

Dokumentacja Projektu CryptoLab Mobile

Agnieszka Ryś

9 grudnia 2025

Spis treści

1 Cel projektu

Celem aplikacji **CryptoLab Mobile** jest edukacja w zakresie kryptografii. Aplikacja mobilna pozwala szyfrować i deszyfrować teksty oraz pliki .txt, sprawdzać poprawność kluczy oraz eksportować wyniki. Aplikacja implementuje zarówno klasyczne szyfry historyczne (Cezara, Vigenère'a, szyfr z kluczem bieżącym), nowoczesny standard szyfrowania symetrycznego AES (Advanced Encryption Standard), jak i przełomowy algorytm kryptografii asymetrycznej RSA (Rivest-Shamir-Adleman). Wszystkie algorytmy są implementowane ręcznie, bez użycia gotowych bibliotek kryptograficznych, co pozwala na głębsze zrozumienie ich działania i różnic między kryptografią symetryczną a asymetryczną.

2 Podstawy kryptografii klasycznej

Wprowadzenie

Kryptografia to dziedzina zajmująca się ochroną informacji poprzez jej przekształcanie w formę nieczytelną dla osób nieuprawnionych. Jej historia sięga starożytności, gdzie stosowano proste metody szyfrowania, znane obecnie jako **kryptografia klasyczna**. Celem było zapewnienie poufności korespondencji wojskowej, dyplomatycznej czy handlowej.

2.1 Kryptografia symetryczna

W kryptografii symetrycznej ten sam klucz służy zarówno do szyfrowania, jak i deszyfrowania wiadomości. Najważniejsze cechy:

- wysoka szybkość działania,
- konieczność bezpiecznej wymiany klucza,
- podatność na ataki brute-force przy krótkich kluczach.

2.2 Przykłady szyfrów klasycznych

- **Szyfr Cezara** – przesunięcie liter alfabetu o stałą liczbę pozycji,
- **Szyfr Vigenère'a** – wieloalfabetyczny szyfr wykorzystujący słowo-klucz,
- **Szyfr z kluczem bieżącym** – rozwinięcie Vigenère'a z długim kluczem tekstowym,
- **Szyfr podstawieniowy (monoalfabetyczny)** – każdej literze alfabetu przypisana jest inna litera,
- **Szyfr Playfair** – operujący na parach liter,
- **Szyfr transpozycyjny** – zmienia kolejność znaków w wiadomości.

2.3 Znaczenie w edukacji

Choć współcześnie klasyczne szyfry nie zapewniają realnego bezpieczeństwa, stanowią doskonałe narzędzie dydaktyczne. Pozwalają zrozumieć podstawowe pojęcia kryptografii, takie jak:

- **klucz** – parametr definiujący szyfrowanie,
- **przestrzeń kluczy** – zbiór możliwych wartości klucza,
- **analiza częstości** – klasyczna metoda łamania szyfrów,
- **brute-force** – przeszukiwanie wszystkich możliwych kluczy.

2.4 Nowoczesna kryptografia symetryczna – AES

Aplikacja CryptoLab zawiera również implementację **AES (Advanced Encryption Standard)**, który jest standardem współczesnego szyfrowania symetrycznego. W przeciwieństwie do szyfrów klasycznych, AES:

- jest szyfrem **blokowym** (operuje na blokach 128 bitów),
- wykorzystuje złożone operacje matematyczne (S-Box, MixColumns, ShiftRows),
- oferuje różne długości kluczy (128, 192, 256 bitów),
- obsługuje różne tryby pracy (ECB, CBC, CTR),
- jest odporny na wszystkie znane praktyczne ataki kryptograficzne.

Dzięki implementacji zarówno szyfrów klasycznych, jak i nowoczesnego AES, użytkownicy mogą porównać podejścia historyczne z obecnie stosowanymi rozwiązaniami i zrozumieć ewolucję kryptografii.

3 Miejsce szyfru Cezara

Szyfr Cezara należy do najprostszych szyfrów podstawieniowych. Choć jego bezpieczeństwo jest znikome, odgrywa on kluczową rolę w nauczaniu, ponieważ wprowadza intuicyjnie pojęcia klucza, szyfrowania i deszyfrowania. CryptoLab wykorzystuje go jako **pierwszy krok** w implementacji i analizie algorytmów kryptograficznych.

4 Technologie wykorzystane w projekcie

React Native + Expo Główna platforma wykorzystana do tworzenia aplikacji mobilnych. React Native umożliwia budowanie natywnych aplikacji na systemy Android i iOS, wykorzystując składnię zbliżoną do Reacta. Expo zostało użyte jako narzędzie wspierające proces developmentu – upraszcza konfigurację środowiska, przyspiesza testowanie na urządzeniach mobilnych i zapewnia dostęp do bogatego ekosystemu bibliotek.

TypeScript Nadzbiór JavaScriptu wprowadzający system typów. Zastosowanie TypeScriptu pozwoliło na:

- wcześniejsze wykrywanie błędów podczas komplikacji,
- lepszą kontrolę nad strukturą danych i interfejsami,
- zwiększoną czytelność oraz przewidywalność kodu,

Expo Document Picker, File System, Vector Icons Dodatkowe biblioteki środowiska Expo:

- `expo-document-picker` – umożliwia wybór plików z pamięci urządzenia,
- `expo-file-system` – zapewnia dostęp do systemu plików (zapisywanie, odczyt, usuwanie plików),
- `expo-vector-icons` – biblioteka ikon pozwalająca wzbogacić interfejs użytkownika.

Git System kontroli wersji użyty do zarządzania historią kodu. Pozwolił na prowadzenie szczegółowego changelogu, śledzenie postępów w projekcie oraz łatwe zarządzanie zmianami w kodzie źródłowym.

LaTeX System składu tekstu wykorzystany do przygotowania dokumentacji. Umożliwia on:

- zachowanie spójności formatowania,
- wygodne dodawanie fragmentów kodu źródłowego i zrzutów ekranu,
- automatyczne generowanie spisów treści i numeracji.

5 Architektura systemu

5.1 Wzorzec projektowy

Aplikacja wykorzystuje **Strategy Pattern** dla algorytmów kryptograficznych. Każdy algorytm dziedziczy z klasy abstrakcyjnej **CryptographicAlgorithm** i implementuje metody:

- `encrypt(plaintext, key)` – szyfruje tekst,
- `decrypt(ciphertext, key)` – deszyfruje tekst,
- `validateKey(key)` – sprawdza poprawność klucza,
- `getKeyRequirements()` – zwraca opis wymagań dla klucza.

