фундаментальні класи для керування ключами, генерації цифрових підписів та обчислення дайджестів повідомлень. Пакет javax.crypto [4] розширює ці можливості, додаючи класи для симетричного та асиметричного шифрування, генерації сесійних ключів та обчислення імітовставок (MAC).

Для підтримки інфраструктури відкритих ключів використовується пакет java.security.cert, який забезпечує створення, перевірку та керування сертифікатами відповідності (зокрема стандарту X.509). Специфікація внутрішнього представлення ключів покладена на пакети java.security.spec та javax.crypto.spec. Фінальну ланку архітектури складають пакети java.security.interfaces та javax.crypto.interfaces, які визначають низькорівневі інтерфейси для конкретних типів ключів (RSA, DSA, EC, AES), що дозволяє розробникам отримувати доступ до специфічних математичних параметрів криптосистем.

Стандартизація імен та концепція криптографічних провайдерів. Одним із головних інженерних досягнень JCA є концепція криптографічних провайдерів (Providers), реалізована через клас Provider. Замість жорсткого зв'язування коду з конкретною реалізацією алгоритму, Java використовує фабричні методи, які приймають назву алгоритму у вигляді текстового рядка. Для забезпечення сумісності між різними платформами та сторонніми бібліотеками впроваджено офіційний стандарт імен криптографічних алгоритмів Java [10].

Цей документ чітко регламентує, які саме назви мають використовувати розробники при виклику методів. Зокрема, у розділі MessageDigest алгоритми стандарту Java визначено такі офіційні ідентифікатори для гешування, як SHA-256, SHA-512, SHA3-256 або SHA3-512. Коли програма викликає метод створення екземпляра алгоритму, JCA послідовно опитує зареєстровані в системі провайдери відповідно до їхнього пріоритету, намагаючись знайти той, який підтримує вказану текстову назву. Це дозволяє прозоро оновлювати криптографічні модулі без необхідності змінювати логіку написаного програмного коду.

Робота з класом MessageDigest та потоками даних. Основним інструментом, який використовується під час виконання цієї лабораторної роботи для обчислення контрольних сум та дослідження лавинного ефекту, є клас java.security.MessageDigest. Повний опис його методів доступний у

документації класу MessageDigest [11]. Цей клас інкапсулює в собі функціонал криптографічних геш-функцій. Процес обчислення гешу починається зі створення об'єкта за допомогою виклику фабричного методу, після чого виконується ініціалізація та обробка інформації, як показано у наступному прикладі програмного коду:

```
// 1. Створення екземпляру MessageDigest для алгоритму SHA-256
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");

// 2. Перетворення вхідного рядка у масив байтів у кодуванні UTF-8
String originalString = "Cybersecurity";
byte[] inputBytes = originalString.getBytes(StandardCharsets.UTF_8);

// 3. Обчислення фінального дайджесту (геш-значення)
byte[] encodedhash = messageDigest.digest(inputBytes);
```

Важливою особливістю класу є те, що його методи оперують виключно бінарними даними. Метод digest(byte[] input) приймає масив байтів відкритого тексту, виконує над ним усі раунди математичних перетворень алгоритму та повертає фіксований за довжиною масив байтів. Якщо дані великого розміру або надходять частинами, розробник може багаторазово викликати метод update(byte[] input), поступово насичуючи внутрішній стан алгоритму, а потім виконати фінальний виклик digest() без аргументів, щоб отримати остаточний результат і автоматично скинути об'єкт MessageDigest до початкового стану для нового циклу обчислень.

Для обробки потокових даних та обчислення контрольних сум файлів архітектура Java пропонує спеціалізований клас-фільтр java.security.DigestInputStream. Його унікальна особливість полягає в тому, що він є обгорткою над вхідними потоками InputStream та автоматично оновлює внутрішній стан зв'язаного об'єкта MessageDigest у фоновому режимі під час

звичайного зчитування даних із файлу через метод read(). Це дозволяє уникнути ручного керування буферами та викликів методу update(), суттєво спрощуючи архітектуру програми та знижуючи ризик виникнення помилок при роботі з великими об'єктами файлової системи.

Оскільки метод digest() повертає результат у вигляді сирого масиву байтів, для коректного відображення та порівняння отриманих значень у консолі розробники зобов'язані конвертувати його у текстовий шістнадцятковий формат (Hex-string). Спроба перетворити масив байтів гешу безпосередньо у рядок через стандартний конструктор new String() призведе до спотворення даних та появи нечитабельних символів, оскільки байтові значення криптографічного дайджесту не є символами жодного текстового кодування.

1.3 Хід роботи

Завдання 1. Реалізація гешування рядків за допомогою геш-функцій SHA-256 та SHA3-256

Необхідно реалізувати методи для гешування рядків із використанням різних геш-функцій. Зверніть увагу, що метод гешування приймає на вхід масив байтів, на який потрібно перетворити рядок перед передачею його методу *digest*.

Наприклад:

```
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[] encodedhash = digest.digest(
originalString.getBytes(StandardCharsets.UTF_8));
```

Результат гешування виводити у форматі hex.

При тестуванні здійснити гешування одного й того самого рядка за допомогою різних алгоритмів, вивести результати.

Завдання 2. Реалізація гешування файлів (обчислення контрольної суми) за допомогою геш-функцій SHA-256 та SHA3-256.

Для реалізації цього завдання нам знадобиться додатковий клас *DigestInputStream*:

Наприклад:

```
try (InputStream is = new FileInputStream(filePath);
DigestInputStream dis = new DigestInputStream(is, md)) {
        while (dis.read() != -1) ;
        md = dis.getMessageDigest();
} catch (IOException e) {
        throw new IllegalArgumentException(e);}
```

Результат обчислення контрольної суми виводити у форматі hex.

При тестуванні здійснити обчислення контрольної суми одного й того самого файла за допомогою різних алгоритмів, вивести результати.

Завдання 3. Дослідження лавинного ефекту геш-функцій SHA-256 та SHA3-256

Для дослідження лавинного ефекту при гешуванні рядків необхідно виконати наступне:

– обрати декілька рядків, що мають дуже малу відмінність (наприклад, відрізняються одним символом) та здійснити гешування за допомогою одного й того ж алгоритму, наприклад:

Hey Jude, don't make it bad. Take a sad song and make it better.

Hey Jule, don't make it bad. Take a sad song and make it better.

Hey Jude don't make it bad. Take a sad song and make it better.

