

Міністерство освіти і науки України
Національний університет "Запорізька політехніка"

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт з дисципліни
"Захист інформації в комп'ютерних системах"
для студентів спеціальності
123 "Комп'ютерна інженерія"
всіх форм навчання

Методичні вказівки до виконання лабораторних робіт з дисципліни "Захист інформації в комп'ютерних системах" для студентів спеціальності 123 "Комп'ютерна інженерія" всіх форм навчання. / Укл. Скрупський С.Ю., Киричек Г.Г. – Запоріжжя: НУ "Запорізька політехніка", 2019. – 56 с.

Укладачі: С.Ю. Скрупський, к.т.н., доцент
Г.Г. Киричек, к.т.н., доцент

Рецензент: В.В. Шкарупило, к.т.н., доцент

Відповідальний за випуск: С.Ю. Скрупський, к.т.н., доцент

Затверджено:
на засіданні кафедри
"Комп'ютерні системи та
мережі"
Протокол № 1 від 21.08.2019

Рекомендовано до видання
НМК ФКНТ
Протокол № 2 від 02.09.2019

ЗМІСТ

1 ЛАБОРАТОРНА РОБОТА № 1 ПРОГРАМУВАННЯ ТСП-СОКЕТІВ В РОЗПОДІЛЕНИХ ЗАСТОСУНКАХ	4
2 ЛАБОРАТОРНА РОБОТА № 2. ВИКОРИСТАННЯ АЛГОРИТМУ ПІДСТАНОВКИ ДЛЯ ЗАХИСТУ ІНФОРМАЦІЇ	13
3 ЛАБОРАТОРНА РОБОТА № 3. АЛГОРИТМИ СИМЕТРИЧНОГО ШИФРУВАННЯ ДЛЯ ЗАХИСТУ ІНФОРМАЦІЇ В КОМП'ЮТЕРНИХ СИСТЕМАХ	21
4 ЛАБОРАТОРНА РОБОТА № 4. АЛГОРИТМИ АСИМЕТРИЧНОГО ШИФРУВАННЯ ДЛЯ ЗАХИСТУ ІНФОРМАЦІЇ В КОМП'ЮТЕРНИХ СИСТЕМАХ.....	34
5 ЛАБОРАТОРНА РОБОТА № 5. ВИКОРИСТАННЯ ФУНКЦІЙ ХЕШУВАННЯ. ЕЛЕКТРОННИЙ ЦИФРОВИЙ ПІДПИС	46
ЛІТЕРАТУРА.....	56

1 ЛАБОРАТОРНА РОБОТА № 1. ПРОГРАМУВАННЯ TCP-SOCKETІВ В РОЗПОДІЛЕНИХ ЗАСТОСУНКАХ

Мета роботи: знайомство з TCP-сокетами для обміну даними між клієнтом та сервером. Програмування типового розподіленого застосунку, в якому клієнт і сервер обмінюються нешифрованими даними.

Коротка теорія.

Розподілений застосунок складається з двох частин (програм, проектів) – клієнтської і серверної. Коли ці програми запускаються, створюються клієнтський і серверний процеси, що взаємодіють один з одним, обмінюючись повідомленнями через сокети. Відмінність між клієнтом та сервером полягає в тому, що клієнт ініціює підключення, а сервер його приймає або відхиляє. Після встановлення з'єднання немає різниці хто є клієнтом, а хто сервером. Будь-яка зі сторін може відправляти та приймати повідомлення. Зазвичай клієнт відправляє дані серверу, той їх обробляє (наприклад, за допомогою запиту до бази даних), а потім надсилає клієнту відповідь. Один сервер може обробляти запити значної кількості клієнтів, навіть одночасно (за допомогою тредів/потоків). Розглянемо звичайний чат. Всі клієнти підключаються до одного сервера, а не напряму один до одного. Сервер виконує функцію поєднувальної ланки при передачі даних. Будь-який клієнт може відправити повідомлення. Воно спочатку потрапляє на сервер, а потім той перенаправляє його клієнту, якому призначене це повідомлення (або всім клієнтам, якщо чат має такий функціонал).

Протокол TCP передбачає встановлення логічного з'єднання між хостами і забезпечує надійну передачу даних. Протокол UDP, навпаки, не встановлює логічного з'єднання, і передача пакетів здійснюється без гарантії їх доставки адресату.

В цій роботі ви створите простий мережевий застосунок, який підтримує протокол TCP. Програми написані мовою Java.

Процеси, що виконуються на різних комп'ютерах, взаємодіють один з одним за допомогою повідомлень, що посилаються через сокети. Як показано на рис. 1.1, сокет знаходиться між процесом і протоколом TCP. Розробник застосунку повністю контролює ту

частину сокета, яка відноситься до прикладного рівня, майже не маючи нагоди впливати на «транспортну частину» (за винятком завдання декількох параметрів протоколу TCP, таких як максимальний розмір буфера і сегменту).

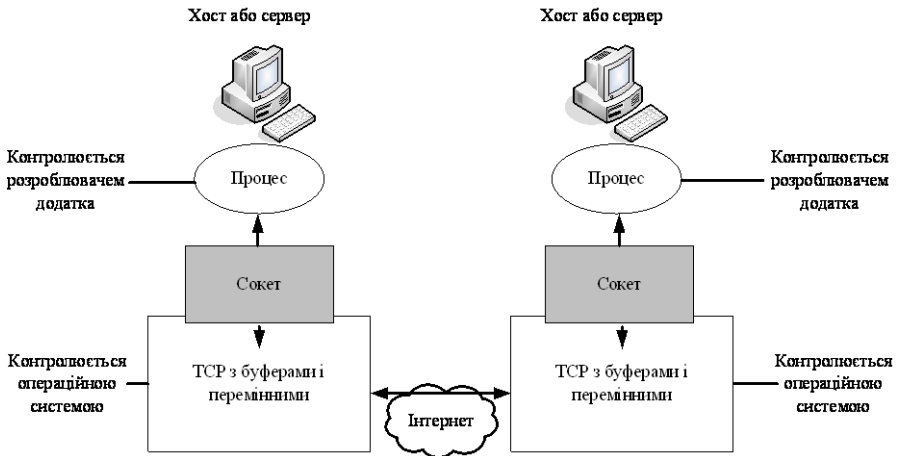


Рисунок 1.1 – Взаємодія процесів за допомогою TCP-сокетів

Тепер розглянемо процес взаємодії клієнтської і серверної програм більш детально. У функції клієнта входить ініціювання з'єднання з сервером, а сервер повинен бути готовим до встановлення з'єднання. Це означає, що, по-перше, програма-сервер повинна бути запущена раніше, ніж клієнт зробить спробу встановити з'єднання, і, по-друге, сервер повинен мати в своєму розпорядженні сокет, за допомогою якого встановлюється з'єднання. Першою дією клієнтської програми є створення сокета, при цьому програма вказує адресу серверного процесу, що складається з IP-адреси і номера порту процесу. Після створення сокета клієнтська сторона протоколу TCP здійснює процедуру потрібного рукоштовування з сервером, що закінчується встановленням з'єднання. Після прийому запиту на серверному сокеті на встановлення з'єднання сервер створює новий сокет для взаємодії з клієнтом. Після закінчення процедури рукоштовування встановлюється TCP-з'єднання між сокетом клієнта і новим сокетом серверу, який називають сокетом з'єднання (рис. 1.2).

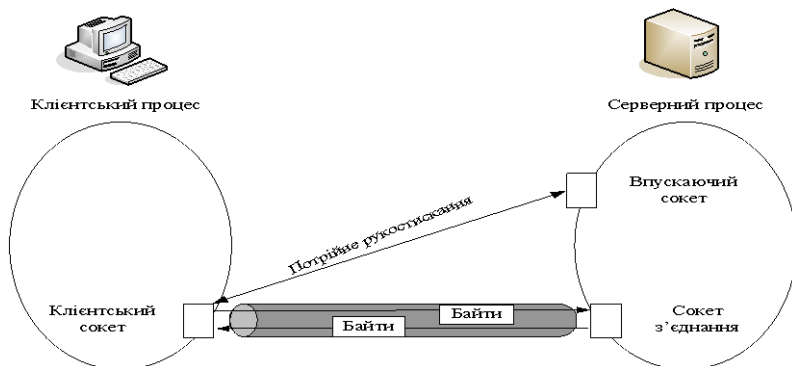


Рисунок 1.2 – Клієнтський сокет, серверний сокет і сокет з'єднання

З погляду застосунку TCP-з'єднання є прямим віртуальним каналом між сокетами з'єднання клієнта і сервера. Клієнт може здійснювати передачу будь-яких байтів (потік даних) через свій сокет, при цьому протокол TCP гарантує, що сервер отримає ці байти через свій сокет без спотворень і в тому ж порядку, в якому вони були передані. Під потоком даних розуміється послідовність байтів, що передається між двома процесами.

Порядок виконання роботи.

Для виконання цієї та наступних лабораторних робіт потрібно встановити JDK 1.8 або новіше x64 (або x86 залежно від версії операційної системи), яку можна безкоштовно завантажити з офіційного сайту компанії ORACLE: <https://www.oracle.com/technetwork/java/javase/downloads/index.html>

Крім цього вам знадобиться середовище розробки, наприклад, IntelliJ IDEA або Eclipse або Netbeans або будь-яке інше. Рекомендується застосування IntelliJ IDEA, оскільки це середовище найчастіше використовується в сучасних ІТ-компаніях. Програмування та тестування застосунків здійснюється на одному комп'ютері, а для демонстрації застосунків викладачу потрібно два комп'ютери (один для запуску серверної частини, а інший – для клієнтської).