Wszystkie algorytmy zarejestrowane są w **AlgorithmRegistry** (Singleton Pattern), co umożliwia łatwe dodawanie nowych szyfrów bez modyfikacji głównej aplikacji.

5.2 Komponenty główne

- **App.tsx** – główny komponent aplikacji, obsługuje interfejs użytkownika,
- **AlgorithmSidebar.tsx** – boczny panel z listą dostępnych algorytmów,
- **LogsViewer.tsx** – komponent wyświetlania historii operacji kryptograficznych,
- **AlgorithmRegistry.ts** – rejestr i zarządzanie algorytmami,
- **CryptographicAlgorithm.ts** – klasa bazowa dla wszystkich algorytmów,
- **LogManager.ts** – menadżer logów z wzorcem Singleton,
- **fileUtils.ts** – funkcje do obsługi operacji na plikach.

6 Struktura projektu

```
crypto-lab-mobile/
    App.tsx                               (główny komponent)
    package.json                           (zależności projektu)
    tsconfig.json                          (konfiguracja TypeScriptu)
    app.json                                (konfiguracja Expo)
    src/
        algorithms/
            CryptographicAlgorithm.ts   (klasa bazowa)
            CaesarCipher.ts           (szyfr Cezara)
            VigenereCipher.ts         (szyfr Vigenere'a)
            RunningKeyCipher.ts       (szyfr z kluczem
                                         bieżącym)
            AESCipher.ts              (szyfr AES)
            RSACipher.ts              (szyfr RSA)
            AlgorithmRegistry.ts      (rejestr algorytmów)
        components/
            AlgorithmSidebar.tsx      (panel z algorytmami)
```

	LogsViewer.tsx	(wyswietlanie logow)
	types/	
	LogTypes.ts	(typy dla systemu)
logow)		
	utils/	
	fileUtils.ts	(obsługa plikow)
	LogManager.ts	(zarzadzanie logami)
	assets/	(zasoby graficzne)

7 Implementacja szyfru Cezara

7.1 Podstawy

Szyfr Cezara to prosty szyfr monoalfabetyczny, w którym litery przesuwane są o wartość klucza k . Przestrzeń kluczy obejmuje wartości 1–25. Metoda jest podatna na ataki brute-force i analizę częstotliwości.

7.2 Model matematyczny

- Szyfrowanie: $E_k(x) = (x + k) \bmod 26$,
- Deszyfrowanie: $D_k(x) = (x - k) \bmod 26$.

7.3 Cechy implementacji

- Obsługuje zarówno wielkie jak i małe litery,
- Znaki niebędące literami pozostają bez zmian,
- Klucz musi być liczbą całkowitą z zakresu 1–25,
- Weryfikacja klucza zwraca szczegółową informację o błędach.

8 Implementacja szyfru Vigenère'a

8.1 Historia i znaczenie

Szyfr Vigenère'a został opracowany w XVI wieku przez Blaise de Vigenère'a. Przez długi czas uważany był za niezniszczalny (*le chiffre indéchiffrable*) aż do jego przełamania przez Charles'a Babbage'a w XIX wieku.

8.2 Podstawy

Szyfr Vigenère'a to szyfr **polialfabetyczny**, który wykorzystuje słowo-klucz do generowania serii przesunięć. W przeciwieństwie do szyfru Cezara, każda litera tekstu może być szyfrowana z innym przesunięciem.

8.3 Model matematyczny

- Szyfrowanie: $E_k(x_i) = (x_i + k_{i \bmod |k|}) \bmod 26$,
- Deszyfrowanie: $D_k(y_i) = (y_i - k_{i \bmod |k|}) \bmod 26$,
- gdzie k to słowo-klucz, a $|k|$ to jego długość.

8.4 Przykład działania

Tekst jawny	A	T	T	A	C	K
Klucz	L	E	M	O	N	L
Przesunięcia	+11	+4	+12	+14	+13	+11
Tekst zaszyfrowany	L	X	F	O	P	V

8.5 Cechy implementacji

- Klucz może zawierać tylko litery (A-Z, a-z),
- Klucz nie może być pusty,
- Znaki niebędące literami w tekście źródłowym są przepisywane bez zmian,
- Klucz automatycznie się powtarza dla długich tekstów,
- Obsługuje zarówno wielkie jak i małe litery w tekście.

9 Implementacja szyfru z kluczem bieżącym

9.1 Historia i zastosowanie

Szyfr z kluczem bieżącym (Running Key Cipher) to rozwinięcie szyfru Vigenère'a. Zamiast krótko słowa, wykorzystuje on klucz o długości co najmniej równej długości tekstu. Gdy klucz jest naprawdę losowy i będzie użyty tylko raz, szyfr ten jest teoretycznie nie do złamania (jest to wariant szyfru jednorazowego – *One-Time Pad*).

9.2 Podstawy

Algorytm jest w zasadzie identyczny z szyfrem Vigenère'a, ale z istotną różnicą: klucz powinien być znacznie dłuższy niż tekst. W praktyce zastosowania edukacyjnego aplikacja automatycznie generuje klucz z tekstu Lorem Ipsum.

9.3 Model matematyczny

- Szyfrowanie: $E_k(x_i) = (x_i + k_i) \bmod 26$,
- Deszyfrowanie: $D_k(y_i) = (y_i - k_i) \bmod 26$,
- gdzie $|k| \geq |x|$ (klucz jest co najmniej tak długi jak tekst).

9.4 Cechy implementacji

- Klucz może zawierać litery i spacje,
- Klucz musi zawierać co najmniej 5 liter,
- Aplikacja automatycznie generuje losowy klucz na bazie Lorem Ipsum,
- Zaszyfrowany tekst zawiera klucz w formacie: <klucz>:<tekst_zaszyfrowany>,
- Deszyfrowanie wymaga podania tekstu w poprawnym formacie.

9.5 Bezpieczeństwo

- Gdy klucz jest losowy i używany tylko raz, szyfr jest teoretycznie bezpieczny,
- Słaba strona: jeśli klucz jest krótszy niż tekst, powtarza się i traci bezpieczeństwo,
- W aplikacji edukacyjnej klucz jest generowany automatycznie i przechowywany w wynikach.

10 Implementacja szyfru AES

10.1 Historia i znaczenie

AES (Advanced Encryption Standard) to symetryczny szyfr blokowy, który w 2001 roku został wybrany przez NIST (National Institute of Standards and Technology) jako następca przestarzałego algorytmu DES. Został opracowany przez belgijskich kryptografów Joana Daemena i Vincenta Rijmena pod nazwą *Rijndael*. AES jest obecnie najpowszechniej stosowanym szyfrem symetrycznym na świecie – chroni dane w protokołach SSL/TLS, systemach bankowych, szyfrowanych dyskach i wielu innych zastosowaniach.