Hey Jude, don't make it bad. Take A sad song and make it better.

– порівняти результати (у hex) гешування обраних рядків за допомогою одного алгоритму, зробити висновки.

Для дослідження лавинного ефекту при обчисленні контрольної суми файлу необхідно виконати наступне:

– обрати файл, здійснити обчислення контрольної суми обраним алгоритмом, вивести результат;

– внести до файлу невеликі зміни (додати/прибрати один символ), здійснити обчислення контрольної суми обраним алгоритмом, вивести результат; повторити декілька разів;

– порівняти отримані результати (hex) і зробити висновки.

Завдання 4. Вимірювання швидкодії

Реалізуйте окремо тестування швидкодії методів гешування рядків або обчислення контрольної суми файлу за допомогою декількох різних алгоритмів (бажано, окрім 256-бітової функції гешування взяти також 512-бітову).

Для вимірювання часу скористатись методом System.nanoTime().

Для отримання більш точного результату, наприклад, вимірювання для одного й того самого файлу (бажано, достатньо великого розміру) необхідно провести певну кількість разів (наприклад, 100), а потім знайти середнє значення.

Знаючи розмір файлу та час виконання, ви можете обчислити швидкість зашифрування/розшифрування у Мбіт/сек, наприклад.

Обов’язково вкажіть у звіті, на якій платформі проводились тестування, процесор з якими характеристиками використовувався.

1.4 Контрольні запитання

1. Які три фундаментальні криптографічні вимоги висуваються до функцій гешування, і чим математично відрізняється поняття колізії від поняття незворотності функції?

2. Поясніть концептуальну різницю у внутрішній архітектурі сімейств алгоритмів SHA-2 (структура Меркла-Дамґарда) та SHA-3 (функція «губки»). Яку теоретичну вразливість SHA-2 було усунено в SHA-3?

3. Опишіть суть криптографічного лавинного ефекту. Які висновки про якість геш-функції можна зробити, якщо при зміні одного символу в тексті результуючий Hex-рядок змінився лише в кількох початкових позиціях?

4. Яким чином клас DigestInputStream автоматизує обчислення контрольної суми файлу під час його зчитування? Чому метод digest() повертає сирий масив байтів, і чому його не можна перетворювати на рядок за допомогою стандартного конструктора new String(byte[])?

5. Чому при вимірюванні швидкодії гешування 512-бітові функції (наприклад, SHA-512) на сучасних 64-бітних процесорах архітектурно можуть демонструвати вищу швидкість обробки даних, ніж їхні 256-бітові аналоги?

ЛАБОРАТОРНА РОБОТА № 4. Створення сертифікатів відкритих ключів

1.1 Мета роботи

Ознайомитись з можливостями створення сертифікатів відкритих ключів за допомогою OpenSSl та keytool; завантаження сертифікатів із використанням методів Java Cryptography API.

1.2 Теоретичні відомості

Концепція асиметричної криптографії та інфраструктури відкритих ключів. Асиметрична криптографія базується на використанні пари математично пов’язаних ключів: відкритого (public key) та таємного (private key). Таємний ключ зберігається в суворій таємниці його власником і використовується для розшифрування даних або створення цифрових підписів. Відкритий ключ може вільно розповсюджуватися загальнодоступними каналами зв'язку і застосовується іншими учасниками взаємодії для зашифрування інформації або перевірки автентичності цифрового підпису.

Однак у масштабних розподілених системах виникає критична проблема довіри, відома як атака «людина посередині» (Man-in-the-Middle), коли зловмисник може підмінити відкритий ключ адресата своїм власним. Для вирішення цієї проблеми та забезпечення автентичності розподілу відкритих ключів було створено Інфраструктуру відкритих ключів (Public Key Infrastructure – PKI). PKI – це комплексна система, що об'єднує апаратні та програмні засоби, політики безпеки та юридичні процедури, призначені для створення, керування, розподілу, використання, зберігання та відкликання цифрових сертифікатів.

Сертифікати відкритих ключів та стандарт X.509. Цифровий сертифікат відкритого ключа є електронним документом, який засвідчує зв’язок між відкритим ключем і особою, організацією або пристроєм, що цим ключем

володіє. Загальноприйнятим світовим стандартом для структури таких сертифікатів є специфікація X.509 [12], яка визначає суворий набір обов'язкових полів у форматі ASN.1.

Кожен сертифікат стандарту X.509 містить версію стандарту, унікальний серійний номер, ідентифікатор алгоритму цифрового підпису, назву видавця (Центру сертифікації), період дії сертифіката (дати «від» і «до»), дані про суб'єкта (власника сертифіката) та сам відкритий ключ суб'єкта разом із параметрами криптосистеми. Завершальним і найважливішим елементом сертифіката є цифровий підпис Центру сертифікації, який накладається на всі вищеперелічені поля. Наявність цього підпису гарантує цілісність документа та захищає його від будь-яких спроб несанкціонованої модифікації даних.

Ієрархія довіри та Центри сертифікації. Головним елементом архітектури PKI є Центр сертифікації (Certificate Authority – CA) – довірена третя сторона, яка перевіряє особу заявника та підтверджує автентичність інформації шляхом підписання сертифіката своїм таємним ключем. Модель довіри в інтернет-технологіях побудована за суворою ієрархічною структурою. На вершині цієї піраміди розташовані Кореневі центри сертифікації (Root CA), які видають самопідписані сертифікати (Root Certificates), де видавець і суб'єкт збігаються, а підпис накладено власним таємним ключем цього ж центру. Кореневі сертифікати зазвичай заздалегідь інтегруються у сховища безпеки операційних систем та веббраузерів.

Для підвищення безпеки Кореневі CA рідко використовуються для щоденного підписання сертифікатів кінцевих користувачів. Замість цього вони делегують свої повноваження Проміжним центрам сертифікації (Intermediate CA). Формується так званий ланцюжок довіри (Chain of Trust). Під час перевірки сертифіката вебсайту клієнтське програмне забезпечення послідовно перевіряє підписи: сертифікат сайту підписаний проміжним CA, сертифікат проміжного CA підписаний кореневим CA, а кореневому сертифікату операційна система довіряє безумовно. Якщо хоча б одна ланка цього ланцюжка пошкоджена або термін дії якогось сертифіката закінчився, вся система маркує підключення як

незахищене. Для локального тестування або внутрішніх корпоративних мереж розробники часто використовують самопідписані сертифікати, створені власноруч, що моделює поведінку Кореневого CA у мініатюрі.