Створюємо новий клієнтський проект. При цьому головний клас називаємо *Client*. Він буде містити метод *main*, в якому відбувається створення клієнтського сокета, підключення до сервера, введення з консолі повідомлення, відправка повідомлення до сервера, прийом

повідомлення від сервера, звільнення ресурсів, пов'язаних з сокетом. В даному випадку клієнт підключається до вузла *localhost* (це доменне ім'я використовується для відладки програмного забезпечення на одному комп'ютері, оскільки *localhost* – це завжди своє ім'я), на якому сервер прослуховує порт 5823. Код клієнтського застосунку:

```
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.util.Scanner;
public class Client
{
    public static void main(String[] args) throws
IOException
    { Socket clientSocket = new Socket("localhost",
        5823);
        Scanner inFromUser = new Scanner(System.in);
        Scanner inFromServer = new Scanner(
            clientSocket.getInputStream());
        DataOutputStream outToServer = new
        DataOutputStream(clientSocket.getOutputStream());
        System.out.print("Input message: ");
        String message = inFromUser.nextLine();
        outToServer.writeBytes(message + "\n");
        String response = inFromServer.nextLine();
        System.out.println("Server's response: " +
response);
        clientSocket.close();
    }
}
```

В конструкторі клієнтського сокета вказується доменне ім'я або ір-адреса сервера, та номер порта, на якому розміщений серверний застосунок. Звертаємо увагу на те, що при відправці повідомлення до нього слід додавати символ термінації строки "\n" для того, щоб приймаюча сторона могла легко відділити повідомлення. Крім того, в Java, як і в будь-яких інших мовах програмування існує багато класів, що реалізують функціонал потоків вводу/виводу, наприклад, *Scanner*, *PrintWriter*, *DataInputStream*, *DataOutputStream*, *ObjectInputStream*, *ObjectOutputStream* та інші. В лабораторній роботі у якості прикладу показано використання найбільш поширених класів *Scanner* для читання даних з потоку та *DataOutputStream* для запису даних в потік.

Далі створюємо серверний застосунок. Код серверного застосунку:

```
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
public class Server
{
    public static void main(String[] args) throws
    IOException
    {
        ServerSocket serverSocket = new ServerSocket(5823);
        while (true)
        {
            Socket socket = serverSocket.accept();
            Scanner inFromClient = new Scanner(
            socket.getInputStream());
            DataOutputStream outToClient = new
            DataOutputStream(socket.getOutputStream());
            String message = inFromClient.nextLine();
            String response = "Server has got " + message;
            outToClient.writeBytes(response + "\n");
        }
    }
}
```

Головний клас – *Server*, в методі *main* якого створюється серверний сокет, що розміщується на порті 5823. В умовно нескінченному циклі сервер прослуховує підключення клієнтів за допомогою блокуючого методу `accept()`. Блокуючим називають той метод, що блокує поточний тред (потік виконання процесу) до виконання якоїсь умови. В даному випадку – до підключення клієнта. Як тільки клієнт підключився, зберігаємо його сокет, який можна використовувати для отримання потоків вводу/виводу для роботи з цим клієнтом. Далі читаємо повідомлення від клієнта та відправляємо клієнту нове повідомлення.

Для запуску застосунку спочатку треба запустити серверну частину, а потім клієнтську. В середовищі IntelliJ IDEA можна натиснути праву кнопку миші по серверному класу *Server* та обрати пункт контекстного меню "Run Server.main()" – це запустить сервер.

Аналогічно запускаємо клієнта. Потім в клієнтському застосунку вводимо повідомлення з консолі та фіксуємо клавішею Enter. Повідомлення відправляється на сервер, він його отримує та відправляє клієнту інше повідомлення, яке буде відображено в консолі клієнта.

Недоліком нашого сервера є те, що він обробляє запити клієнтів послідовно, а не паралельно. Якщо до сервера підключаються багато клієнтів, то їм потрібно буде чекати поки черга обробки запитів не дійде до них. Для вирішення цього недоліку перепишемо сервер таким чином, щоб він приймаючи запит від клієнта, передавав його в інший тред (потік виконання коду), який виконується паралельно по відношенню до всіх тредів. Таким чином, сервер в умовно нескінченному циклі приймає підключення клієнта, отримує його сокет та передає його в інший тред. При цьому поточний тред переходить до наступної ітерації циклу (очікує нове підключення клієнта), а інші треди виконують обмін з клієнтом за допомогою сокету. Код багатопоточного сервера:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class MultiThreadedServer
{
    public static void main(String[] args) throws
IOException {
        ServerSocket serverSocket = new ServerSocket(5823);
        while (true)
        {
            Socket socket = serverSocket.accept();
            Thread t = new Thread(new ServerThread(socket));
            t.start();
        }
    }
}

import java.io.DataOutputStream;
import java.net.Socket;
import java.util.Scanner;
public class ServerThread implements Runnable
{
    private Socket socket;
```

```

ServerThread (Socket s)
{
    socket = s;
}
@Override
public void run()
{
    try
    {
        Scanner inFromClient = new Scanner(
            socket.getInputStream());
        DataOutputStream outToClient = new
            DataOutputStream(socket.getOutputStream());
        String message = inFromClient.nextLine();
        String response = "Server has got " + message;
        outToClient.writeBytes(response+"\n");
    }
    catch (Exception e)
    {
        System.out.print(e.getMessage());
    }
}
}

```

Запуск застосунку виконується аналогічно. Спочатку запускаємо "Run MultiThreadedServer.main()", а потім запускаємо клієнта та відправляємо повідомлення серверу. З боку клієнта нічого не змінюється, а з боку архітектури застосунку – багатопоточний сервер значно кращий за однопоточний.

Ми поліпшили наш застосунок з точки зору продуктивності, однак з боку захисту інформації наш застосунок далекий від ідеалу. Справа в тім, що клієнт та сервер обмінюються повідомленнями у відкритому вигляді, що вразливо до атаки "людина посередині". Використаємо простий мережевий сніфер для перехвату повідомлень, якими обмінюються клієнт та сервер. У якості такого сніфера візьмемо "SmartSniff" (Nir Sofer: <http://www.nirsoft.net>). Запускаємо сніфер, потім – сервер, а потім клієнта. Відправляємо дані від клієнта до сервера, отримуємо відповідь від сервера, потім переглядаємо результати захвату сніфера (рис. 1.3).

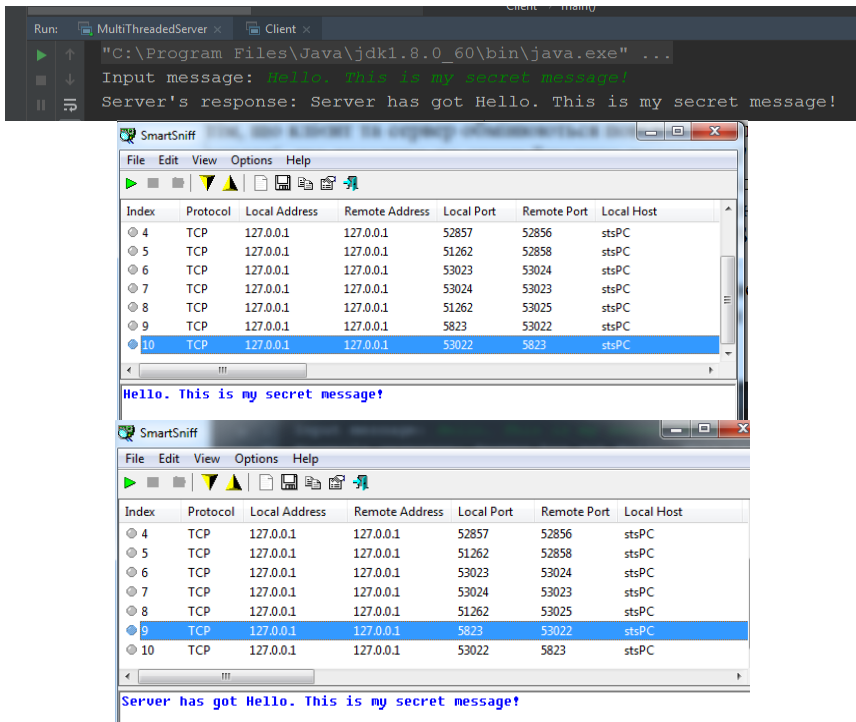


Рисунок 1.3 – Результати роботи сніфера

Таким чином, обмін нешифрованими повідомленнями між клієнтами та сервером у розподілених застосунках небезпечний для перехоплення та атак "людина посередині". Зловмисник може з легкістю прочитати та навіть змінити повідомлення.

Завдання для самостійного виконання:

Варіант 1: Створіть новий розподілений застосунок, в якому клієнт відправляє серверу текстовий документ (формат файлу txt). Сервер приймає документ та зберігає його під оригінальним ім'ям. Введення шляху до файлу, а також читання файлу клієнт виконує до підключення до серверу.

Примітка: для читання файлу з накопичувача зручно використовувати клас Scanner, у конструктор якого слід передати шлях до файлу:

```
Scanner sc = new Scanner(new FileInputStream(
    new File("./1.txt")));
```

Сам процес читання можна організувати по строках доки не дійдемо до кінця файлу:

```
StringBuilder stringBuilder = new StringBuilder("");
while (sc.hasNextLine())
{
    stringBuilder.append(scanner.nextLine());
}
String textFromFile = stringBuilder.toString();
```

Так само зручно виконувати передачу даних від клієнта до сервера. Читання сервер може виконувати по строках за допомогою сканера, а запис даних від клієнта до сервера – за допомогою класу `PrintWriter`:

```
PrintWriter printWriter = new
PrintWriter(socket.getOutputStream());
printWriter.println(textFromFile);
printWriter.flush();
```

Аналогічно сервер може записувати дані у свій файл. Для збереження файлу сервером під оригінальним ім'ям, спочатку треба від клієнта отримати строку з назвою файлу, а далі – передавати сам файл по строках.

Варіант 2: Створіть новий розподілений застосунок, в якому сервер відправляє клієнту вміст текстового файлу (файл обирається при старті серверу), а клієнт демонструє текст отриманого файлу у своїй консолі. Примітка з ідеєю застосунку така сама як для першого варіанту.

Зміст звіту:

- структура розподіленого застосунку;
- код програми згідно індивідуального варіанту;
- результати роботи сніфера під час перехоплення даних, якими обмінюються клієнт та сервер згідно індивідуального варіанту;
- відповіді на контрольні питання.

Контрольні питання:

- що собою являє сокет?
- як створити клієнтський та серверний TCP-сокет?
- у чому полягає вразливість розподілених застосунків, які обмінюються нешифрованими повідомленнями?
- перелічіть сучасні методи шифрування.
- які принципи покладено у роботу типового сніфера?

2 ЛАБОРАТОРНА РОБОТА № 2. ВИКОРИСТАННЯ АЛГОРИТМУ ПІДСТАНОВКИ ДЛЯ ЗАХИСТУ ІНФОРМАЦІЇ

Мета роботи: знайомство з принципами шифрування даних на прикладі найпростішого алгоритму підстановки. Програмування типового застосунку, в якому дані спочатку шифруються та записуються на накопичувач в шифрованому вигляді, а потім зчитуються та розшифровуються.

Коротка теорія.

Шифр підстановки є простим симетричним шифром (для шифрування та дешифрування використовується один ключ): літери початкового тексту замінюються літерами шифрованого тексту. Підстанівка Цезаря є найпростішим варіантом підстановки, що відноситься до групи моноалфавітних підстановок. В такому алгоритмі кожен символ початкового тексту замінюється на символ, зсунутий по алфавіту від початкового символу на певну сталу кількість позицій (на яку саме визначається ключем). Таким чином, початковий текст перетворюється в "безглуздий" набір символів. Для дешифрування необхідно кожен шифрований символ зсунути у протилежному напрямку на таку ж кількість позицій.

Алгоритм підстановки дає змогу познайомитися з найтривіальнішим принципом симетричної криптосистеми, однак він є легко зламним, якщо зломисник має шифрований і відповідний початковий текст, або шифрований текст обраного зломисником початкового тексту. Існує не так багато варіантів значень зсуву, всі вони можуть бути перевірені методом грубої сили. Один із способів зробити це – виписати уривок зашифрованого тексту в стовпець всіх можливих зсувів – техніка, іноді звана як "завершення простого компонента". Інший підхід до застосування методу грубої сили для злому – перевірити частоти зустрічальності букв. Зобразивши діаграмою частоти зустрічальності літер в шифрованому тексті, і знаючи очікуваний розподіл букв для звичайного тексту в алфавіті, можна легко визначити зсув, глянувши на зміщення деяких характерних рис на діаграмі. Цей метод відомий як частотний аналіз. Наприклад, в тексті англійською мовою частота літер E, T, (звичай найбільш частих), і Q, Z (звичай більш рідкісних) особливо відрізняються. Цей процес можна автоматизувати, зробивши, щоб

комп'ютерна програма оцінювала, наскільки добре фактичний розподіл частот відповідає очікуваному розподілу. Наприклад, можна використовуватися критерій χ^2 -квадрат. Для звичайного тексту на природній мові, швидше за все, буде лише один варіант декодування. Але, якщо використовувати дуже короткі повідомлення, то можливі випадки, коли можливі кілька варіантів розшифровки з різними зсувами. Наприклад зашифрований текст "MPQY" може бути розшифрований як "aden" так і як "know" (припускаючи, що початковий текст написаний англійською мовою). Багаторазове шифрування ніяк не покращує стійкість, так як застосування шифрів із зсувом 1 і 2 еквівалентно застосуванню шифру із зсувом $1 + 2$.

Існує багато модифікацій алгоритму підстановки, криптостійкість яких краща, за моноалфавітний шифр. Наприклад, однозвучний шифр підстановки схожий на моноалфавітний за винятком того, що символ початкового тексту може бути замінений одним з декількох можливих символів. Поліграмний шифр підстановки замінює не один символ, а цілу групу. Приклади: шифр Плейфера, шифр Хілла. Поліалфавітний шифр підстановки складається з декількох шифрів простої заміни, наприклад, шифр Віженера, шифр Бофора, одноразовий блокнот.

Порядок виконання роботи.

Створюємо новий проект під назвою "Simple cipher". Розділимо логіку нашого застосунку на три класи: клас *SubstitutionCipherProcessor* буде містити методи шифрування та дешифрування, клас *FileProcessor* – методи роботи з файлами, *Main* – основний клас, в якому створюються об'єкти інших класів та демонструється робота застосунку. Почнемо з класу *SubstitutionCipherProcessor*, код якого наведено нижче:

```
public class SubstitutionCipherProcessor
{
    private byte[] encryptedMessage;
    private String decryptedMessage;
    private int key;
    public SubstitutionCipherProcessor(int key)
    {
        this.key = key;
    }
}
```

```

public void encrypt(String message)
{
    encryptedMessage=message.getBytes();
    for(int i=0;i<encryptedMessage.length;i++)
    {
        encryptedMessage[i]+=key;
    }
}

public void decrypt(byte[] message)
{
    //cloning the encrypted message
    // in order to save the original message
    byte[] tmp = message.clone();
    for(int i=0;i<tmp.length;i++)
    {
        tmp[i]-=key;
    }
    decryptedMessage=new String(tmp);
}

public byte[] getEncryptedMessage()
{
    return encryptedMessage;
}

public String getDecryptedMessage()
{
    return decryptedMessage;
}

public int getKey()
{
    return key;
}
}

```

Клас виконує обробку процесів (processing) зберігання ключа, шифрування та дешифрування. В конструктор класу передається ключ шифрування у числовому вигляді – кількість позицій для зсуву по алфавіту. Методи-геттери повертають ключ, шифрований текст та розшифрований текст.

Поля класу:

Поле	Пояснення
<code>byte[] encryptedMessage</code>	байтова форма шифрованого повідомлення. Для підвищення криптостійкості застосунку потрібно завжди зберігати шифрований текст у байтовій формі, а не у формі строки <i>String</i>
<code>String decryptedMessage</code>	розшифроване повідомлення
<code>int key</code>	Ключ шифрування/розшифрування, по суті кількість позицій для зсуву в алфавіті.

Методи класу:

Метод	Пояснення
<code>void encrypt(String message)</code>	приймає початкову строку, шифрує її та зберігає у байтовому уявленні
<code>void decrypt(byte[] message)</code>	приймає байтове уявлення шифрованого тексту та розшифровує його

Клас *FileProcessor* містить всю логіку роботи застосунку з файлами:

```
import java.io.*;
```

```
public class FileProcessor
{
    private File f;
    private String path;

    public FileProcessor(String path)
    {
        setFile(path);
    }
}
```

```

public void setFile(String path)
{
    this.path = path;
    f = new File(path);
}
public void writeKey(int key) throws IOException
{
    f.getParentFile().mkdirs();
    FileOutputStream fos = new FileOutputStream(f);
    DataOutputStream dos = new
        DataOutputStream(fos);
    dos.writeInt(key);
    dos.flush();
    dos.close();
}
public int readKey() throws IOException
{
    FileInputStream fis = new FileInputStream(f);
    DataInputStream dis = new DataInputStream(fis);
    int key = dis.readInt();
    dis.close();
    return key;
}
public void writeMessage(byte[] message)
    throws IOException
{
    FileOutputStream fos = new FileOutputStream(f);
    fos.write(message);
    fos.flush();
    fos.close();
}
public byte[] readMessage() throws IOException
{
    FileInputStream fis = new FileInputStream(f);
    byte[] encryptedBytes = new
        byte[(int)f.length()];
    fis.read(encryptedBytes);
    fis.close();
    return encryptedBytes;
}
}

```

Конструктор класу приймає шлях до файлу, обробка якого буде виконуватися екземпляром даного класу. Поля класу:

Поле	Пояснення
File f	об'єкт, що містить зручні методи для роботи з файлом
String path	шлях до файлу

Методи класу:

Метод	Пояснення
void setFile(String path)	назначає об'єкту типу File шлях до файлу, обробка якого буде виконуватися даним класом
void writeKey(int key)	створює новий файл та записує туди ключ шифрування, який передається в метод у якості параметру
int readKey()	повертає прочитаний з файлу ключ
void writeMessage(byte[] message)	записує в файл байтове уявлення шифрованого повідомлення, що передається в метод у якості параметру
byte[] readMessage()	зчитує з файлу шифроване повідомлення та повертає його байтове уявлення

Робота застосунку виконується в класі *Main*, код якого наведено нижче. Пояснення до коду наведено в коментарях. Спочатку створюється повідомлення, а потім процесор шифрування. Далі повідомлення шифрується та виводиться на консоль в шифрованому вигляді. Шифроване повідомлення та ключ записуються у відповідні файли. Вважаємо все це першою частиною застосунку. У другій частині, що логічно відділена від першої, створюється файловий процесор, що зчитує ключ та шифроване повідомлення з файлів. Процесор шифрування розшифровує повідомлення за допомогою ключа. Розшифроване повідомлення виводиться на консоль.

Код класу:

```

import java.io.IOException;
public class Main
{
    public static void main(String[] args)
    throws IOException
    {
        //creating a string to encrypt
        String message = "!Simple_substitution_cipher!";
        //creating the cipherProcessor with a key=7
        SubstitutionCipherProcessor encryptionProcessor=
        new SubstitutionCipherProcessor(7);
        //encrypting
        encryptionProcessor.encrypt(message);
        //writing an encrypted message to console
        System.out.println("Encrypted message is\n"+
new String(encryptionProcessor.getEncryptedMessage()));
        //writing an encrypted message and a key to files
        FileProcessor fileProcessor = new
        FileProcessor("./message.txt");
        fileProcessor.writeMessage(
        encryptionProcessor.getEncryptedMessage());
        fileProcessor.setFile("./key.txt");
        fileProcessor.writeKey(
        encryptionProcessor.getKey());

        //creating new fileProcessor
        //for reading the cipher and a key
        //assuming this as a message receiver side
        FileProcessor readFileProcessor = new
        FileProcessor("./key.txt");
        int key = readFileProcessor.readKey();
        //creating the cipher for decryption
        SubstitutionCipherProcessor decryptionProcessor=
        new SubstitutionCipherProcessor(key);
        readFileProcessor.setFile("message.txt");
        byte[] encryptedMessage =
        readFileProcessor.readMessage();
        //decrypting the read message
        decryptionProcessor.decrypt(encryptedMessage);
        String decryptedMessage =
        decryptionProcessor.getDecryptedMessage();
    }
}

```

