3 Kryptografia stosowana – szyfrowanie RSA i bezpieczna komunikacja

Zgłębiaj kryptografię stosowaną poprzez implementację systemu bezpiecznej komunikacji opartego na szyfrowaniu RSA z kluczem publicznym. Celem jest stworzenie działającego prototypu oprogramowania, który może generować klucze kryptograficzne, szyfrować i deszyfrować wiadomości, a opcjonalnie integrować szyfrowanie symetryczne dla efektywnego transferu danych. Rdzeniem projektu jest kryptosystem RSA, który opiera się na podstawowych algorytmach teorii liczb. RSA jest szeroko stosowanym schematem kryptografii klucza publicznego do bezpiecznej transmisji danych, bazującym na matematycznym fakcie, że chociaż łatwo jest pomnożyć dwie duże liczby pierwsze, niezwykle trudno jest rozłożyć ich iloczyn na czynniki pierwsze.

Zaimplementuj generowanie kluczy RSA (w tym generowanie liczb pierwszych i obliczanie odwrotności modularnych), funkcje szyfrowania i deszyfrowania, a następnie wykorzystaj je do zabezpieczenia prostego systemu wymiany wiadomości lub przechowywania danych. Projekt wymaga ścisłej matematycznej wiedzy: konieczne będzie zrozumienie arytmetyki modularnej, faktoryzacji liczb oraz własności funkcji Eulera. Zespoły muszą zapewnić poprawność implementacji (np. weryfikując, że szyfrowanie i deszyfrowanie są do siebie odwrotne pod wygenerowanymi kluczami) oraz rozważyć wydajność (arytmetyka dużych liczb i optymalizacja dla dużych liczb pierwszych). Opcjonalnie projekt może obejmować szyfr symetryczny (np. AES), aby szyfrować dane masowe, natomiast RSA posłuży do wymiany klucza symetrycznego – naśladując rzeczywiste protokoły bezpiecznej komunikacji. Finalny system powinien demonstrować cały proces bezpiecznej komunikacji: generowanie kluczy, szyfrowanie, transmisję (zasymulowaną) oraz deszyfrowanie — wszystko zaimplementowane przez zespół (najlepiej w C++).

3.1 Wymagane komponenty i algorytmy

1. Generowanie kluczy RSA

Zaimplementuj algorytm generowania kluczy RSA, zapewniając, że:

- Generowane liczby pierwsze mają co najmniej 16 bitów do celów demonstracyjnych, a najlepiej 1024 bity lub więcej dla rzeczywistego bezpieczeństwa.
- Do testowania pierwszości używany jest probabilistyczny test pierwszości (np. Miller-Rabin) lub odpowiednia funkcja z biblioteki.
- Wynikowymi kluczami są klucz publiczny i klucz prywatny.
- Obsługiwane są liczby wielkiej precyzji (użyj np. boost::multiprecision::cpp_int lub zaimplementuj własne wsparcie).

2. Szyfrowanie/Deszyfrowanie RSA

- Zaimplementuj funkcje szyfrowania i deszyfrowania RSA (wymagana szybka potęgowanie modularne – użyj metody szybkiego potęgowania przez podnoszenie do kwadratu).

3. Właściwe formatowanie/padding wiadomości

- Do tekstów można użyć uproszczonego kodowania (np. konwersja bajtów na liczby bez złożonych schematów paddingu), ale należy być świadomym implikacji bezpieczeństwa.

4. Integracja szyfrowania symetrycznego (opcjonalnie)

Aby zasymulować pełną sesję bezpiecznej komunikacji:

- Użyj standardowego szyfru symetrycznego, takiego jak AES, do szyfrowania właściwych danych pod losowym kluczem symetrycznym.
- Klucz symetryczny zaszyfruj RSA (jest to klasyczna metoda szyfrowania hybrydowego).
- Odbiorca używa RSA do odszyfrowania klucza symetrycznego, a następnie używa go do odszyfrowania właściwej wiadomości.

5. Protokół bezpiecznej komunikacji

Stwórz prosty protokół lub scenariusz aplikacyjny, np.:

- Program konsolowy, w którym użytkownik generuje klucze, publikuje klucz publiczny, a inny użytkownik szyfruje nim wiadomość. Wiadomość jest „przesyłana” przez plik lub konsolę, a odbiorca używa klucza prywatnego do jej odszyfrowania.
- Alternatywnie, symulacja klient-serwer z kluczem publicznym/ prywatnym po stronie serwera.

Protokół powinien obejmować kroki wymiany kluczy, szyfrowania danych i ich deszyfrowania.

6. Walidacja i testowanie

Przeprowadź obszerne testy:

- Test algorytmu rozszerzonego Euklidesa.
- Test szyfrowania/deszyfrowania RSA na małych, ręcznie liczonych przykładach.
- Testy różnych długości wiadomości, w tym danych binarnych.
- Test szyfrowania hybrydowego, jeśli zaimplementowane.

7. Analiza wydajności

Mierz czas generowania kluczy, szyfrowania i deszyfrowania, zwłaszcza przy różnych rozmiarach kluczy.

8. Rozważania dotyczące bezpieczeństwa

W dokumentacji omów:

- Dlaczego RSA jest bezpieczne (trudność faktoryzacji).
- Znaczenie stosowania paddingu w aplikacjach rzeczywistych.
- Różnice między kryptografią symetryczną i asymetryczną.
- Ograniczenia implementacji (np. mniejsze klucze, brak paddingu).

3.2 Ostateczne materiały do oddania

1. Zaimplementowane oprogramowanie

Pełny działający program kryptograficzny (preferowane C++), obejmujący:

- Generowanie kluczy RSA
- Szyfrowanie/dekryptowanie RSA
- Opcjonalnie szyfrowanie symetryczne i szyfrowanie hybrydowe
- Prosty interfejs użytkownika (np. CLI)

2. Repozytorium Git

Zawiera:

- Kod źródłowy
- Strukturalny podział modułów (RSA, szyfr symetryczny itd.)
- README z instrukcją kompilacji, uruchamiania i przykładami użycia

3. Dokumentacja techniczna (ok. 10 stron)

Sekcje:

- **Wprowadzenie** – cel projektu, opis RSA
- **Podłoże matematyczne** – liczby pierwsze, arytmetyka modularna, funkcja Eulera, trudność faktoryzacji
- **Projekt i implementacja** – opis algorytmów, pseudokod, architektura
- **Wyniki** – przykłady działania, testy, pomiary czasu

- **Analiza bezpieczeństwa** – ograniczenia implementacji
- **Wnioski** – podsumowanie i możliwe rozszerzenia

4. Podręcznik użytkownika

Krótka instrukcja obsługi programu z przykładami wejścia i wyjścia.

5. Prezentacja/Demo (opcjonalnie)

3.3 Wymagania dotyczące języka i narzędzi

- **Język programowania:** C++ (preferowany) lub C. Python niewskazany.
- **Obsługa dużych liczb:** np. Boost.Multiprecision, GMP.
- **Losowość:** używaj kryptograficznie bezpiecznych generatorów (np. std::random_device).
- **Praktyki programistyczne:** użycie Gita, recenzje kodu, testy jednostkowe.
- **Narzędzia testujące:** testy wektorów RSA, debugowanie, narzędzia do sprawdzania pamięci.
- **Dokumentacja:** wyraźne komentarze, staranny styl kodu, zachęta do użycia LaTeX.