Сховища ключів та сертифікатів: KeyStore в Java. Для безпечного зберігання криптографічних ключів, ланцюжків сертифікатів та довірених сертифікатів у середовищі Java використовується спеціалізована абстракція – сховище ключів, представлене класом java.security.KeyStore. Це захищений паролем контейнер, який на фізичному рівні зазвичай є зашифрованим файлом на диску. Вміст сховища організовано за допомогою текстових міток, які називаються аліасами (aliases). За допомогою аліасів програма може чітко звертатися до конкретного об'єкта всередині контейнера.

Існує два основних типи записів у KeyStore. Перший тип – це довірені сертифікати (Trusted Certificates), які містять лише відкритий ключ та інформацію про власника і використовуються для автентифікації інших сторін. Другий тип – це закриті ключі (Private Keys), які зберігаються разом із відповідним ланцюжком сертифікатів відкритого ключа та захищаються окремим додатковим паролем.

Протягом тривалого часу в Java стандартним форматом сховища був пропрієтарний формат JKS (Java Key Store). Проте в сучасних реалізаціях платформи за замовчуванням використовується відкритий та універсальний індустріальний стандарт PKCS#12 (файли з розширенням .p12 або .pfx), який підтримується більшістю криптографічних утиліт та операційних систем, що забезпечує високу кросплатформну сумісність.

Інструменти генерації та керування сертифікатами. Для практичного створення елементів PKI розробники використовують спеціалізовані консольні утиліти. Основним індустріальним інструментом є OpenSSL – потужний повнофункціональний криптографічний пакет із відкритим вихідним кодом. За допомогою командного інтерфейсу OpenSSL можна генерувати пари ключів RSA або EC, створювати запити на отримання сертифіката (Certificate Signing

Request – CSR), які потім передаються до CA, а також підписувати сертифікати, виступаючи в ролі власного Центру сертифікації.

З іншого боку, екосистема Java містить власну спеціалізовану утиліту командного рядка під назвою keytool, що постачається разом із Java Development Kit (JDK). Головне призначення keytool – це адміністрування файлів KeyStore. Вона дозволяє генерувати пари ключів безпосередньо всередині сховища Java, імпортувати наявні сертифікати, експортувати відкриті ключі у вигляді файлів сертифікатів та керувати списком довірених сертифікатів системи, забезпечуючи розробнику повний контроль над криптографічним оточенням програми.

Завантаження та обробка сертифікатів у Java Cryptography API. Програмна робота з цифровими сертифікатами в Java реалізується через пакети java.security та java.security.cert. Оскільки сертифікати на диску зазвичай зберігаються у бінарному форматі DER або текстовому форматі PEM (який є закодованим у Base64 варіантом DER із сигнальними рядками заголовка та підвалу), для їх обробки в Java використовується абстрактна фабрика java.security.cert.CertificateFactory.

Розробник ініціалізує фабрику для роботи зі стандартним типом сертифікатів, викликаючи CertificateFactory.getInstance("X.509"). Після цього обробка вхідного файлового потоку InputStream здійснюється за допомогою методу generateCertificate(), який автоматично розпізнає формат кодування, парсить структуру ASN.1 та повертає об'єкт типу java.security.cert.X509Certificate. Отриманий об'єкт надає розробнику повний програмний доступ до всіх метаданих сертифіката. Через відповідні методи класу можна перевірити термін дії документа за допомогою checkValidity(), отримати унікальний серійний номер, дізнатися унікальне ім'я суб'єкта чи видавця, а також вилучити сам об'єкт відкритого ключа (getPublicKey()) для його подальшого використання в операціях асиметричного шифрування або верифікації цифрових підписів у коді додатку.

1.3 Хід роботи

Завдання 1. Створення сертифіката відкритого ключа за допомогою OpenSSL

Ось перефразований варіант завдання, приведений до академічного стандарту лабораторних робіт. Текст структуровано, додано чіткі інструкції, технічний контекст, а також повністю збережено всі оригінальні посилання та команди з урахуванням правил форматування для лабораторних звітів.

Завдання 1. Розгортання локального Центру сертифікації та генерація самопідписаного сертифіката X.509 за допомогою OpenSSL

Мета етапу: Набути практичних навичок роботи з криптографічним пакетом OpenSSL для керування життєвим циклом сертифікатів: від генерації ключових пар до створення запитів і випуску фінального цифрового сертифіката.

Методичні вказівки та послідовність виконання:

1. **Підготовка робочого середовища.** Завантажте та встановіть криптографічний пакет OpenSSL з офіційного сайту [13]. Для користувачів операційної системи Windows, за бажанням або за потреби в UNIX-подібному CLI-оточенні, допускається використання емулятора командного рядка Cygwin.

2. **Генерація криптографічних матеріалів (Покроковий алгоритм):**

– створення закритого (таємного) ключа. Згенеруйте приватний ключ за допомогою алгоритму RSA з довжиною ключа 2048 біт. Замість {KeyFile} вкажіть власне ім'я файлу (наприклад, private.key):

```
openssl genpkey -out {KeyFile} -algorithm RSA -pkeyopt rsa_keygen_bits:2048
```

– створення запиту на підписання сертифіката (CSR). Ініціюйте процес генерації CSR-запиту, використовуючи створений на попередньому кроці

закритий ключ. Замість {CsrFile} вкажіть ім'я вихідного файлу запиту (наприклад, request.csr):

openssl req -new -key {KeyFile} -out {CsrFile}

Під час виконання заповніть поля метаданих для відмінного імені (DN) X.500. Зверніть особливу увагу на поле **Загальне ім'я (CN)**, де потрібно замінити {DeviceID} на ідентифікатор, ім'я або домен сервера вашого студента:

Country Name (2 letter code) [XX]:.
State or Province Name (full name) []:.
Locality Name (eg, city) [Default City]:.
Organization Name (eg, company) [Default Company Ltd]:.
Organizational Unit Name (eg, section) []:.
Common Name (eg, your name or your server hostname) []:{DeviceID}
Email Address []:.

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:.
An optional company name []:.

3. Верифікація та перевірка цілісності CSR. Обов'язково виконайте перевірку згенерованого запиту у текстовому вигляді без виведення закодованого тіла, щоб переконатися у відсутності помилок:

openssl req -text -in {CsrFile} -verify -noout

4. Випуск самопідписаного сертифіката. Підпишіть CSR-запит власним закритим ключем, виступаючи в ролі локального Центру сертифікації (CA).