```

        System.out.println("Decrypted message is\n"+
        decryptedMessage);
    }
}

```

Завдання для самостійного виконання:

Варіант 1: Створіть розподілений застосунок, в якому клієнт вводить в консолі повідомлення та ключ шифрування алгоритму підстановки. Шифроване повідомлення відправляється серверу. Після отримання повідомлення на сервері в консолі вводиться ключ розшифрування. Розшифроване повідомлення виводиться на консоль.

Примітка: сумісно з алгоритмом шифрування використовуйте сокети, описані в попередній лабораторній роботі.

Варіант 2: Створіть застосунок, в якому в консолі вводиться PIN-код, що складається з чотирьох цифр та ключ шифрування (від одного до семи) алгоритму підстановки. PIN-код шифрується та записується в файл. Далі з файлу читається шифрований PIN-код та до нього застосовується метод грубої сили (уявіть, що ви не знаєте ключ шифрування, але знаєте алгоритм та межі ключа). Всі варіанти розшифрованого PIN-коду записуються в файл.

Примітка: прочитавши з файлу шифрований PIN-код перебирайте всі можливі ключі (в діапазоні від одного до семи) для розшифрування.

Зміст звіту:

- код програми згідно індивідуального варіанту;
- скріншоти, що демонструють роботу застосунку;
- для першого варіанту: результати роботи сніфера під час перехоплення даних, якими обмінюються клієнт та сервер;
- для другого варіанту: вміст файлу з усіма можливими розшифрованими PIN-кодами та час, що було витрачено на метод грубої сили та характеристики комп'ютера, на якому виконувався злам шифру. Для цього в Java зручно використовувати метод `System.currentTimeMillis()`;

– відповіді на контрольні питання.

Контрольні питання:

- в чому сутність симетричної криптосистеми?
- переваги та недоліки шифру підстановки;
- методи зламу шифрів в симетричних криптосистемах;
- переваги розділення функціоналу застосунку на класи.

3 ЛАБОРАТОРНА РОБОТА № 3.

АЛГОРИТМИ СИМЕТРИЧНОГО ШИФРУВАННЯ ДЛЯ ЗАХИСТУ ІНФОРМАЦІЇ В КОМП'ЮТЕРНИХ СИСТЕМАХ

Мета роботи: знайомство з сучасними алгоритмами блочного симетричного шифрування, що використовуються для захисту інформації в комп'ютерних системах. Програмування типового застосунку, в якому дані шифруються та розшифровуються блоками алгоритмом AES.

Коротка теорія.

До найвідоміших алгоритмів блочного симетричного шифрування традиційно відносять американські DES та AES, та український ДСТУ ГОСТ 28147:2009 (перевиданий радянський ГОСТ 28147-89). Ці алгоритми свого часу стали стандартами шифрування даних в США та країнах Радянського Союзу. В симетричних алгоритмах для шифрування та розшифрування використовується один і той самий ключ, на відміну від асиметричних алгоритмів.

Алгоритми DES та ДСТУ ГОСТ 28147:2009 являють собою схему мережі Фейстеля. Коротко пояснемо її сутність. Інформація розбивається на блоки однакової (фіксованої) довжини. Отримані блоки називаються вхідними. У разі, якщо довжина вхідного блоку менше, ніж розмір, який обраний алгоритм шифрування здатен зашифрувати одноразово (розмір блоку), то блок подовжується будь-яким способом. Далі обраний блок ділиться на два підблока однакового розміру. Лівий підблок змінюється функцією з використанням раундового ключа. Результат складається по модулю 2 з правим підблоком. Результат буде використаний у наступному раунді в ролі лівого підблока. Лівий підблок поточного раунду буде використаний у наступному раунді в ролі правого підблока. За яким-небудь математичним правилом обчислюється раундовий ключ – ключ, який буде використовуватися в наступному раунді. Перераховані операції виконуються $N-1$ разів, де N – кількість раундів в обраному алгоритмі шифрування. При цьому між переходами від одного раунду до іншого ключі змінюються. Розшифрування інформації відбувається так само, як і шифрування, з тим лише винятком, що ключі йдуть в зворотному порядку, тобто не від першого до N -го, а від N -го до першого. Алгоритми шифрування на основі мережі Фейстеля вносять певні модифікації в базовий

алгоритм, що наведено вище. Так, в алгоритмі DES кількість раундів дорівнює 16, довжина ключа – 56 біт, розмір блоку – 64 біти. Алгоритм ДСТУ ГОСТ 28147:2009 використовує 32 раунди, розмір ключа – 256 біт, розмір блоку – 64 біта.

Наприкінці 90х років 20 століття було доведено, що DES нестійкий до різноманітних методів криптоаналізу. Тоді за результатами відкритого конкурсу в якості нового стандарту симетричного шифрування в США було обрано алгоритм Rijndael, який отримав назву AES. Основу AES складають так звані лінійно-підстановочні перетворення. Алгоритм AES представляє блок даних у вигляді двовимірного байтового масиву розміром 4x4. Всі операції проводяться над окремими байтами масиву, а також над незалежними стовпцями і рядками. У кожному раунді алгоритму виконуються наступні перетворення:

- операція `subBytes`, яка представляє собою табличну заміну кожного байта масиву даних, що запобігає використанню сучасних методів криптоаналізу;

- операція `shiftRows`, яка виконує циклічний зсув вліво всіх рядків масиву даних, за винятком нульового. Зсув i -го рядка масиву (для $i = 1, 2, 3$) проводиться на i байт. Це виконується для маскування статистичних зв'язків;

- операція `mixColumns` виконує множення кожного стовпця масиву даних на фіксований поліном;

- операція `addRoundKey` виконує накладення на масив даних матеріалу ключа – безпосередньо виконується шифрування.

Перед першим раундом алгоритму виконується попереднє накладення матеріалу ключа за допомогою операції `addRoundKey`, яка виконує накладення на відкритий текст перших чотирьох слів розширеного ключа. Останній же раунд відрізняється від попередніх тим, що в ньому не виконується операція `mixColumns`.

В алгоритмі широко використовуються табличні обчислення, причому всі необхідні таблиці задаються константно, тобто не залежать ні від ключа, ні від даних. На відміну від шифрів, побудованих по мережі Фейстеля, в AES функції шифрування і розшифрування різні. AES використовує ключі 128 біт, 192 біт, 256 біт. Розмір блоків – 128 біт. Число раундів залежить від довжини ключа і може складати 10, 12 або 14.

Для шифрування та дешифрування інформації, її спочатку потрібно розділити на блоки. Якщо її розмір не ділиться без залишку на довжину блока, що використовується певним алгоритмом, потрібно використати доповнення до необхідної кількості байт (*Padding*). Крім того, існує декілька режимів блочного шифрування, наприклад, найпростіший ECB, режим ланцюгування CBC, та інші. В режимі ECB повідомлення розбивається на блоки і кожен блок шифрується окремо, тому однакові блоки відкритого тексту шифруються в однакові блоки шифротексту, як наслідок, шифротекст легше піддається криптоаналізу. В режимі CBC, кожен блок відкритого тексту проходить операцію "додавання по модулю 2" з попереднім шифроблоком перед шифруванням. Таким чином, кожен шифроблок, залежить від усіх блоків оброблених до нього. В цьому режимі для шифрування першого блоку потрібно використовувати ініціалізаційний вектор – масив байт, що використовується для доповнення першого блоку та для додавання по модулю 2.

Порядок виконання роботи.

Створюємо новий проект під назвою "AES local". Розділимо логіку нашого застосунку на чотири класи: клас *AESProcessor* буде містити методи шифрування та дешифрування за алгоритмом AES, клас *KeyProcessor* – методи роботи з ключем, клас *FileProcessor* – методи роботи з файлами, *Main* – основний клас, в якому створюються об'єкти інших класів та демонструється робота застосунку.

Код класу *Keyprocessor* наведено нижче:

```
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

public class KeyProcessor
{
    private SecretKey secretKey;
    private int keySize;
    private IvParameterSpec ivspec;//init vector

    public KeyProcessor(int keySize)
    {
        if(keySize==256 || keySize==192 || keySize==128)
```

```

    {
        this.keySize=keySize;
    }
    else
    {
        this.keySize=128;
    }
    //The block size required depends
    //on the AES encryption block size.
    //For the default block size of 128 bits,
    //we need an initialization vector of 16 bytes.
    byte[] iv = new byte[keySize/8];
    SecureRandom sr = new SecureRandom();
    sr.nextBytes(iv);
    ivspec = new IvParameterSpec(iv);
}

public void generateRandomKey()
throws NoSuchAlgorithmException
{
    KeyGenerator keyGen =
        KeyGenerator.getInstance("AES");
    keyGen.init(keySize);
    secretKey = keyGen.generateKey();
}

public SecretKey getSecretKey()
{
    return secretKey;
}

public int getKeySize()
{
    return keySize;
}

public void setSecretKey(SecretKey secretKey)
{
    this.secretKey = secretKey;
}

```

```

public IvParameterSpec getIvSpec()
{
    return ivspec;
}

public void setIvSpec(byte[] iv)
{
    this.ivspec = new IvParameterSpec(iv);
}
}

```

Конструктор класу приймає довжину ключа шифрування в бітах. Якщо задати невірну довжину ключа – приймається найменша довжина 128 біт. Крім того, в конструкторі генерується випадковий ініціалізаційний вектор, розмір якого залежить від розміру ключа. Поля класу:

Поле	Пояснення
SecretKey secretKey	згенерований секретний ключ
int keySize	розмір ключа в бітах
IvParameterSpec ivspec	ініціалізаційний вектор, який потрібен для коректного шифрування першого блоку в режимі CBC

Методи класу:

Метод	Пояснення
void generateRandomKey()	створює випадковий секретний ключ
SecretKey getSecretKey()	повертає секретний ключ
int getKeySize()	повертає розмір секретного ключа
void setSecretKey(SecretKey secretKey)	замінює секретний ключ на заданий
IvParameterSpec getIvSpec()	повертає ініціалізаційний вектор
void setIvSpec(byte[] iv)	замінює ініціалізаційний вектор

Безпосередньо шифрування за алгоритмом AES виконується в методах класу *AESProcessor*:

```
import javax.crypto.*;
import javax.crypto.spec.IvParameterSpec;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import
java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;

public class AESProcessor
{
    private Cipher cipher;
    private byte[] encryptedMessage;
    private String decryptedMessage;

    public void encrypt(String message,
        SecretKey key, IvParameterSpec ivSpec)
        throws NoSuchPaddingException, NoSuchAlgorithmException,
        InvalidAlgorithmParameterException,
        InvalidKeyException, UnsupportedEncodingException,
        BadPaddingException, IllegalBlockSizeException
    {
        cipher =
            Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);
        byte[] inputMessage = message.getBytes("UTF-8");
        encryptedMessage = cipher.doFinal(inputMessage);
    }

    public void decrypt(byte[] message,
        SecretKey key, IvParameterSpec ivSpec) throws
        NoSuchPaddingException, NoSuchAlgorithmException,
        InvalidAlgorithmParameterException,
        InvalidKeyException, BadPaddingException,
        IllegalBlockSizeException, IOException
    {
        cipher =
            Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);
```

```

        encryptedMessage = message.clone();
        decryptedMessage = new
        String(cipher.doFinal(encryptedMessage));
    }

    public byte[] getEncryptedMessage()
    {
        return encryptedMessage;
    }

    public String getDecryptedMessage()
    {
        return decryptedMessage;
    }
}

```

В цьому класі використовується конструктор за замовчанням.
Поля класу:

Поле	Пояснення
Cipher cipher	використовуваний шифр
byte[] encryptedMessage	масив байтів, що відображає шифроване повідомлення
String decryptedMessage	розшифроване повідомлення

Методи класу:

Метод	Пояснення
void encrypt(String message, SecretKey key, IvParameterSpec ivSpec)	шифрує задане повідомлення, заданим секретним ключем, за допомогою заданого ініціалізаційного вектора
void decrypt(byte[] message, SecretKey key, IvParameterSpec ivSpec)	розшифровує задане шифроване повідомлення, заданим секретним ключем, за допомогою заданого ініціалізаційного вектора
byte[] getEncryptedMessage()	повертає масив байтів – шифроване повідомлення
String getDecryptedMessage()	повертає розшифроване повідомлення

Клас `FileProcessor` виконує весь необхідний функціонал стосовно обробки файлів, а саме: запис ключа в файл, запис ініціалізаційного вектора в файл, читання ключа з файлу, читання ініціалізаційного вектора з файлу, читання та запис повідомлення. Код класу наведено нижче:

```
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileProcessor
{
    private File f;
    private String path;

    public FileProcessor(String path)
    {
        setFile(path);
    }

    public void setFile(String path)
    {
        this.path = path;
        f = new File(path);
    }

    public void writeKeyToFile(SecretKey key)
    throws IOException
    {
        f.getParentFile().mkdirs();
        FileOutputStream fos = new FileOutputStream(f);
        fos.write(key.getEncoded());
        fos.flush();
        fos.close();
    }
}
```

```

public void writeInitVectorToFile(
    IvParameterSpec iv) throws IOException
{
    f.getParentFile().mkdirs();
    FileOutputStream fos = new FileOutputStream(f);
    fos.write(iv.getIV());
    fos.flush();
    fos.close();
}

public SecretKey readKeyFromFile()
    throws IOException
{
    FileInputStream fis = new FileInputStream(f);
    byte[] bytesOfKey = new byte[(int) f.length()];
    fis.read(bytesOfKey);
    SecretKey key =
        new SecretKeySpec(bytesOfKey, "AES");
    fis.close();
    return key;
}

public byte[] readInitVectorFromFile()
    throws IOException
{
    FileInputStream fis = new FileInputStream(f);
    byte[] bytesOfIv = new byte[(int) f.length()];
    fis.read(bytesOfIv);
    fis.close();
    return bytesOfIv;
}

public void writeMessage(byte[] message)
    throws IOException
{
    FileOutputStream fos = new FileOutputStream(f);
    fos.write(message);
    fos.flush();
    fos.close();
}

```

```

public byte[] readMessage() throws IOException
{
    FileInputStream fis = new FileInputStream(f);
    byte[] encryptedBytes =
        new byte[(int)f.length()];
    fis.read(encryptedBytes);
    fis.close();
    return encryptedBytes;
}
}

```

Робота застосунку виконується в класі *Main*, код якого наведено нижче. Пояснення до коду наведено в коментарях. Спочатку генеруємо випадковий 128-бітний ключ шифрування та ініціалізаційний вектор для алгоритму AES в режимі CBC з заповненням до стандартної довжини блоку. Далі записуємо ключ та вектор у відповідні файли. Після цього шифруємо повідомлення та записуємо шифрований варіант в файл та на консоль. Вважаємо все це першою частиною застосунку. У другій частині, що логічно відділена від першої, читаємо ключ шифрування та ініціалізаційний вектор з файлів, читаємо шифроване повідомлення, виконуємо розшифрування та виводимо результат на консоль.

Код класу *Main*:

```

import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import java.io.IOException;
import
java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;

public class Main
{
    public static void main(String[] args)
        throws IOException, IllegalBlockSizeException,
        InvalidKeyException, NoSuchPaddingException,
        NoSuchAlgorithmException, BadPaddingException,
        InvalidKeySpecException,
        InvalidAlgorithmParameterException
    {

```

```

//key generation. Standard JDK supports only
//128bit (for longer key length you have to
//install JCE extensions)
KeyProcessor initialKeyProcessor =
new KeyProcessor(128);
initialKeyProcessor.generateRandomKey();

//writing key to file
FileProcessor fileProcessor =
new FileProcessor("./key.txt");
fileProcessor.writeKeyToFile(
initialKeyProcessor.getSecretKey());

//writing initvector to file
fileProcessor.setFile("./initvector.txt");
fileProcessor.writeInitVectorToFile(
initialKeyProcessor.getIvspec());

//encryption method
KeyProcessor encryptionKeyProcessor =
new KeyProcessor(128);
encryptionKeyProcessor.setIvSpec(
fileProcessor.readInitVectorFromFile());
fileProcessor.setFile("./key.txt");
encryptionKeyProcessor.setSecretKey(
fileProcessor.readKeyFromFile());
AESProcessor aesProcessor = new AESProcessor();
aesProcessor.encrypt("My Secret _message_",
encryptionKeyProcessor.getSecretKey(),
encryptionKeyProcessor.getIvspec());

//write encrypted message to file
FileProcessor messageFileProcessor =
new FileProcessor("./encrypted.txt");
messageFileProcessor.writeMessage(
aesProcessor.getEncryptedMessage());

//write encrypted message to console
System.out.println("Encrypted Message is\n" +
new String(aesProcessor.getEncryptedMessage()));

//decryption method.

```

```

// Reading key and initvector from files
//and creating new keyProcessor for decryption
KeyProcessor decryptionKeyProcessor =
new KeyProcessor(128);
FileProcessor decryptionFileProcessor =
new FileProcessor("./key.txt");
decryptionKeyProcessor.setSecretKey(
decryptionFileProcessor.readKeyFromFile());
decryptionFileProcessor.setFile("./initvector.txt");
decryptionKeyProcessor.setIvSpec(
decryptionFileProcessor.readInitVectorFromFile());
AESProcessor AESDecrypter = new AESProcessor();
AESDecrypter.decrypt(
messageFileProcessor.readMessage(),
decryptionKeyProcessor.getSecretKey(),
decryptionKeyProcessor.getIvSpec());

//write decrypted message to console
System.out.print("Decrypted Message is\n" +
AESDecrypter.getDecryptedMessage());
}
}

```

Завдання для самостійного виконання:

Варіант 1: Створіть застосунок – шифрувальник файлів, в якому три режими роботи:

- генерація ключа та ініціалізаційного вектора і їх збереження в файли;

- шифрування папки. Клієнт вводить в консолі шлях до папки. Застосунок шифрує всі файли в цій папці (замінюючи оригінальні файли);

- розшифрування папки. Клієнт вводить в консолі шлях до папки. Застосунок розшифровує всі файли в цій папці (замінюючи шифровані файли на розшифровані).

Примітка: На мові Java для отримання списку файлів в папці зручно скористатися методами класу *File*, наприклад,

```

File folder = new File("D:/test");
if (folder.isDirectory())
{
    File[] files = folder.listFiles();
    for (File file : files)
    {

```