10.2 Podstawy

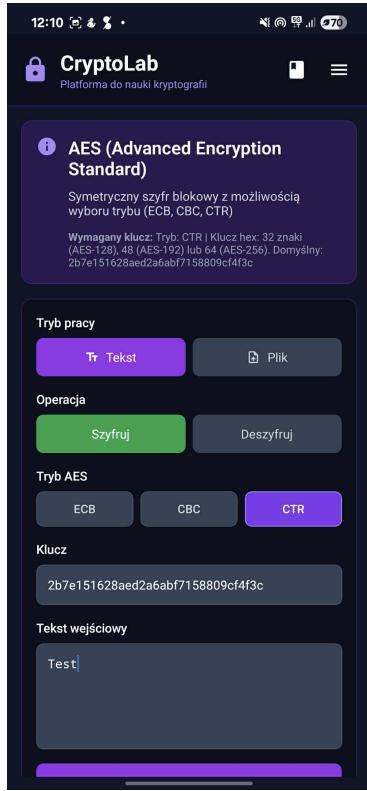
AES to szyfr **blokowy**, który operuje na blokach danych o długości 128 bitów (16 bajtów). W przeciwieństwie do szyfrów klasycznych, AES wykorzystuje skomplikowane operacje matematyczne na macierzach bajtów, w tym podstawienia (S-Box), permutacje, mieszanie kolumn i dodawanie klucza rundowego.

10.3 Warianty AES

AES występuje w trzech wariantach, różniących się długością klucza:

- **AES-128** – klucz 128-bitowy (32 znaki hex), 10 rund szyfrowania,
- **AES-192** – klucz 192-bitowy (48 znaków hex), 12 rund szyfrowania,
- **AES-256** – klucz 256-bitowy (64 znaki hex), 14 rund szyfrowania.

Im dłuższy klucz, tym wyższe bezpieczeństwo, ale także nieznacznie wolniejsze działanie.



Rysunek 1: Strona główna szyfru AES w aplikacji CryptoLab

Rysunek ?? przedstawia interfejs użytkownika aplikacji CryptoLab Mobile dla szyfru AES.

10.4 Tryby pracy AES

Szyfr blokowy wymaga określenia **trybu pracy**, który definiuje sposób szyfrowania wielu bloków danych:

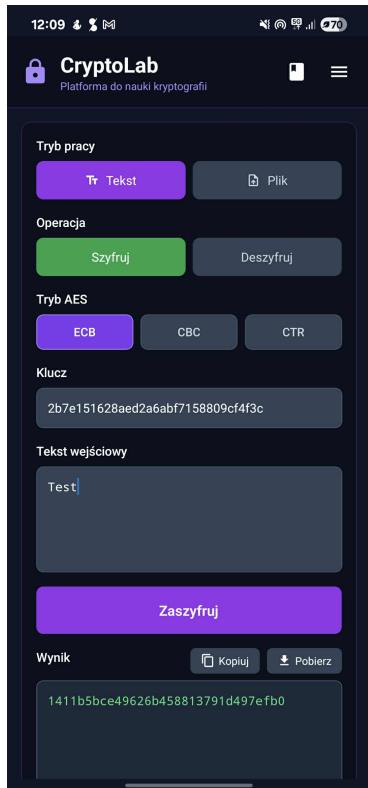
ECB (Electronic Codebook) Najprostszy tryb – każdy blok szyfrowany jest niezależnie tym samym kluczem. **Niezalecany** w praktyce, ponieważ identyczne bloki tekstu jawnego dają identyczne bloki szyfrogramu, co może ujawnić wzorce w danych. Rysunek ?? przedstawia przykładowe szyfrowanie tekstu w trybie ECB.

CBC (Cipher Block Chaining) Każdy blok tekstu jawnego jest najpierw XOR-owany z poprzednim blokiem szyfrogramu przed zaszyfrowaniem. Wymaga wektora inicjalizującego (IV). Tryb ten ukrywa wzorce w danych i jest szeroko stosowany. Rysunek ?? przedstawia przykładowe deszyfrowanie tekstu w trybie CBC.

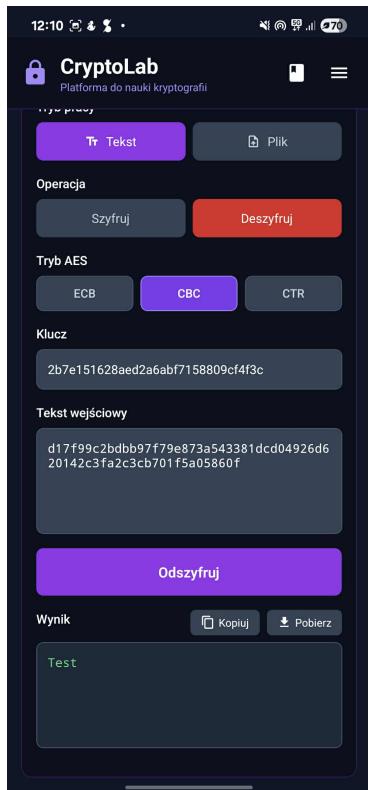
CTR (Counter Mode) Przekształca szyfr blokowy w szyfr strumieniowy. Szyfruje kolejne wartości licznika, a wyniki XOR-uje z blokami tekstu jawnego. Umożliwia równoległe szyfrowanie i deszyfrowanie. Rysunek ?? przedstawia przykładowe szyfrowanie tekstu w trybie CTR.

10.5 Struktura algorytmu AES

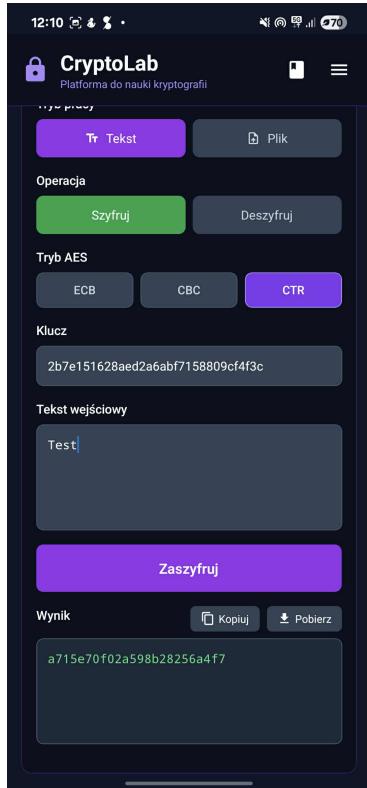
Algorytm AES składa się z następujących kroków (dla każdej rundy):



Rysunek 2: Tryb ECB w szyfrze AES



Rysunek 3: Tryb CBC w szyfrze AES



Rysunek 4: Tryb CTR w szyfrze AES

1. **SubBytes** – podstawienie bajtów zgodnie z tablicą S-Box,
2. **ShiftRows** – przesunięcie wierszy macierzy stanu,
3. **MixColumns** – mieszanie kolumn macierzy (pomijane w ostatniej rundzie),
4. **AddRoundKey** – dodanie klucza rundowego (operacja XOR).