Встановіть термін дії сертифіката у 365 днів. Замість {CrtFile} вкажіть ім'я підсумкового файлу сертифіката (наприклад, certificate.crt):

openssl x509 -req -days 365 -in {CsrFile} -signkey {KeyFile} -out {CrtFile}

5. Аналіз отриманих результатів:

Виведіть у консоль фінгерпринт (унікальний цифровий відбиток) створеного сертифіката для підтвердження успішності операції:

openssl x509 -in {CrtFile} -noout -fingerprint

Для глибшого розуміння структури сертифікатів концепції X.509 та ознайомлення з розширеними параметрами команд OpenSSL скористайтеся наступним посиланням [14].

Завдання 2. Створення сертифіката відкритого ключа за допомогою keytool

Перед початком генерації криптографічних матеріалів необхідно детально опрацювати базовий синтаксис утиліти та особливості її поведінки на різних платформах. Для цього слід ознайомитися з офіційними специфікаціями, серед яких ключовою є офіційна технічна документація Oracle з утиліти keytool [15].

Безпосередній алгоритм виконання завдання починається з ініціалізації сховища та генерації ключової пари. Для цього у терміналі викликається команда створення нового сховища ключів (або відкриття вже наявного файлу-контейнера) з одночасною генерацією всередині нього асиметричної пари ключів за алгоритмом RSA з довжиною 2048 біт та відповідного самопідписаного сертифіката. Під час конфігурування замість параметра {KeyStoreFile} підставляється реальна назва файлу (наприклад, keystore.p12), а замість параметра {AliasName} вказується унікальна текстова мітка, яка слугуватиме ідентифікатором запису в межах цього контейнера. Повна команда має вигляд

keytool -genkeypair -alias {AliasName} -keyalg RSA -keysize 2048 -keystore {KeyStoreFile} -validity 365. Під час її виконання утиліта в інтерактивному режимі запитає пароль для захисту всього сховища, а також запропонує ввести текстові метадані про власника сертифіката, такі як ім'я, організація, місто та дволітерний код країни. При цьому важливо переконатися, що створюване сховище відповідає сучасному стандарту PKCS#12, який є типовим для нових версій Java.

Наступним кроком є експорт сертифіката відкритого ключа з контейнера для його подальшого розповсюдження або використання в інших додатках. Для того, щоб вилучити відкритий ключ у вигляді окремого файлу сертифіката, застосовується команда keytool -exportcert -alias {AliasName} -file {CrtFile} -keystore {KeyStoreFile}, де параметр {CrtFile} замінюється на бажане ім'я вихідного документа, наприклад java_cert.crt. Після успішного експорту необхідно виконати верифікацію та повну перевірку вмісту контейнера, щоб переконатися, що всі записи прив'язані до правильних аліасів, а метадані записані без помилок. Перегляд детальної інформації про вміст сховища у розгорнутому текстовому форматі здійснюється за допомогою команди keytool -list -v -keystore {KeyStoreFile}.

Завдання 3. Робота з сертифікатами за допомогою Java Cryptography API

До базового набору інструментів Java Cryptography API, які забезпечують повний життєвий цикл обробки сертифікатів та побудови ланцюжків довіри, належать кілька ключових класів та інтерфейсів. Початкову ланку становить клас CertificateFactory, який діє як криптографічна фабрика для генерації об'єктів сертифікатів із вхідних бінарних або текстових потоків даних.

Для роботи зі складними структурами та перевірки автентичності застосовуються класи CertPathBuilder, який відповідає за автоматичну побудову валідного криптографічного шляху від кінцевого сертифіката до довіреного кореневого вузла, та CertPathValidator, призначений для суворої перевірки

побудованого ланцюжка на відповідність криптографічним стандартам та політикам безпеки. Додатково архітектура підтримує клас CertStore, який функціонує як загальнодоступне сховище або репозиторій для динамічного пошуку та вилучення сертифікатів і списків відкликаних сертифікатів (CRL) з метою оперативного контролю їхньої актуальності.

Замість базових форматів типу DER або PEM, обробку необхідно виконати для складнішого криптографічного стандарту PKCS#7 (який зазвичай має розширення .p7b або .p7c), оскільки цей формат здатний містити всередині не поодинокий сертифікат, а цілий конгломерат криптографічних даних, включаючи повні ланцюжки довіри проміжних центрів сертифікації. Для успішного виконання цієї процедури необхідно програмно відкрити вхідний файловий потік FileInputStream до створеного раніше сертифіката та ініціалізувати екземпляр фабрики за допомогою виклику CertificateFactory.getInstance("X.509").

Оскільки стандарт PKCS#7 може інкапсулювати колекцію об'єктів, для читання такого файлу замість методу generateCertificate() доцільно використати метод generateCertificates(InputStream inStream), який повертає параметризовану колекцію Collection<? extends Certificate>. Після успішного зчитування у програмі слід реалізувати ітерацію по отриманій колекції, привести кожен об'єкт до типу X509Certificate та вивести у консоль його ключові метадані, такі як серійний номер, алгоритм підпису, унікальні імена видавця (Issuer DN) та суб'єкта (Subject DN), а також межі періоду його чинності. Під час проєктування архітектури програми та написання коду необхідно спиратися на офіційні специфікації, наведені у документації Oracle з класу CertificateFactory для версії Java SE 23 [16].

1.4 Контрольні запитання

Що таке Інфраструктура відкритих ключів (PKI) та яку фундаментальну проблему асиметричної криптографії вона вирішує?

Чим самопідписаний сертифікат, створений під час лабораторної роботи, концептуально відрізняється від сертифіката, виданого комерційним Центром сертифікації?

Поясніть призначення та структуру запиту на підписання сертифіката (CSR).

У чому полягає різниця між утилітами OpenSSL та keytool з погляду керування криптографічними матеріалами?

Чому для читання сертифіката у форматі PKCS#7 в Java Cryptography API слід використовувати метод generateCertificates() замість generateCertificate() класу CertificateFactory?

ЛАБОРАТОРНА РОБОТА № 5. Програмна реалізація алгоритму направленого шифрування RSA із використанням методів бібліотеки Java

1.1 Мета роботи

Ознайомитись з можливостями Java Cryptography API в частині реалізації шифрування з відкритим ключем за допомогою алгоритму RSA; розробити власну реалізацію алгоритму RSA; порівняти результати виконання перетворень.