```

if(file.isFile())
{
    //do something with file
}
}
}

```

Варіант 2: Створіть розподілений застосунок – захищений алгоритмом шифрування AES чат між клієнтом та сервером. Клієнт вводить повідомлення в консоль, воно шифрується та передається на сервер, який його розшифровує та показує на консолі. Далі сервер вводить повідомлення-відповідь в консоль, шифрує та відправляє клієнту, який його розшифровує та показує. Так до тих пір, поки один з учасників чату не відправить іншому повідомлення "end".

Примітка: спочатку згенеруйте ключ шифрування та ініціалізаційний вектор для сеансу шифрованого зв'язку між клієнтом та сервером. Далі за допомогою захищеного каналу зв'язку (наприклад, флешка) передайте ключ та вектор обоим учасникам чату. При старті застосунків учасників чату прочитайте з файлів ключ шифрування та ініціалізаційний вектор, які будуть використано для організації захищеного чату.

Зміст звіту:

- код програми згідно індивідуального варіанту;
- скріншоти, що демонструють роботу застосунку;
- вміст файлу з ключем шифрування;
- вміст файлу з ініціалізаційним вектором;
- відповіді на контрольні питання.

Контрольні питання:

- в чому сутність мережі Фейстеля?
- які переваги алгоритму ДСТУ ГОСТ 28147:2009 над DES?
- коротко опишіть основні етапи алгоритму AES;
- які переваги алгоритму AES над DES?
- коротко опишіть алгоритм 3DES;
- коротко опишіть основні режими блочного шифрування. Чому режим ECB небезпечно використовувати?

4 ЛАБОРАТОРНА РОБОТА № 4.

АЛГОРИТМИ АСИМЕТРИЧНОГО ШИФРУВАННЯ ДЛЯ ЗАХИСТУ ІНФОРМАЦІЇ В КОМП'ЮТЕРНИХ СИСТЕМАХ

Мета роботи: знайомство з сучасними алгоритмами асиметричного шифрування, що використовуються для захисту інформації в комп'ютерних системах. Програмування типового застосунку, в якому дані шифруються та розшифровуються алгоритмом RSA.

Коротка теорія.

Однією з основних проблем будь-якої криптосистеми є розподіл ключів. Для обміну шифрованими даними між клієнтами криптосистеми спочатку потрібно передати таким клієнтам ключ. Передача ключа вимагає використання іншої криптосистеми. Для вирішення цієї проблеми були запропоновані криптосистеми з відкритим ключем (асиметричні). Коротко опишемо їх сутність. Спочатку генеруються два зв'язаних за математичними правилами ключі: один відкритий, а інший закритий. Відкритий ключ може бути вільно передано по незахищеному каналу зв'язку. Він використовується для шифрування повідомлень. Тобто будь-який учасник відкритої криптосистеми може зашифрувати повідомлення та передати його іншому учаснику. Для дешифрування не можна використовувати той самий відкритий ключ. Закритий ключ зберігається в секреті та використовується для дешифрування повідомлень.

Сучасні криптосистеми з відкритим ключем спираються на один з наступних типів необоротних перетворень:

- розкладання великих чисел в прості множники;
- обчислення логарифму в кінцевому полі;
- обчислення коренів алгебраїчних рівнянь.

Асиметричні криптосистеми використовуються в якості повноцінних засобів захисту інформації, в якості механізмів розподілу ключів для симетричних криптосистем в якості засобів електронного цифрового підпису. До найвідоміших алгоритмів асиметричного шифрування традиційно відносять RSA та ElGamal. В лабораторній роботі детальніше розглянемо алгоритм RSA. Стійкість алгоритму RSA визначається складністю розкладання великих чисел на множники. Нижче коротко описаний принцип дії алгоритму RSA:

- генерується два простих числа p і q (100 цифр і більше), причому $n = pq$.
- в якості відкритого ключа вибирається ціле число e , взаємно просте з числом $(p-1)(q-1)$.
- закритий ключ d обчислюється з умови $ed \bmod (p-1)(q-1) = 1$.
- шифрування відкритого числа m (де $m < n$), здійснюється шляхом обчислення $m^e \bmod n$.
- розшифрування шифрованого числа c здійснюється шляхом обчислення $c^d \bmod m$.

На даний момент довжина ключа в 2048 біт вважається надійною для забезпечення криптостійкості алгоритму. Основним недоліком алгоритму RSA та інших алгоритмів з відкритим ключем є їх низька продуктивність в порівнянні з симетричними алгоритмами шифрування.

Порядок виконання роботи.

Створюємо новий проект під назвою "RSA local". Розділимо логіку нашого застосунку на чотири класи: клас *RSAProcessor* буде містити методи шифрування та дешифрування згідно до алгоритму RSA, клас *FileProcessor* – методи роботи з файлами, *KeyProcessor* – методи роботи з ключами, *Main* – основний клас, в якому створюються об'єкти інших класів та демонструється робота застосунку. Почнемо з класу *KeyProcessor*, код якого наведено нижче:

```
import java.security.*;
public class KeyProcessor
{
    private PublicKey publicKey;
    private PrivateKey privateKey;
    private int keySize;

    public KeyProcessor(int keySize)
    {
        if(keySize==2048) this.keySize=2048;
        else this.keySize=1024;
    }

    public PublicKey getPublicKey()
    {
        return publicKey;
    }
}
```

```

public PrivateKey getPrivateKey()
{
    return privateKey;
}

public void setPublicKey(PublicKey publicKey)
{
    this.publicKey = publicKey;
}

public void setPrivateKey(PrivateKey privateKey)
{
    this.privateKey = privateKey;
}

public void generateRandomKeys()
    throws NoSuchAlgorithmException
{
    KeyPairGenerator keyGen = null;
    keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(keySize);
    KeyPair pair = keyGen.generateKeyPair();
    privateKey = pair.getPrivate();
    publicKey = pair.getPublic();
}
}

```

Конструктор ініціалізує розмір ключа: 2048 біт або 1024 біти.

Поля класу:

Поле	Пояснення
PublicKey publicKey	відкритий ключ
PrivateKey privateKey	закритий ключ
int keySize	розмір ключа

Методи класу:

Метод	Пояснення
void generateRandomKeys()	генерує випадкову пару ключів (відкритий та закритий) згідно заданої довжини ключа
PublicKey getPublicKey()	повертає відкритий ключ
PrivateKey getPrivateKey()	повертає закритий ключ

<code>void setPrivateKey(PrivateKey privateKey)</code>	заміняє закритий ключ
<code>void setPublicKey(PublicKey publicKey)</code>	заміняє відкритий ключ

Код класу *RSAProcessor* наведено нижче:

```
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.util.Base64;

public class RSAProcessor
{
    private PublicKey publicKey;
    private PrivateKey privateKey;
    private byte[] encryptedMessage;
    private String decryptedMessage;

    public RSAProcessor()
    {
        publicKey=null;
        privateKey=null;
    }

    public RSAProcessor(PublicKey publicKey)
    {
        this.publicKey=publicKey;
        this.privateKey=null;
    }

    public RSAProcessor(PrivateKey privateKey)
    {
        this.privateKey=privateKey;
        this.publicKey=null;
    }
}
```

```

public void setPublicKey(PublicKey publicKey)
{
    this.publicKey = publicKey;
}

public void setPrivateKey(PrivateKey privateKey)
{
    this.privateKey = privateKey;
}

public PublicKey getPublicKey()
{
    return publicKey;
}

public PrivateKey getPrivateKey()
{
    return privateKey;
}

public byte[] getEncryptedMessage()
{
    return encryptedMessage;
}

public String getDecryptedMessage()
{
    return decryptedMessage;
}

public void encrypt(String data) throws
BadPaddingException, IllegalBlockSizeException,
InvalidKeyException, NoSuchPaddingException,
NoSuchAlgorithmException
{
    Cipher cipher =
    Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.ENCRYPT_MODE,
                getPublicKey());
    this.encryptedMessage=
    cipher.doFinal(data.getBytes());
}

```

```

public void decrypt(byte[] data) throws
NoSuchPaddingException, NoSuchAlgorithmException,
InvalidKeyException, BadPaddingException,
IllegalBlockSizeException
{
    Cipher cipher =
        Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.DECRYPT_MODE, privateKey);
    this.decryptedMessage=
        new String(cipher.doFinal(data));
}

public void decrypt(String data) throws
IllegalBlockSizeException, InvalidKeyException,
BadPaddingException, NoSuchAlgorithmException,
NoSuchPaddingException
{
    decrypt(Base64.getDecoder()
        .decode(data.getBytes()));
}
}

```

Конструктор класу виконує ініціалізацію ключів значенням посилання за замовчанням (null). Методи-гетери та сетери відповідно повертають та замінюють значення полів. Поля класу:

Поле	Пояснення
PublicKey publicKey	відкритий ключ
PrivateKey privateKey	закритий ключ
byte[] encryptedMessage	шифроване повідомлення
String decryptedMessage	розшифроване повідомлення

Основні методи класу:

Метод	Пояснення
void encrypt(String data)	шифрує повідомлення алгоритмом RSA відкритим ключем в режимі ECB з заповнювачем блоків до стандартного розміру

<code>void decrypt(byte[] data)</code>	приймає шифроване повідомлення та розшифровує його за допомогою закритого ключа та алгоритму RSA. Результат записується у поле <code>encryptedMessage</code>
<code>void decrypt(String data)</code>	приймає шифроване повідомлення у вигляді строки та розшифровує його

Клас *FileProcessor* містить методи запису ключів в файли, читання ключів з файлів, а також читання повідомлення з файлу, запис повідомлення в файл. Код класу:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.KeyFactory;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
```

```
public class FileProcessor
{
    private File f;
    private String path;

    public FileProcessor(String path)
    {
        setFile(path);
    }

    public void setFile(String path)
    {
        this.path = path;
        f = new File(path);
    }
}
```

```

public void writePrivateKeyToFile(
    PrivateKey privateKey) throws IOException
{
    f.getParentFile().mkdirs();
    FileOutputStream fos = new FileOutputStream(f);
    fos.write(privateKey.getEncoded());
    fos.flush();
    fos.close();
}

public void writePublicKeyToFile(
    PublicKey publicKey) throws IOException
{
    f.getParentFile().mkdirs();
    FileOutputStream fos = new FileOutputStream(f);
    fos.write(publicKey.getEncoded());
    fos.flush();
    fos.close();
}

public PublicKey readPublicKeyFromFile()
    throws IOException, NoSuchAlgorithmException,
    InvalidKeySpecException
{
    PublicKey publicKey = null;
    FileInputStream fis = new FileInputStream(f);
    byte[] bytesOfPublicKey =
        new byte[(int) f.length()];
    fis.read(bytesOfPublicKey);
    publicKey = KeyFactory.getInstance("RSA")
        .generatePublic(new
        X509EncodedKeySpec(bytesOfPublicKey));
    fis.close();
    return publicKey;
}

public PrivateKey readPrivateKeyFromFile()
    throws IOException, NoSuchAlgorithmException,
    InvalidKeySpecException
{
    PrivateKey privateKey = null;
    FileInputStream fis = new FileInputStream(f);

```

```

        byte[] bytesOfPrivateKey =
        new byte[(int) f.length()];
        fis.read(bytesOfPrivateKey);
        privateKey = KeyFactory.getInstance("RSA")
                        .generatePrivate(
        new PKCS8EncodedKeySpec(bytesOfPrivateKey));
        fis.close();
        return privateKey;
    }

    public void writeMessage(byte[] message) throws
    IOException
    {
        FileOutputStream fos = new FileOutputStream(f);
        fos.write(message);
        fos.flush();
        fos.close();
    }

    public byte[] readMessage() throws IOException
    {
        FileInputStream fis = new FileInputStream(f);
        byte[] encryptedBytes =
        new byte[(int)f.length()];
        fis.read(encryptedBytes);
        return encryptedBytes;
    }
}