Przed pierwszą rundą wykonywana jest operacja **AddRoundKey** z kluczem początkowym.

10.6 Cechy implementacji

- Obsługuje klucze w formacie szesnastkowym (hex),
- Klucz musi mieć długość 32, 48 lub 64 znaki hex (AES-128/192/256),
- Domyślny klucz: `2b7e151628aed2a6abf7158809cf4f3c` (AES-128),
- Implementuje trzy tryby pracy: ECB, CBC, CTR,
- Używa paddingu PKCS#7 dla dopełnienia bloków,
- Generuje losowy wektor inicjalizujący (IV) dla trybów CBC i CTR,
- Wynik szyfrowania zwracany w formacie hex,
- Pełna implementacja bez użycia zewnętrznych bibliotek kryptograficznych.

10.7 Bezpieczeństwo

- AES jest uważany za **kryptograficznie bezpieczny** przy prawidłowym użyciu,
- Nie znaleziono praktycznych ataków na pełny AES-128, AES-192 ani AES-256,
- Teoretyczne ataki istnieją, ale wymagają zasobów przekraczających możliwości obecnej technologii,
- Bezpieczeństwo zależy od:
 - wyboru odpowiedniego trybu pracy (CBC lub CTR zamiast ECB),
 - użycia losowego IV dla trybów CBC i CTR,
 - odpowiedniej długości klucza (zalecane minimum: AES-128),
 - bezpiecznego przechowywania i dystrybucji klucza.

10.8 Zastosowania

AES jest wykorzystywany w:

- szyfrowanie połączeń internetowych (HTTPS, SSL/TLS),
- pełne szyfrowanie dysków (BitLocker, FileVault),
- sieci bezprzewodowe (WPA2, WPA3),
- aplikacje bankowe i systemy płatności,
- komunikatory szyfrowane (Signal, WhatsApp),
- archiwizacja danych (7-Zip, WinRAR z szyfrowaniem AES).

11 Kryptografia asymetryczna – RSA

11.1 Historia i znaczenie

RSA (Rivest-Shamir-Adleman) to pierwszy praktyczny algorytm kryptografii asymetrycznej, opublikowany w 1977 roku przez Rona Rivesta, Adi Shamira i Leonarda Adlemana z MIT. RSA rozwiązał fundamentalny problem kryptografii symetrycznej: **bezpieczną wymianę kluczy**.

W kryptografii asymetrycznej każdy użytkownik posiada parę kluczy:

- **klucz publiczny** – może być swobodnie udostępniany, służy do szyfrowania,
- **klucz prywatny** – musi być tajny, służy do deszyfrowania.

To rewolucyjne podejście umożliwiło bezpieczną komunikację bez wcześniejszej wymiany tajnego klucza.

11.2 Podstawy matematyczne

Bezpieczeństwo RSA opiera się na trudności **faktoryzacji dużych liczb złożonych**. Łatwo jest pomnożyć dwie duże liczby pierwsze, ale bardzo trudno rozłożyć ich iloczyn na czynniki pierwsze.

11.3 Model matematyczny

Generowanie kluczy:

1. Wybierz dwie duże liczby pierwsze: p i q
2. Oblicz moduł: $n = p \cdot q$
3. Oblicz funkcję Eulera: $\phi(n) = (p - 1)(q - 1)$
4. Wybierz wykładnik publiczny e , taki że: $1 < e < \phi(n)$ oraz $\text{nwd}(e, \phi(n)) = 1$
5. Oblicz wykładnik prywatny d , taki że: $d \cdot e \equiv 1 \pmod{\phi(n)}$
6. Klucz publiczny: (e, n)
7. Klucz prywatny: (d, n)

Szyfrowanie i deszyfrowanie:

- Szyfrowanie (klucz publiczny): $c = m^e \pmod{n}$
- Deszyfrowanie (klucz prywatny): $m = c^d \pmod{n}$
- gdzie m – wiadomość, c – szyfrogram

11.4 Przykład działania

Niech $p = 61$, $q = 53$: $n = 61 \cdot 53 = 3233$

$$\phi(n) = 60 \cdot 52 = 3120$$

$$e = 17 \quad (\text{nwd}(17, 3120) = 1)$$

$$d = 2753 \quad (17 \cdot 2753 \equiv 1 \pmod{3120})$$

Klucz publiczny: (17, 3233)

Klucz prywatny: (2753, 3233)

Szyfrowanie litery 'A' (kod ASCII: 65):

$$c = 65^{17} \pmod{3233} = 2790$$

Deszyfrowanie:

$$m = 2790^{2753} \pmod{3233} = 65$$

11.5 Cechy implementacji

- Generowanie par kluczy z losowymi liczbami pierwszymi,
- Dla celów edukacyjnych używa małych liczb pierwszych (100-300),
- W praktyce RSA wymaga liczb o długości 2048+ bitów,
- Implementuje algorytm Euklidesa dla obliczenia NWD,
- Rozszerzony algorytm Euklidesa dla odwrotności modularnej,
- Szybkie potęgowanie modularne dla efektywnego szyfrowania,
- Format kluczy: "wykładnik,moduł" (np. "17,3233"),
- Każdy znak tekstu szyfrowany osobno,
- Wynik w postaci liczb rozdzielonych spacjami.

11.6 Jak wygenerować klucze RSA

Aplikacja mobilna posiada wbudowaną funkcję generowania kluczy RSA bezpośrednio w interfejsie użytkownika.

Generowanie kluczy w aplikacji (zalecane):

1. Wybierz algorytm RSA z listy
2. Kliknij przycisk "**Generuj klucze**" obok pola klucza
3. Aplikacja wyświetli okno z wygenerowanymi kluczami:
 - Klucz publiczny (do szyfrowania)
 - Klucz prywatny (do deszyfrowania)
4. Możesz skopiować klucze lub bezpośrednio użyć jednego z nich
5. Zapisz oba klucze w bezpiecznym miejscu!