1.2 Теоретичні відомості

Математичні основи та архітектура алгоритму RSA. Алгоритм RSA, створений у 1977 році Рональдом Рівестом, Аді Шаміром і Леонардом Адлеманом, став першою повноцінною криптосистемою з відкритим ключем. Безпека алгоритму базується на обчислювальній складності математичної задачі факторизації великих цілих чисел, яка полягає у знаходженні простих множників для числа [17]. Процес генерації ключів починається з вибору двох довільних великих простих чисел p та q, які тримаються в суворій таємниці. На їх основі обчислюється модуль системи $n = pq$, довжина якого в бітах (наприклад, 2048 або 4096 біт) визначає поточний розмір ключа та криптостійкість алгоритму. Наступним кроком є обчислення значення функції Ейлера від отриманого модуля, яка для двох простих чисел набуває вигляду $\phi(n) = (p-1) \cdot (q-1)$.

Після цього обирається відкрита експонента e – ціле число, яке є взаємно простим із значенням $\phi(n)$ і лежить у діапазоні між одиницею та функцією Ейлера. Зазвичай у сучасній практиці як відкриту експоненту використовують фіксоване число 65537, оскільки воно містить мало одиниць у двійковому представленні, що суттєво прискорює процедуру піднесення до степеня під час шифрування. Далі за допомогою розширеного алгоритму Евкліда обчислюється таємна експонента d, яка є мультиплікативно оберненою до числа e за модулем

функції Ейлера, що математично виражається як $e \cdot d \equiv 1 \pmod{\phi(n)}$. Фінальний відкритий ключ складається з пари чисел (e, n), а таємний ключ містить пару (d, n), тоді як початкові числа p, q та значення $\phi(n)$ мають бути безповоротно знищені для запобігання компрометації системи.

Процедури зашифрування, розшифрування та концепція пасток. Математичні операції в алгоритмі RSA виконуються над числовими представленнями повідомлень у межах кінцевого кільця лишків за модулем n. Для зашифрування повідомлення, яке попередньо транслюється у велике ціле число M (де $M < n$), використовується відкритий ключ одержувача. Процес обчислення шифртексту C реалізується через операцію модульного піднесення до степеня $C = M^e \pmod{n}$. Процедура зворотного перетворення (розшифрування) виконується виключно власником таємного ключа за схожою формулою $M = C^d \pmod{n}$. Коректність цього відновлення гарантується малою теоремою Ферма та теоремою Ейлера, завдяки яким піднесення шифртексту до степеня d нівелює дію відкритої експоненти e.

З погляду теорії складності, RSA працює як односпрямована функція з потаємним входом або «пасткою» (trapdoor permutation). Обчислити значення функції в одному напрямку, тобто виконати шифрування, може будь-хто, володіючи загальнодоступними числами e та n. Проте виконати зворотне перетворення без знання таємної експоненти d обчислювально неможливо за прийнятний час, оскільки для її пошуку зловмиснику довелося б розкласти модуль n на початкові множники p та q, що для великих чисел є нереалізованим завданням на сучасному етапі розвитку обчислювальної техніки.

Реалізація асиметричних перетворень у Java Cryptography Architecture. В архітектурі Java Cryptography API (JCA) робота з алгоритмом RSA розділена на два рівні: високорівневий інтерфейс для швидкої інтеграції стандартних криптографічних конвеєрів та низькорівневий математичний апарат, який дозволяє розробити власну реалізацію алгоритму, що є завданням даної лабораторної роботи. Для генерації стандартних ключів у Java використовується клас `java.security.KeyPairGenerator`, який після ініціалізації

методом `initialize(2048)` створює об'єкт `KeyPair`, що містить у собі інтерфейси відкритого (`PublicKey`) та таємного (`PrivateKey`) ключів. Безпосередня обробка даних у JCA покладена на вже знайомий клас `javax.crypto.Cipher`, який ініціалізується рядком трансформації, наприклад `Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding")`.

Для виконання власної програмної реалізації алгоритму RSA стандартний клас `Cipher` не підходить через приховану внутрішню логіку. Замість цього розробники використовують клас `java.math.BigInteger`. Цей клас призначений для роботи з цілими числами довільної точності, розмір яких обмежений лише доступною оперативною пам'яттю системи. Клас `BigInteger` містить вбудовані високоефективні математичні методи, оптимізовані для криптографічних обчислень. Зокрема, для власної генерації простих чисел p та q використовується спеціальний конструктор `BigInteger(int bitlength, int certainty, Random rnd)`, який гарантує знаходження простого числа із заданою ймовірністю тесту Міллера-Рабіна. Обчислення найбільшого спільного дільника та пошук мультиплікативно оберненого числа для експоненти здійснюються методами `gcd()` та `modInverse()`. Фінальні етапи зашифрування та розшифрування у власному коді реалізуються через метод `modPow(BigInteger exponent, BigInteger m)`, який виконує піднесення до степеня за модулем, використовуючи швидкі алгоритми бінарного піднесення, що мінімізує обчислювальні витрати.

Порівняльний аналіз швидкодії симетричної та асиметричної криптографії. Невіддільною частиною дослідження властивостей RSA є аналіз його продуктивності порівняно з симетричними шифрами типу AES. Математичні операції RSA (піднесення великих 2048-бітних чисел до степеня) є обчислювально надзвичайно важкими для центрального процесора, через що швидкість роботи RSA у сотні або навіть тисячі разів нижча, ніж у блокових симетричних шифрів, побудованих на простих побітових замінах та перестановках.

З цієї причини в сучасних протоколах безпеки (таких як TLS/SSL, SSH або PGP) алгоритм RSA ніколи не застосовується для шифрування безпосередніх

великих масивів інформації або файлів. Замість цього розгортається гібридна криптосистема, де RSA використовується виключно на етапі рукостискання для безпечної передачі короткого випадкового симетричного ключа (сесійного ключа AES) або для перевірки автентичності цифрових підписів, а всі подальші потоки даних шифруються швидким алгоритмом AES.