```

Код основного класу *Main*, в якому демонструється робота застосунку:

```

import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;
public class Main
{
    public static void main(String[] args) throws
    IOException, IllegalBlockSizeException,
    InvalidKeyException, NoSuchPaddingException,

```

```

NoSuchAlgorithmException, BadPaddingException,
InvalidKeySpecException
{
    //keys generation
    KeyProcessor initialKeyProcessor =
    new KeyProcessor(2048);
    initialKeyProcessor.generateRandomKeys();

    //writing private key to file
    FileProcessor fileProcessor =
    new FileProcessor("./private.txt");
    fileProcessor.writePrivateKeyToFile(
    initialKeyProcessor.getPrivateKey());

    //writing public key to file
    fileProcessor.setFile("./public.txt");
    fileProcessor.writePublicKeyToFile(
    initialKeyProcessor.getPublicKey());

    //encryption method. Reading public key
    //from file and creating
    //new keyProcessor for encryption
    KeyProcessor encryptionKeyProcessor =
    new KeyProcessor(2048);
    encryptionKeyProcessor.setPublicKey(
    fileProcessor.readPublicKeyFromFile());
    RSAProcessor rsaEncryptor =
    new RSAProcessor(
    encryptionKeyProcessor.getPublicKey());
    rsaEncryptor.encrypt(
    "My super @confidential@ string!!!");

    //write encrypted message to file
    FileProcessor messageFileProcessor =
    new FileProcessor("./encrypted.txt");
    messageFileProcessor.writeMessage(
    rsaEncryptor.getEncryptedMessage());
    //write encrypted message to console
    System.out.println("Encrypted Message is\n" +
    new String(rsaEncryptor.getEncryptedMessage()));
}

```

```

//decryption method. Reading private key
//from file and creating new
//keyProcessor for decryption
KeyProcessor decryptionKeyProcessor =
new KeyProcessor(2048);
fileProcessor.setFile("./private.txt");
decryptionKeyProcessor.setPrivateKey(
fileProcessor.readPrivateKeyFromFile());
RSAProcessor rsaDecryptor = new RSAProcessor(
decryptionKeyProcessor.getPrivateKey());
rsaDecryptor.decrypt(
messageFileProcessor.readMessage());

//write decrypted message to console
System.out.print("Decrypted Message is\n" +
rsaDecryptor.getDecryptedMessage());
}
}

```

В першій логічній частині застосунку генеруємо ключі розміром 2048 біт та записуємо їх у відповідні файли. У другій частині – читаємо відкритий ключ з файлу та виконуємо шифрування повідомлення, яке в шифрованому вигляді записуємо у відповідний файл. В третій частині застосунку читаємо з файлу закритий ключ, за допомогою якого виконуємо розшифрування шифрованого повідомлення, та показуємо результат на консоль.

Завдання для самостійного виконання:

Варіант 1: Створіть захищений алгоритмом RSA розподілений застосунок для банку: сервер при старті з консолі отримує поточний курс гривні до євро. Клієнти надсилають суму для конвертації. Сервер виконує розрахунок та показує результат на своїй консолі.

Примітка: згенеруйте пару ключів. На сервер скопіюйте по захищеному каналу зв'язку (наприклад, флешка) закритий ключ, серед клієнтів розповсюдьте відкритий ключ. Коли клієнт хоче зконвертувати певну сумму, він спочатку шифрує її відкритим ключем, а потім надсилає серверу. Сервер розшифровує повідомлення своїм закритим ключем, виконує конвертацію та показує результат в своїй консолі.

Варіант 2: Створіть два застосунки: перший приймає відкритий ключ та шифрує алгоритмом *RSA* файли-зображення (bmp, png, jpeg).

Результат шифрування зберігається в окремому файлі під оригінальним ім'ям та з розширенням gsa. Другий – приймає закритий ключ та розшифровує відповідні файли. Після шифрування/розшифрування файли мають відкриватися будь яким призначеним для цього застосунком, наприклад, MSPaint.

Примітка: для генерації ключів можна написати окремий застосунок, що зберігає ключі в окремих файлах.

Зміст звіту:

- код програми згідно індивідуального варіанту;
- скріншоти, що демонструють роботу застосунку;
- вміст файлів з ключами шифрування;
- відповіді на контрольні питання.

Контрольні питання:

– в чому відмінність асиметричних алгоритмів шифрування від симетричних?

– коротко опишіть основні кроки алгоритму RSA;

– яка сфера використання асиметричних алгоритмів шифрування?

– як створити ключі для алгоритму RSA?

5 ЛАБОРАТОРНА РОБОТА № 5. ВИКОРИСТАННЯ ФУНКЦІЙ ХЕШУВАННЯ. ЕЛЕКТРОННИЙ ЦИФРОВИЙ ПІДПИС

Мета роботи: знайомство з функціями хешування та електронним цифровим підписом на основі алгоритму асиметричного шифрування RSA. Програмування типового застосунку, в якому повідомлення спочатку підписується електронним цифровим підписом, а потім перевіряється на справжність.

Коротка теорія.

Одним з варіантів використання асиметричних криптосистем є електронний цифровий підпис – значення, що обчислюється з послідовності байтів з використанням секретного ключа. Він свідчить про те, що власник цього секретного ключа підтвердив коректність і достовірність вмісту повідомлення. Закритий ключ при цьому застосовується для шифрування, а відкритий – для розшифрування. Коротко опишемо загальний алгоритм роботи криптосистеми, де використовується електронний цифровий підпис:

а) створюється цифровий підпис:

- обчислюється дайджест (хеш) повідомлення;
- дайджест повідомлення шифрується за допомогою закритого ключа з пари "відкритий / закритий ключ", в результаті чого виходить цифровий підпис;
- повідомлення разом з цифровим підписом передається іншому учаснику криптосистеми;

б) учасником криптосистеми, що отримав підписане повідомлення, проводиться верифікація цифрового підпису:

- підпис розшифровується за допомогою відкритого ключа з вищезгаданої пари "відкритий / закритий ключ", в результаті чого відновлюється дайджест повідомлення;
- значення розшифрованого дайджесту повідомлення порівнюється зі значенням дайджесту повідомлення, обчисленого з отриманого повідомлення;
- якщо обидва значення дайджесту повідомлення збігаються, то підпис справжній, тобто отримане повідомлення не було викривлено (змінено) в процесі передачі по незахищеному каналу зв'язку. В іншому випадку або підпис, або повідомлення було змінено.

Підпис не можна підробити, адже при підписуванні повідомлення використовується секретний закритий ключ, що не розповсюджується. Справжність підпису завжди можна перевірити – будь-яка особа, що має повідомлення і цифровий підпис може перевірити, що повідомлення було підписано власником закритого ключа і, що ні власник, ні підпис не були змінені. Підпис неповторний – кожному повідомленню відповідає свій підпис. Зміна одного біту в повідомленні призведе до зміни дайджесту повідомлення, а отже і до зміни підпису. Факт підпису неможливо заперечувати – після того як повідомлення підписано і надіслано разом з підписом, особа, яка зробила це не може заперечувати, що вона підписала повідомлення, адже тільки вона є власником секретного ключа (якщо тільки не доведе, що закритий ключ було викрадено).

Розглянемо що собою являє дайджест (хеш) повідомлення. Дайджест повідомлення створюється за допомогою хеш функції – математичного алгоритму, який перетворює дані довільного розміру в бітову строку фіксованого розміру. Це перетворення є незворотним. До найвідоміших на даний момент хеш функцій відносять наступні:

- MD5 створює дайджест довжиною 128 біт. Колізії відтворюються за секунди, тому ця функція, не дивлячись на її популярність та розповсюдженість, вже не рекомендована до використання в криптосистемах;

- SHA-1 створює дайджест довжиною 160 біт вважається відносно криптостійкою, однак вже зроблено багато спроб зламати цю функцію, в результаті чого значно знижена обчислювальна складність злому. Не рекомендована до використання в надійних криптосистемах;

- SHA-2: SHA-256 створює дайджест 256 біт, SHA-512 – дайджест 512 біт. Ці функції вважають відносно криптостійкими, однак з часом криптоаналітики знайшли і в них можливість створювати колізії;

- SHA-3 створює дайджест від 224 до 512 біт. Кардинально відрізняється від SHA-1 та SHA-2. Вважається криптостійкою.

Порядок виконання роботи.

Створюємо новий проект під назвою "DS local". Розділимо логіку нашого застосунку на п'ять класів: клас *RSAProcessor* буде містити методи шифрування та дешифрування згідно до алгоритму

RSA, клас *HashProcessor* – методи створення дайджеста повідомлень, клас *FileProcessor* – методи роботи з файлами, *KeyProcessor* – методи роботи з ключами, *Main* – основний клас, в якому створюються об'єкти інших класів та демонструється робота застосунку. Почнемо з класу *HashProcessor*, код якого наведено нижче:

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class HashProcessor
{
    private byte[] hash;
    private byte[] inMessage;
    private String hashMethod;
    private MessageDigest messageDigest;

    public HashProcessor(byte[] inMessage,
                        String hashMethod)
    {
        this.inMessage = inMessage;
        if (hashMethod.equals("MD5") ||
            hashMethod.equals("SHA-1") ||
            hashMethod.equals("SHA-256") ||
            hashMethod.equals("SHA-512"))
        {
            this.hashMethod = hashMethod;
        }
        else
        {
            this.hashMethod = "SHA-256";
        }
    }

    public void setInMessage(byte[] inMessage)
    {
        this.inMessage = inMessage;
    }

    public void hash() throws NoSuchAlgorithmException
    {
        messageDigest =
            MessageDigest.getInstance(hashMethod);
```