Opcja 1: Użycie przykładowych kluczy testowych

- Klucz publiczny: 17,323 (e=17, n=323)
- Klucz prywatny: 233,323 (d=233, n=323)

Opcja 2: Wygenerowanie własnych kluczy w konsoli przeglądarki

Listing 1: Generowanie kluczy RSA w konsoli

```
// Skopiuj kod RSACipher do konsoli, a następnie:  
const rsa = new RSACipher();  
const keys = rsa.generateKeyPair();  
console.log('Klucz publiczny:', rsa.formatPublicKey());  
console.log('Klucz prywatny:', rsa.formatPrivateKey());
```

Opcja 3: Obliczenie ręczne (cel edukacyjny)

1. Wybierz dwie małe liczby pierwsze, np. p=17, q=19
2. Oblicz $n = p \times q = 323$
3. Oblicz $(n) = (p-1)(q-1) = 16 \times 18 = 288$
4. Wybierz e takie, że $\text{NWD}(e, 288) = 1$, np. e=17
5. Oblicz $d = e^{-1} \bmod (n)$, np. d=233
6. Klucz publiczny: (17, 323), klucz prywatny: (233, 323)

11.7 Bezpieczeństwo

- RSA jest bezpieczny przy użyciu odpowiednio dużych kluczy (2048+ bitów),
- Bezpieczeństwo opiera się na trudności faktoryzacji dużych liczb,
- Zagrożenia:
 - Komputery kwantowe (algorytm Shora może złamać RSA),
 - Zbyt małe klucze (łatwa faktoryzacja),
 - Słabe generatory liczb pierwszych,
 - Ataki czasowe (timing attacks) przy nieodpowiedniej implementacji.
- Zalecenia:
 - Minimum 2048 bitów dla zastosowań praktycznych,
 - 3072-4096 bitów dla długoterminowego bezpieczeństwa,
 - Używanie sprawdzonych bibliotek kryptograficznych w produkcji.

11.8 Zastosowania

RSA jest wykorzystywany w:

- **Podpisy cyfrowe** – uwierzytelnianie dokumentów i oprogramowania,
- **Wymiana kluczy** – bezpieczne przesyłanie kluczy symetrycznych (SSL/TLS),
- **Certyfikaty SSL/TLS** – zabezpieczenie połączeń HTTPS,
- **SSH** – bezpieczne logowanie do serwerów,
- **PGP/GPG** – szyfrowanie poczty elektronicznej,
- **Blockchain** – weryfikacja transakcji w kryptowalutach.

11.9 RSA vs AES

Cecha	RSA	AES
Typ	Asymetryczny	Symetryczny
Klucze	Para: publiczny + prywatny	Jeden klucz dla obu stron
Szybkość	Wolniejszy (100-1000x)	Bardzo szybki
Rozmiar klucza	2048-4096 bitów	128-256 bitów
Wymiana klucza	Nie wymaga	Wymaga bezpiecznego kanału
Zastosowanie	Wymiana kluczy, podpisy	Szyfrowanie danych

W praktyce RSA i AES są używane razem: RSA do bezpiecznej wymiany klucza AES, a następnie AES do szyfrowania właściwych danych (*hybrid cryptography*).

12 Algorytm ElGamal

12.1 Historia i znaczenie

System szyfrowania ElGamal został zaproponowany przez Tahera ElGamala w 1985 roku. Jest to asymetryczny algorytm klucza publicznego, którego bezpieczeństwo opiera się na trudności problemu **logarytmu dyskretnego** w ciałach skończonych. Algorytm ten jest rozwinięciem protokołu wymiany kluczy Diffiego-Hellmana.

12.2 Model matematyczny

Bezpieczeństwo systemu opiera się na fakcie, że choć łatwo jest obliczyć potęgę $g^x \pmod{p}$, to bardzo trudno jest obliczyć wykładnik x , znając jedynie wynik potęgowania, podstawę i moduł (gdy liczby są odpowiednio duże).

Generowanie kluczy:

1. Wybierz dużą liczbę pierwszą p .
2. Znajdź generator g (pierwiastek pierwotny modulo p).
3. Wybierz losowy klucz prywatny x , taki że $1 < x < p - 1$.
4. Oblicz klucz publiczny $y = g^x \pmod{p}$.
5. Klucz publiczny: (p, g, y) , Klucz prywatny: (x, p) .

Szyfrowanie: Aby zaszyfrować wiadomość m dla odbiorcy z kluczem publicznym (p, g, y) :

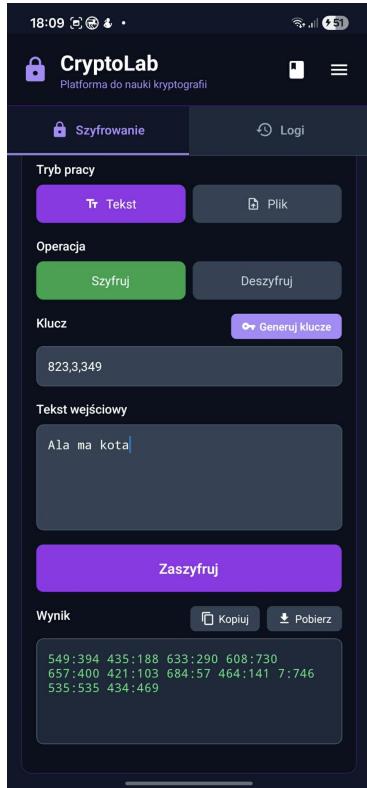
1. Wybierz losową liczbę k (klucz efemeryczny), taką że $1 < k < p - 1$.
2. Oblicz $a = g^k \pmod{p}$.
3. Oblicz $b = (y^k \cdot m) \pmod{p}$.
4. Szyfrogram to para (a, b) .

Deszyfrowanie: Aby odszyfrować parę (a, b) przy użyciu klucza prywatnego x :

1. Oblicz współczynnik $s = a^x \pmod{p}$.
2. Oblicz odwrotność $s^{-1} \pmod{p}$.
3. Wiadomość $m = b \cdot s^{-1} \pmod{p}$.

12.3 Cechy implementacji

- Implementacja wykorzystuje liczby pierwsze w zakresie 300-1000 dla celów edukacyjnych (umożliwia kodowanie znaków ASCII).
- Każdy znak wiadomości jest szyfrowany osobno, co generuje parę liczb (a, b) .
- Wykorzystuje losowy klucz k dla każdego szyfrowania, co oznacza, że ten sam tekst zaszyfrowany dwukrotnie da inny wynik (niedeterministyczność).



Rysunek 5: Ekran szyfrowania ElGamal w aplikacji CryptoLab

- Implementuje algorytm znajdowania pierwiastka pierwotnego.