1.3 Хід роботи

Завдання 1. Реалізація направленого шифрування за допомогою алгоритму RSA з бібліотеки Java:

– генерація ключової пари:

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
generator.initialize(2048);
KeyPair pair = generator.generateKeyPair();
```

– отримання приватного та відкритого ключів:

```
PrivateKey privateKey = pair.getPrivate();
PublicKey publicKey = pair.getPublic();
```

– запис ключа у файл:

```
try (FileOutputStream fos = new FileOutputStream("public.key")) {
        fos.write(publicKey.getEncoded()); }
```

– читання ключа за файлу:

```
File publicKeyFile = new File("public.key");
byte[] publicKeyBytes = Files.readAllBytes(publicKeyFile.toPath());
```

– відтворення екземпляру ключа:

```
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
EncodedKeySpec publicKeySpec = new
X509EncodedKeySpec(publicKeyBytes);
keyFactory.generatePublic(publicKeySpec);
```

– зашифрування рядків:

```
String secretMessage = "This is our lab work on RSA!";
Cipher encryptCipher = Cipher.getInstance("RSA");
encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey);
byte[] secretMessageBytes =
secretMessage.getBytes(StandardCharsets.UTF_8);)
byte[] encryptedMessageBytes = encryptCipher.doFinal(secretMessageBytes);
```

– розшифрування рядків:

```
Cipher decryptCipher = Cipher.getInstance("RSA");
decryptCipher.init(Cipher.DECRYPT_MODE, privateKey);
byte[] decryptedMessageBytes =
decryptCipher.doFinal(encryptedMessageBytes);
String decryptedMessage = new String(decryptedMessageBytes,
StandardCharsets.UTF_8);
```

– робота з файлами:

```
Path tempFile = Files.createTempFile("temp", "txt");
Files.writeString(tempFile, "some secret message");
byte[] fileBytes = Files.readAllBytes(tempFile);
```

```
Cipher encryptCipher = Cipher.getInstance("RSA");
encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey);
byte[] encryptedFileBytes = encryptCipher.doFinal(fileBytes);
try ( FileOutputStream stream = new FileOutputStream(tempFile.toFile())) {
        stream.write(encryptedFileBytes); }
```

Завдання 2. Створення власної реалізації алгоритму шифрування RSA мовою Java

Напишіть власну реалізацію алгоритму шифрування RSA мовою Java. Логіку реалізації розширеного алгоритму Евкліда можна взяти з [18].

Перевірити коректність реалізації.

Завдання 3. Вимірювання швидкодії

Реалізуйте окремо тестування швидкодії методу зашифрування алгоритмом RSA, який надає бібліотека, і розробленого вами методу. Порівняйте результати. Для вимірювання часу скористатись методом System.nanoTime().

Для отримання більш точного результату, наприклад, вимірювання для одного й того самого файлу (бажано, достатньо великого розміру) необхідно провести певну кількість разів (наприклад, 100), а потім знайти середнє значення.

Знаючи розмір файлу та час виконання, ви можете обчислити швидкість зашифрування/розшифрування у Мбіт/сек, наприклад. Обов’язково вкажіть у звіті, на якій платформі проводились тестування, процесор з якими характеристиками використовувався.

1.4 Контрольні запитання

1. На якій складній математичній задачі базується криптографічна стійкість алгоритму RSA, і які саме компоненти цієї системи формують відкритий та таємний ключі користувача?

2. Поясніть концепцію «пастки» (trapdoor permutation) в асиметричній криптографії. Завдяки якій математичній теоремі стає можливим відновлення початкового повідомлення M під час розшифрування за формулою $M = C^d \pmod{n}$?

3. З якої причини для власної програмної реалізації RSA в Java використовується клас BigInteger, а не стандартні типи даних? Які його вбудовані методи оптимізують виконання модульних математичних операцій?

ЛАБОРАТОРНА РОБОТА № 6. Програмна реалізація та дослідження властивостей алгоритмів цифрового підпису DSA та ECDSA із використанням методів бібліотеки Java

1.1 Мета роботи

Ознайомитись з можливостями Java Cryptography API в частині реалізації алгоритмів цифрового підпису DSA та ECDSA, дослідити властивості схем підпису.

1.2 Теоретичні відомості

Концепція та призначення електронного цифрового підпису. Електронний цифровий підпис (ЕЦП) [17] – це криптографічний інструмент, який забезпечує автентифікацію автора документа, контроль цілісності переданих даних та захист від відмови від авторства. На відміну від звичайного рукописного підпису, який залишається незмінним для будь-яких документів, ЕЦП є математичною функцією, яка жорстко пов’язує вміст конкретного повідомлення із таємним ключем підписувача. Будь-яка, навіть мінімальна модифікація тексту після його підписання робить ЕЦП невалідним, що миттєво фіксується стороною перевірки.

Процес створення підпису завжди базується на асиметричній криптографії, проте, на відміну від направленого шифрування, операції виконуються у зворотному порядку. Автор підписує повідомлення за допомогою свого таємного ключа (private key), а будь-який інший учасник взаємодії може перевірити цей підпис, використовуючи загальнодоступний відкритий ключ (public key) автора. При цьому для оптимізації обчислень підпис накладається не на саме повідомлення, яке може мати величезний розмір, а на його фіксований за довжиною криптографічний геш (дайджест).

Алгоритм цифрового підпису DSA та проблема дискретного логарифмування. Алгоритм DSA (Digital Signature Algorithm) [19] був

прийнятий Національним інститутом стандартів і технологій США (NIST) у 1991 році як федеральний стандарт цифрового підпису. Математична стійкість DSA базується на складності задачі обчислення дискретного логарифма у кінцевому полі. Параметрами системи є велике просте число p (модуль), просте число q (дільник числа $p-1$) та число g, яке породжує підгрупу порядку q за модулем p. Таємним ключем є випадкове число x, а відкритим ключем – значення $y = g^x \pmod p$.

Процес підписання повідомлення в DSA є рандомізованим: для кожного нового підпису обов'язково генерується тимчасове випадкове число k (nonce), яке повинно утримуватися в суворій таємниці і ніколи не повторюватися. Результатом підписання є пара чисел r and s, де r обчислюється на основі значення $g^k \pmod p \pmod q$, а s зв'язує геш-значення повідомлення, таємний ключ x та випадкове число k через модульне інвертування. Під час верифікації сторона перевірки обчислює допоміжні математичні компоненти на основі відкритого ключа y та гешу повідомлення, після чого зіставляє результат із числом r. Якщо числа збігаються, підпис вважається автентичним.

Еволюція алгоритму: Цифровий підпис на еліптичних кривих (ECDSA). Сучасним етапом розвитку технологій ЕЦП є алгоритм ECDSA (Elliptic Curve Digital Signature Algorithm) [19], який є аналогом DSA, перенесеним у математичний простір груп точок еліптичних кривих. Замість обчислень у полі великих цілих чисел, ECDSA оперує точками на кривій, заданій рівнянням Веєрштрасса над скінченним полем. Криптостійкість цієї схеми базується на задачі дискретного логарифмування в групі точок еліптичної кривої (ECDLP), яка є обчислювально набагато складнішою, ніж класичний дискретний логарифм.

Завдяки вищій математичній складності, ECDSA дозволяє досягти аналогічного рівня безпеки при значно меншій довжині ключів. Наприклад, рівень стійкості, який у класичному DSA забезпечується ключем розміром 3072 біти, в алгоритмі ECDSA досягається за допомогою ключа довжиною всього 256 бітів. Менший розмір ключів безпосередньо транслюється у значне скорочення

довжини самого цифрового підпису, зменшення витрат оперативної пам'яті та підвищення швидкості виконання математичних операцій на процесорі. Саме тому ECDSA є де-факто стандартом у сучасних протоколах безпеки, мобільних додатках, вебсертифікатах нового покоління та блокчейн-технологіях.

Особливості реалізації ЕЦП в Java Cryptography Architecture. В архітектурі Java Cryptography API (JCA) для роботи з цифровими підписами розроблено спеціалізований високорівневий клас java.security.Signature. На відміну від класу Cipher, який призначений для симетричного та асиметричного шифрування, клас Signature оптимізований під специфічний життєвий цикл ЕЦП, що включає фази генерації підпису (signing) та його верифікації (verification). Об'єкт цього класу створюється за допомогою фабричного методу, де вказується комбінація геш-функції та алгоритму підпису, наприклад Signature.getInstance("SHA256withDSA") або Signature.getInstance("SHA256withECDSA").

Робота з класом Signature організована як чіткий автомат із трьома станами. Після створення об'єкт перебуває в інактивованому стані. Для підписання його необхідно перевести в стан генерації за допомогою методу initSign(PrivateKey privateKey), передавши туди таємний ключ підписувача, згенерований за допомогою KeyPairGenerator. Для перевірки підпису об'єкт ініціалізується методом initVerify(PublicKey publicKey), куди передається відкритий ключ. Після ініціалізації дані повідомлення порційно або повністю передаються в об'єкт через метод update(byte[] data), який внутрішньо обчислює геш. Фіналізація процесу відбувається за допомогою методу sign(), який повертає масив байтів підпису, або методу verify(byte[] signature), який повертає логічне значення true або false, сигналізуючи про успішність або невдачу верифікації.

Безпека випадкових чисел та критичні вразливості DSA/ECDSA. Однією з найважливіших практичних властивостей алгоритмів DSA та ECDSA, яку студенти досліджують у межах лабораторної роботи, є їхня абсолютна залежність від якості генератора випадкових чисел. Тимчасовий параметр k

(nonce), який використовується при створенні кожного підпису, повинен бути не просто випадковим, а й унікальним.

Якщо зловмисник зможе перехопити значення k або якщо генератор випадкових чисел через програмну помилку видасть одне й те саме значення k для двох різних підписів, виконаних на одному ключі, виникає критична криптографічна вразливість. Володіючи двома такими підписами, зловмисник за допомогою простих арифметичних операцій у кільці лишків може повністю вичислити секретний приватний ключ користувача x. Для запобігання цій вразливості сучасні реалізації часто використовують детерміновану генерацію параметра k за стандартом RFC 6979, де це число обчислюється через стійкий HMAC-алгоритм на основі самого повідомлення та приватного ключа, що гарантує унікальність k для кожного унікального тексту без ризику повторення.

1.3 Хід роботи

Завдання 1. Реалізація застосування цифрового підпису за допомогою алгоритму DSA з бібліотеки Java.

Створення ключової пари. Це можна здійснити двома способами.

– аналогічно до попередньої роботи, **згенерувати нову пару ключів**:

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("DSA");
generator.initialize(2048);
KeyPair pair = generator.generateKeyPair();
PrivateKey privateKey = pair.getPrivate();
PublicKey publicKey = pair.getPublic();
```