```

        hash=messageDigest.digest(inMessage);
    }

    public byte[] getHash()
    {
        return hash;
    }
}

```

Конструктор класу приймає повідомлення для хешування та назву хеш-функції. Поля класу:

Поле	Пояснення
byte[] hash	обчислений дайджест
byte[] inMessage	повідомлення, з якого потрібно обчислити дайджест
String hashMethod	назва функції хешування
MessageDigest messageDigest	об'єкт, що реалізує функцію хешування

Основні методи класу:

Метод	Пояснення
void hash()	обчислює дайджест повідомлення
void setInMessage(byte[] inMessage)	заміняє повідомлення
byte[] getHash()	повертає обчислений хеш

Код класу *KeyProcessor* аналогічний тому, котрий описано в попередній лабораторній роботі, адже ми використовуємо той самий алгоритм RSA. Згідно з алгоритмом обчислення електронного цифрового підпису в класі *RSAProcessor* на відміну від попередньої лабораторної роботи відкритий ключ використовується для розшифрування, а закритий – для шифрування:

```

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;

```

```
import java.util.Base64;
public class RSAProcessor
{
    private PublicKey publicKey;
    private PrivateKey privateKey;
    private byte[] encryptedMessage;
    private String decryptedMessage;

    public RSAProcessor()
    {
        publicKey=null;
        privateKey=null;
    }
    public RSAProcessor(PublicKey publicKey)
    {
        this.publicKey=publicKey;
        this.privateKey=null;
    }
    public RSAProcessor(PrivateKey privateKey)
    {
        this.privateKey=privateKey;
        this.publicKey=null;
    }
    public void setPublicKey(PublicKey publicKey)
    {
        this.publicKey = publicKey;
    }
    public void setPrivateKey(PrivateKey privateKey)
    {
        this.privateKey = privateKey;
    }
    public PublicKey getPublicKey()
    {
        return publicKey;
    }
    public PrivateKey getPrivateKey()
    {
        return privateKey;
    }
    public byte[] getEncryptedMessage()
    {
        return encryptedMessage;
    }
}
```

```

public String getDecryptedMessage()
{
    return decryptedMessage;
}
//for digital signature
//encryption is processed with a private key
public void encrypt(String data) throws
BadPaddingException, IllegalBlockSizeException,
InvalidKeyException, NoSuchPaddingException,
NoSuchAlgorithmException
{
    Cipher cipher =
    Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.ENCRYPT_MODE,
    getPrivateKey());
    this.encryptedMessage=
    cipher.doFinal(data.getBytes());
}
//for digital signature
//decryption is processed with a public key
public void decrypt(byte[] data) throws
NoSuchPaddingException, NoSuchAlgorithmException,
InvalidKeyException, BadPaddingException,
IllegalBlockSizeException
{
    Cipher cipher =
    Cipher.getInstance("RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.DECRYPT_MODE,
    getPublicKey());
    this.decryptedMessage=
    new String(cipher.doFinal(data));
}

public void decrypt(String data) throws
IllegalBlockSizeException, InvalidKeyException,
BadPaddingException, NoSuchAlgorithmException,
NoSuchPaddingException
{
    decrypt(Base64.getDecoder()
    .decode(data.getBytes()));
}
}

```

Код класу для роботи з файлами *FileProcessor* ідентичний класу в попередній лабораторній роботі. Код класу *Main*, що реалізує алгоритм цифрового підпису, наведено нижче:

```
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;

public class Main
{
    public static void main(String[] args) throws
        IOException, IllegalBlockSizeException,
        InvalidKeyException, NoSuchPaddingException,
        NoSuchAlgorithmException, BadPaddingException,
        InvalidKeySpecException
    {
        //initial keys generation
        KeyProcessor initialKeyProcessor =
            new KeyProcessor(2048);
        initialKeyProcessor.generateRandomKeys();

        //writing private key to file
        FileProcessor fileProcessor = new
            FileProcessor("./private.txt");
        fileProcessor.writePrivateKeyToFile(
            initialKeyProcessor.getPrivateKey());

        //writing public key to file
        fileProcessor.setFile("./public.txt");
        fileProcessor.writePublicKeyToFile(
            initialKeyProcessor.getPublicKey());

        //writing secret message to file
        fileProcessor.setFile("./message.txt");
        String secretMessage =
            "This is my super @confidential@ string!!!";
        fileProcessor.writeMessage(
            secretMessage.getBytes());
    }
}
```

```

//encryption method.
//Reading private key from file and
// creating new keyProcessor for encryption
KeyProcessor encryptionKeyProcessor =
new KeyProcessor(2048);
fileProcessor.setFile("./private.txt");
encryptionKeyProcessor.setPrivateKey(
fileProcessor.readPrivateKeyFromFile());
RSAProcessor rsaEncryptor =
new RSAProcessor(
encryptionKeyProcessor.getPrivateKey());

//calculating the hash of secret message
HashProcessor hashProcessor =
new HashProcessor(
secretMessage.getBytes(), "SHA-256");
hashProcessor.hash();
//writing hash to console
System.out.println("Hash of initial message
is\n"+new String(hashProcessor.getHash()));

//encrypting (signing) the hash of message
rsaEncryptor.encrypt(
new String(hashProcessor.getHash()));

//write encrypted hash (signature) to file
FileProcessor messageFileProcessor =
new FileProcessor("./signature.txt");
messageFileProcessor.writeMessage(
rsaEncryptor.getEncryptedMessage());

//decryption method.
//Reading public key from file
//and creating new keyProcessor for decryption
KeyProcessor decryptionKeyProcessor =
new KeyProcessor(2048);
fileProcessor.setFile("./public.txt");
decryptionKeyProcessor.setPublicKey(
fileProcessor.readPublicKeyFromFile());
RSAProcessor rsaDecryptor =
new RSAProcessor(
decryptionKeyProcessor.getPublicKey());

```

```

//decrypting the signature
rsaDecryptor.decrypt(
messageFileProcessor.readMessage());

//write decrypted hash to console
System.out.println("Decrypted hash is\n" +
rsaDecryptor.getDecryptedMessage());

//reading the message
messageFileProcessor.setFile("./message.txt");
String readMessage =
new String(messageFileProcessor.readMessage());

//calculating the hash of read message
hashProcessor.setInMessage(
readMessage.getBytes());
hashProcessor.hash();

//writing the calculated hash to console
System.out.println("Calculated hash is\n"+
new String(hashProcessor.getHash()));

//evaluating the identity of signature
if (rsaDecryptor.getDecryptedMessage()
.equals(new String(hashProcessor.getHash())))
{
    System.out.println("Hashes are equal.
    Digital signature has been checked.
    The message is real");
}
else
{
    System.out.println("Hashes are not equal.
    The message has been compromised");
}
}
}

```

Завдання для самостійного виконання:

Варіант 1: Створіть застосунок, що дозволяє обчислювати обрану хеш-функцію обраного файлу чи папки з файлами. Застосунок має містити консольне меню, в якому користувач обирає хеш-

функцію, наприклад, 1 – MD5, 2 – SHA-1 і так далі. Далі в консолі вводиться шлях до файлу чи папки та обчислюється дайджест, який записується в окремий файл, ім'я якого має збігатися з оригінальним ім'ям файлу чи папки з файлами. Якщо користувач обрав папку, то необхідно в потоках (threads) паралельно обчислити дайджести всіх файлів в папці.

Примітка: для створення потоку виконання необхідно написати клас, який реалізує інтерфейс *Runnable*, та реалізувати метод *run()*, код якого буде виконуватися окремим потоком.

Варіант 2: Створіть розподілений застосунок, в якому сервер відправляє клієнтам підписані повідомлення (повідомлення вводяться з консолі). Клієнти після перевірки справжності повідомлення показують його на консоль.

Примітка: спочатку згенеруйте ключі, потім закритий ключ залиште на сервері для підписування повідомлень, а відкритий ключ розповсюдьте по клієнтах.

Зміст звіту:

- код програми згідно індивідуального варіанту;
- скріншоти, що демонструють роботу застосунку;
- вміст файлів з ключами шифрування, дайджестом повідомлення, електронним цифровим підписом;
- відповіді на контрольні питання.

Контрольні питання:

- в чому полягає призначення електронного цифрового підпису?
- яка роль алгоритму RSA в електронному цифровому підписі?
- що таке хеш-функція?
- наведіть детальний алгоритм будь-якої хеш-функції.

ЛІТЕРАТУРА

1. Остапов С. Технології захисту інформації. Посібник / С. Остапов. – Родовід, 2014. – 428 с.
2. Нестеров С. Основы информационной безопасности. Учебное пособие / С. Нестеров. – СПб. : Изд-во Политехн. Ун-та, 2016. – 324 с.
3. Бондарев В. Введение в информационную безопасность автоматизированных систем. Учебное пособие / В. Бондарев. – М. : МГТУ им. Н. Э. Баумана, 2016. – 252 с.
4. Антонюк А. О., Жора В. В. Теоретичні основи моделювання та аналізу систем захисту інформації / А. О. Антонюк, В. В. Жора. – Ірпінь, Національний університет ДПС України, 2010. – 310 с.
5. Шорошев В. В. Теоретичні і практичні аспекти організації і побудови архітектури захищених комп'ютерних систем. Монографія. – К.: ДУПСТ, 2011. – 257 с.
6. Бутузов В. М. Протидія комп'ютерній злочинності в Україні (системно-структурний аналіз): монографія / В.М. Бутузов. – К. : КИТ, 2010. - 408 с.
7. Захарченко В.Ю., Лазуренко В.И., Олифиров А.В., Рогозин С.Н. Компьютерные преступления: их выявление и предотвращение: Учебное пособие /Под общ. редакцией В.И. Лазуренко.– К: Центр учебной литературы, 2007. – 170 с.
8. Лісневський М. М. Технології комп'ютерної безпеки. Монографія. Книга 2. Науковий керівник Р.М.Літнарівч. МЕНУ, Рівне, 2012.– 100 с.
9. Коханович Г. Захист інформації в телекомунікаційних системах. – МК-Пресс, 2005, 288 с.
10. Ленков, С.В. Методы и средства защиты информации. В 2-х томах / Ленков С.В., Перегудов Д.А., Хорошко В.А., Под ред. В.А. Хорошко. – К. : Арий, 2008
11. Кавун С. В. Інформаційна безпека: підручник / С.В. Кавун. – Харків : Вид. ХНЕУ, 2009. - 368 с.