Rysunek ?? przedstawia interfejs użytkownika aplikacji CryptoLab Mobile dla szyfrowania ElGamal.

Rysunek ?? przedstawia interfejs użytkownika aplikacji CryptoLab Mobile dla deszyfrowania ElGamal.

13 Protokół ECDH i Krzywe Eliptyczne

13.1 Wprowadzenie do ECC

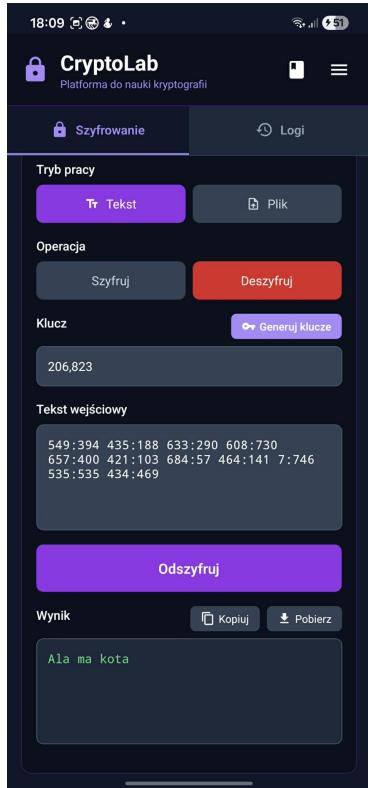
Kryptografia krzywych eliptycznych (ECC - Elliptic Curve Cryptography) opiera się na strukturze algebraicznej krzywych eliptycznych nad ciałami skończonymi. Główną zaletą ECC jest to, że zapewnia ten sam poziom bezpieczeństwa co RSA, ale przy znacznie krótszych kluczach (np. 256-bitowy klucz ECC jest porównywalny z 3072-bitowym kluczem RSA).

Równanie krzywej Weierstrassa:

$$y^2 = x^3 + ax + b \pmod{p}$$

13.2 Protokół ECDH (Elliptic Curve Diffie-Hellman)

ECDH to wariant protokołu Diffiego-Hellmana wykorzystujący krzywe eliptyczne. Służy do uzgodnienia wspólnego sekretu przez dwie strony, które nigdy się nie spotkały, korzystając z niezabezpieczonego kanału.



Rysunek 6: Ekran deszyfro ElGamal w aplikacji CryptoLab

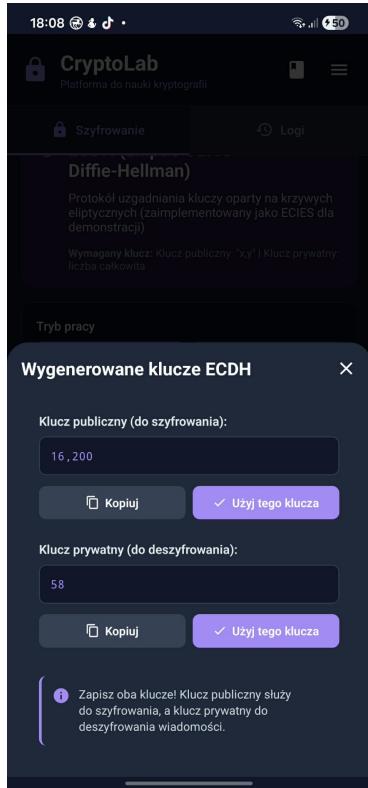
Zasada działania:

1. Strony uzgadniają parametry krzywej (punkt bazowy G , parametry a, b, p).
2. Alice generuje prywatny klucz d_A i oblicza publiczny $Q_A = d_A \cdot G$.
3. Bob generuje prywatny klucz d_B i oblicza publiczny $Q_B = d_B \cdot G$.
4. Strony wymieniają się kluczami publicznymi.
5. Alice oblicza sekret $S = d_A \cdot Q_B$.
6. Bob oblicza sekret $S = d_B \cdot Q_A$.
7. Ponieważ $d_A \cdot (d_B \cdot G) = d_B \cdot (d_A \cdot G)$, obie strony uzyskują ten sam punkt S .

13.3 Implementacja w CryptoLab (ECIES)

W aplikacji zaimplementowano schemat szyfrowania ECIES (Elliptic Curve Integrated Encryption Scheme) wykorzystujący ECDH:

- Nadawca generuje tymczasową parę kluczy (klucz efemeryczny).
- Oblicza wspólny sekret z kluczem publicznym odbiorcy.
- Współrzędna X punktu sekretnego służy jako klucz symetryczny (w uproszczeniu XOR).



Rysunek 7: Wygenerowane klucze ECDH w aplikacji CryptoLab

- Do szyfrogramu dołączany jest klucz publiczny efemeryczny, aby odbiorca mógł odtworzyć sekret.

Rysunek ?? przedstawia wygenerowane klucze ECDH w aplikacji CryptoLab Mobile. Rysunek ?? przedstawia proces szyfrowania ECDH w aplikacji CryptoLab Mobile. Rysunek ?? przedstawia proces deszyfrowania ECDH w aplikacji CryptoLab Mobile.

13.4 Cechy implementacji

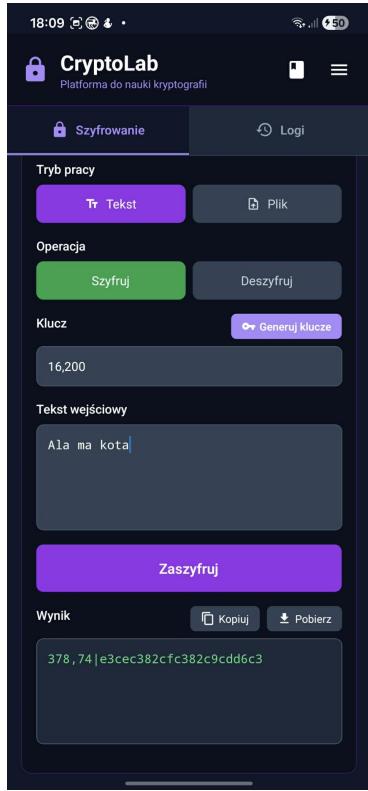
- Używa zdefiniowanej na sztywno małej krzywej eliptycznej dla celów edukacyjnych.
- Implementuje dodawanie punktów i mnożenie skalarne na krzywej.
- Wizualizuje proces generowania wspólnego sekretu w logach.

14 Funkcja skrótu SHA-256

14.1 Wprowadzenie do funkcji skrótu

Funkcje skrótu (ang. *hash functions*) to kluczowy element współczesnej kryptografii. W odróżnieniu od szyfrów, które są dwukierunkowymi przekształceniami, funkcje skrótu są **jednokierunkowe** – nie można ich odwrócić. Dla dowolnego wejścia generują skrót stałej długości.