– експортувати ключі, використовуючи вже існуючу пару (сертифікат), створену за допомогою keytool.

Завантаження приватного ключа:

```
KeyStore keyStore = KeyStore.getInstance("JKS");
keyStore.load(new FileInputStream("sender_keystore.jks"), "changeit");
PrivateKey privateKey = (PrivateKey) keyStore.getKey("senderKeyPair", "changeit");
```

Завантаження відкритого ключа:

```
KeyStore keyStore = KeyStore.getInstance("JKS");
keyStore.load(new FileInputStream("receiver_keytore.jks"), "changeit");
Certificate certificate = keyStore.getCertificate("receiverKeyPair");
PublicKey publicKey = certificate.getPublicKey();
```

Створення підпису:

```
Signature signature = Signature.getInstance("SHA1withDSA");
signature.initSign(privateKey);

byte[] messageBytes = Files.readAllBytes(Paths.get("message.txt"));

signature.update(messageBytes);
byte[] digitalSignature = signature.sign();
```

Перевірка підпису:

```
Signature signature = Signature.getInstance("SHA1withDSA");
signature.initVerify(publicKey);

byte[] messageBytes = Files.readAllBytes(Paths.get("message.txt"));

signature.update(messageBytes);
```

```
boolean isCorrect = signature.verify(digitalSignature);
```

Вносячи незначну зміну у вихідне повідомлення, перевірити, що в такому разі підпис буде визначено, як недійсний.

Додатковий приклад може бути знайдений у [20].

Завдання 2

Реалізація застосування цифрового підпису за допомогою алгоритму ECDSA з бібліотеки Java

– генерація ключової пари:

```
ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256k1");
KeyPairGenerator g = KeyPairGenerator.getInstance("EC");
g.initialize(ecSpec, new SecureRandom());
KeyPair keypair = g.generateKeyPair();
PublicKey publicKey = keypair.getPublic();
PrivateKey privateKey = keypair.getPrivate();
```

Підписування і перевірку можна реалізувати аналогічно до попереднього прикладу або із використанням формату JSON для представлення/передачі даних підпису:

– підписування:

```
Signature ecdsaSign = Signature.getInstance("SHA256withECDSA");
ecdsaSign.initSign(privateKey);
ecdsaSign.update(plaintext.getBytes("UTF-8"));
byte[] signature = ecdsaSign.sign();
String pub = Base64.getEncoder().encodeToString(publicKey.getEncoded());
String sig = Base64.getEncoder().encodeToString(signature);
```

– формат підписаних даних:

```
{
   "publicKey":
"MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEMEV3EPREEDc0t4MPeuYgreL
MHMVfD7iYJ2Cnkd0ucwf3GYVySvYTttMVMNMEKF554NYmdrOlqwo2s8J2tKt
/oQ==",
   "message": "Hello",
   "signature":
"MEUCIQCsuI4OcBAyA163kiWji1lb7xAtC8S0znf62EpdA+U4zQIgBcLbXtcuxXH
cwQ9/DmiVfoiigKnefeYgpVXZzjIuYn8=",
   "algorithm": "SHA256withECDSA"
}
```

– перевірка:

```
Signature ecdsaVerify = Signature.getInstance(obj.getString("algorithm"));
KeyFactory kf = KeyFactory.getInstance("EC");
EncodedKeySpec publicKeySpec = new
X509EncodedKeySpec(Base64.getDecoder().decode(obj.getString("publicKey")));
KeyFactory keyFactory = KeyFactory.getInstance("EC");
PublicKey publicKey = keyFactory.generatePublic(publicKeySpec);
ecdsaVerify.initVerify(publicKey);
ecdsaVerify.update(obj.getString("message").getBytes("UTF-8"));
boolean result =
ecdsaVerify.verify(Base64.getDecoder().decode(obj.getString("signature")));
```

1.4 Контрольні запитання:

1. Які три фундаментальні технологічні задачі безпеки вирішує електронний цифровий підпис (ЕЦП)? Чим за своєю математичною логікою та використанням ключів процес підписання відрізняється від процесу направленого асиметричного ширвування?

2. На якій складній математичній задачі базується стійкість класичного алгоритму DSA? Чому для перевірки підпису DSA не потрібно проводити операцію розшифрування самого документа?

3. Завдяки чому алгоритм ECDSA забезпечує такий самий рівень криптографічної стійкості, як і класичний DSA, але при значно меншому розмірі ключів? Як це впливає на продуктивність та довжину підпису на практиці?

4. Чому для роботи з ЕЦП в Java Cryptography Architecture використовується спеціалізований клас Signature, а не клас Cipher? Опишіть три стани цього автомата та методи, що забезпечують перехід між ними.

ПЕРЕЛІК ДЖЕРЕЛ

1. Claude E. Shannon The Mathematical Theory of Communication (16th Edition). The University of Illinois Press. 1971. 144 p.
2. Specification for the Advanced Encryption Standard (AES) : Federal Information Processing Standards Publication 197 (FIPS PUB 197). National Institute of Standards and Technology (NIST), 2001. 51 p.
3. Evans, Benjamin J., Jason Clark, and David Flanagan. Java in a Nutshell. O'Reilly Media, Inc., 2023.
4. Package javax.crypto. URL: https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html (дата звернення: 05.06.2026).
5. The Cipher Stream Classes. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#CipherStream (дата звернення: 05.06.2026).
6. Горбенко І.Д., Горбенко Ю.І. Прикладна криптологія. Підручник. Харків, ХНУРЕ, Форт, 2013 р., 1 та 2 видання, 878 с.
7. Інформаційні технології. Методи захисту. Геш-функції. Частина 3. Спеціалізовані геш-функції (ISO/IEC 10118-3:2018, IDT) : ДСТУ ISO/IEC 10118-3:2023 [Електронний ресурс]. [Чинний від 2023-11-01]. URL: https://online.budstandart.com/ua/catalog/doc-page?id_doc=103598 (дата звернення: 05.06.2026).
8. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions [Electronic resource] : FIPS PUB 202 / National Institute of Standards and Technology. Gaithersburg : NIST, 2015. 37 p. URL: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf (дата звернення: 05.06.2026).
9. Package java.security. URL: https://docs.oracle.com/javase/8/docs/api/java/security/package-summary.html (дата звернення: 05.06.2026).
10. Standard Algorithm Names. URL: https://docs.oracle.com/en/java/javase/25/security/java-security-standard-algorithm-names.html (дата звернення: 05.06.2026).
11. Class MessageDigest. URL: https://docs.oracle.com/javase/9/docs/api/java/security/MessageDigest.html (дата звернення: 05.06.2026).
12. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile [Electronic resource] : RFC 5280 / D. Cooper, S. Santesson, S. Farrell et al. — [Published May 2008]. — URL: https://datatracker.ietf.org/doc/html/rfc5280 (дата звернення: 05.06.2026).
13. OpenSSL. URL: https://openssl-library.org/source/ (дата звернення: 05.06.2026).
14. X.509 certificates. Microsoft Build 2026. URL: https://learn.microsoft.com/uk-ua/azure/iot-hub/reference-x509-certificates (дата звернення: 05.06.2026).
15. keytool - Key and Certificate Management Tool. Oracle. URL: https://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html (дата звернення: 05.06.2026).

16. Class CertificateFactory. URL: https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/security/cert/CertificateFactory.html (дата звернення: 05.06.2026).
17. Boneh, D., Shoup, V. A graduate course in applied cryptography. Draft 0.5. 2020. 400 p.
18. Розширений алгоритм Евкліда. URL: https://algoua.com/algorithms/algebra/extended_euclid_algorithm (дата звернення: 05.06.2026).
19. Digital Signature Standard (DSS). FIPS PUB 186-4 / National Institute of Standards and Technology (NIST). Gaithersburg, 2013. 130 p. URL: https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf (дата звернення: 05.06.2026).
20. Sign the Data. The Java™ Tutorials. https://docs.oracle.com/javase/tutorial/security/apisign/step3.html (дата звернення: 05.06.2026).

Електронне навчальне видання комбінованого використання
Можна використовувати в локальному та мережному режимі

Родінко Марія Юріївна

ПРОГРАМУВАННЯ КРИПТОГРАФІЧНИХ ПРИМІТИВІВ

Методичні вказівки
до лабораторного практикуму

В авторській редакції

Підписано до розміщення 21.05.2025. Гарнітура Times New Roman.
Ум. друк. арк. 3,07. Обсяг 0,589 Мб. Зам. № 338/26.

Харківський національний університет імені В. Н. Каразіна,
61022, м. Харків, майдан Свободи, 4.
Свідоцтво суб'єкта видавничої справи ДК № 3367 від 13.01.2009
Видавництво ХНУ імені В. Н. Каразіна