Główne cechy funkcji skrótu:



Rysunek 8: Proces szyfrowania ECDH w aplikacji CryptoLab

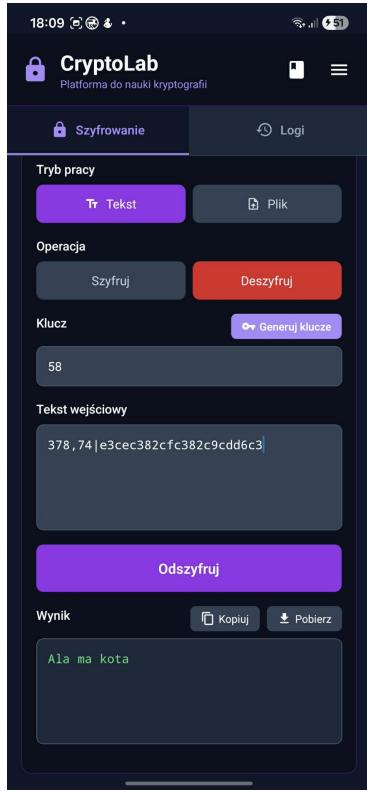
- **Jednokierunkowość** – obliczeniowo niemożliwe jest odtworzenie wiadomości ze skrótu,
- **Deterministyczność** – ta sama wiadomość zawsze daje ten sam skrót,
- **Efekt lawiny** – najmniejsza zmiana w wejściu powoduje drastyczną zmianę skrótu,
- **Odporność na kolizje** – trudno znaleźć dwie różne wiadomości dające ten sam skrót,
- **Stała długość wyjścia** – niezależnie od rozmiaru wejścia, skrót ma zawsze tę samą długość.

14.2 SHA-256 (Secure Hash Algorithm 256-bit)

SHA-256 należy do rodziny algorytmów SHA-2, opracowanych przez NSA i opublikowanych przez NIST w 2001 roku. Jest to aktualnie jeden z najpopularniejszych i najbezpieczniejszych algorytmów haszujących.

Parametry SHA-256:

- Długość skrótu: 256 bitów (64 znaki szesnastkowe),
- Rozmiar bloku: 512 bitów,
- Liczba rund: 64,
- Używa operacji bitowych (XOR, AND, OR, NOT, rotacje, przesunięcia).



Rysunek 9: Proces deszyfrowania ECDH w aplikacji CryptoLab

14.3 Algorytm SHA-256

Proces haszowania SHA-256 składa się z następujących etapów:

1. Preprocessing (przygotowanie wiadomości):

- Dodanie bitu '1' na końcu wiadomości,
- Dodanie zer dopełniających do osiągnięcia długości $\equiv 448 \pmod{512}$ bitów,
- Dodanie 64-bitowej reprezentacji długości oryginalnej wiadomości.

2. Inicjalizacja wartości początkowych (H):

Ośmiu 32-bitowych stałych (pierwsze 32 bity ułamkowych części pierwiastków kwadratowych pierwszych 8 liczb pierwszych):

$$H_0 = 0x6a09e667, H_1 = 0xbb67ae85, \dots, H_7 = 0x5be0cd19$$

3. Przetwarzanie bloków 512-bitowych:

Dla każdego bloku:

1. Przygotowanie harmonogramu wiadomości W_0, W_1, \dots, W_{63} (pierwsze 16 słów to dane wejściowe, pozostałe obliczane z funkcji σ),
2. 64 rundy kompresji z użyciem funkcji logicznych:

- $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$ – funkcja wyboru,
- $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$ – funkcja większości,
- $\Sigma_0, \Sigma_1, \sigma_0, \sigma_1$ – funkcje rotacji i przesunięć bitowych.

3. Aktualizacja wartości roboczych a, b, c, d, e, f, g, h ,
4. Dodanie wyników do wartości H_0, \dots, H_7 .

4. Konkatenacja wyników:

Połączenie wartości H_0 do H_7 daje 256-bitowy skrót.

14.4 Zastosowania SHA-256

- **Podpisy cyfrowe** – skrót dokumentu jest podpisywany zamiast całego dokumentu,
- **Blockchain i Bitcoin** – mechanizm proof-of-work, identyfikatory transakcji,
- **Certyfikaty SSL/TLS** – weryfikacja autentyczności certyfikatów,
- **Sprawdzanie integralności plików** – porównywanie sum kontrolnych,
- **Przechowywanie haseł** – haszowanie haseł przed zapisem w bazie danych,
- **HMAC** – uwierzytelnianie wiadomości z użyciem klucza tajnego.

14.5 Bezpieczeństwo SHA-256

SHA-256 jest uznawany za bezpieczny algorytm:

- Brak znanych praktycznych ataków kolizyjnych,
- Złożoność czasowa ataku brutalnego: 2^{256} operacji,
- Odporność na ataki urodzinowe: 2^{128} operacji,
- Używany w krytycznych systemach bezpieczeństwa (TLS, IPsec, SSH).

14.6 Implementacja w CryptoLab

Aplikacja zawiera pełną implementację algorytmu SHA-256 bez użycia zewnętrznych bibliotek. Implementacja obejmuje:

- Preprocessing z padding i dodawaniem długości,
- Wszystkie funkcje logiczne (Ch, Maj, Σ, σ),
- 64 rundy kompresji dla każdego bloku,
- Obsługę wiadomości UTF-8,
- Szczegółowe logowanie każdego kroku procesu haszowania.

Uwaga: SHA-256 to funkcja jednokierunkowa, więc **nie ma operacji deszyfrowania**. Próba ”odszyfrowania” zwraca komunikat informujący o jednokierunkowości funkcji.

15 Podpis Elektroniczny (Digital Signature)

15.1 Wprowadzenie do podpisów cyfrowych

Podpis elektroniczny (cyfrowy) to kryptograficzny mechanizm zapewniający:

- **Autentyczność** – potwierdzenie tożsamości nadawcy,
- **Integralność** – gwarancja, że dokument nie został zmieniony,
- **Niezaprzecjalność** – nadawca nie może zaprzeczyć podpisaniu dokumentu.

W odróżnieniu od szyfrowania, które zapewnia *poufność*, podpis cyfrowy zapewnia *autentyczność* i *integralność*.

15.2 Schemat podpisu RSA-SHA256

Aplikacja CryptoLab implementuje schemat podpisu cyfrowego łączący RSA z SHA-256:

Proces podpisywania:

1. Oblicz skrót SHA-256 dokumentu: $h = \text{SHA-256}(\text{dokument})$
2. Konwertuj skrót na liczbę całkowitą (użyj pierwszych 3 znaków hex dla małego RSA)
3. Podpisz skrót kluczem prywatnym: $s = h^d \bmod n$
4. Zwróć: dokument, hash, podpis, klucz publiczny

Proces weryfikacji:

1. Odbierz: dokument, hash oryginalny, podpis, klucz publiczny
2. Oblicz skrót SHA-256 otrzymanego dokumentu: $h' = \text{SHA-256}(\text{dokument})$
3. Porównaj h' z oryginalnym hashem (sprawdzenie integralności)
4. Odszyfruj podpis kluczem publicznym: $h'' = s^e \bmod n$
5. Porównaj h' z h'' – jeśli zgodne, podpis jest ważny

15.3 Dlaczego SHA-256 + RSA?

Połączenie funkcji skrótu z RSA jest kluczowe z kilku powodów:

- **Wydajność** – RSA jest wolny dla dużych danych. Zamiast podpisywać cały dokument, podpisujemy tylko 256-bitowy hash.
- **Bezpieczeństwo** – bezpośrednie podpisywane danych RSA jest podatne na ataki. Haszowanie eliminuje te problemy.
- **Rozmiar** – podpis ma stały rozmiar niezależnie od wielkości dokumentu.
- **Standardy** – RSA-SHA256 to powszechnie uznany standard (PKCS#1, RFC 8017).

15.4 Format podpisu w CryptoLab

Podpis w aplikacji ma format:

$$\text{dokument}|\text{hash}|\text{podpis}|\text{klucz_publiczny}$$

Gdzie:

- **dokument** – oryginalny tekst (do weryfikacji integralności),
- **hash** – skrót SHA-256 dokumentu (64 znaki hex),
- **podpis** – zaszyfrowany hash kluczem prywatnym (hex),
- **klucz_publiczny** – para (n, e) do weryfikacji.

15.5 Parametry RSA dla celów edukacyjnych

Dla demonstracji algorytmu używamy małych liczb pierwszych:

- $p = 61, q = 53$
- $n = p \times q = 3233$
- $\phi(n) = (p - 1)(q - 1) = 3120$
- $e = 17$ (wykładnik publiczny)
- $d = 2753$ (wykładnik prywatny, $e \cdot d \equiv 1 \pmod{\phi(n)}$)

Uwaga: W rzeczywistych zastosowaniach używa się kluczy 2048-4096 bitowych!

15.6 Bezpieczeństwo podpisu cyfrowego

Podpis cyfrowy RSA-SHA256 jest bezpieczny pod warunkiem:

- **Klucz prywatny pozostaje tajny** – tylko właściciel może podpisywać,
- **Silne generowanie kluczy** – losowe, duże liczby pierwsze,
- **Bezpieczna funkcja skrótu** – SHA-256 jest odporna na kolizje,
- **Odpowiedni padding** – w produkcji używa się PSS (Probabilistic Signature Scheme).

15.7 Zastosowania podpisów cyfrowych

- **Certyfikaty SSL/TLS** – weryfikacja autentyczności serwerów WWW,
- **Podpisywanie oprogramowania** – weryfikacja pochodzenia aplikacji,
- **Dokumenty elektroniczne** – prawnie wiążące podpisy (eIDAS, esignature),
- **Transakcje finansowe** – autoryzacja przelewów bankowych,
- **Blockchain** – potwierdzanie transakcji kryptowalut,
- **E-mail** – S/MIME, PGP do podpisywania wiadomości,
- **Aktualizacje systemu** – weryfikacja integralności pakietów.

15.8 Implementacja w CryptoLab

Aplikacja zawiera pełną implementację podpisu elektronicznego:

- Generowanie pary kluczy RSA (edukacyjna wersja 12-bitowa),
- Integracja z SHA-256 do haszowania dokumentów,
- Modular exponentiation dla operacji RSA,
- Rozszerzony algorytm Euklidesa dla odwrotności modularnej,
- Szczegółowe logowanie każdego kroku podpisywania i weryfikacji,
- Wykrywanie modyfikacji dokumentu,
- Walidacja podpisu z komunikatami WAŻNY / NIEWAŻNY.

Demonstracja:

1. Użytkownik wpisuje dokument i kliką "Podpisz dokument"
2. System oblicza hash SHA-256 i podpisuje go kluczem prywatnym
3. Zwraca pełny podpis (dokument—hash—podpis—klucz_publiczny)
4. Do weryfikacji użytkownik wkleja cały podpis i kliką "Weryfikuj podpis"
5. System sprawdza integralność dokumentu i autentyczność podpisu
6. Wyświetla wynik: PODPIS WAŻNY lub PODPIS NIEWAŻNY

16 Wybrane fragmenty kodu

16.1 Klasa bazowa algorytmu

Listing 2: Klasa abstrakcyjna CryptographicAlgorithm

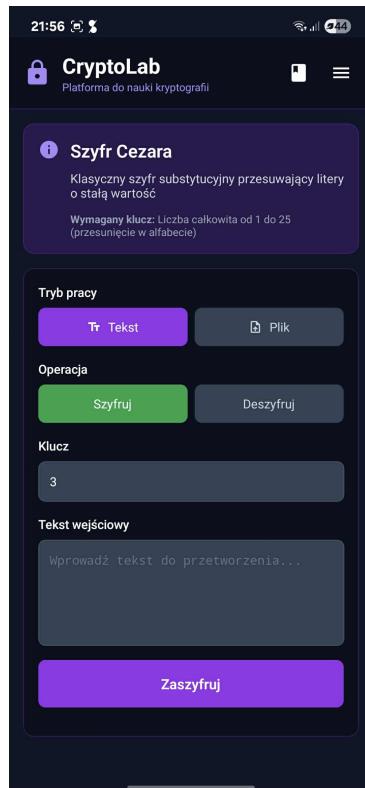
```
export default class CryptographicAlgorithm {
    name: string;
    description: string;
    category: string;

    encrypt(plaintext: string, key: string): string {
        throw new Error('Metoda encrypt() musi być zaimplementowana');
    }

    decrypt(ciphertext: string, key: string): string {
        throw new Error('Metoda decrypt() musi być zaimplementowana');
    }

    validateKey(key: string): { valid: boolean; error?: string } {
        throw new Error('Metoda validateKey() musi być zaimplementowana');
    }

    getKeyRequirements(): string {
        throw new Error('Metoda getKeyRequirements() musi być zaimplementowana');
    }
}
```



Rysunek 10: Ekran główny aplikacji CryptoLab



Rysunek 11: Lista z możliwością wyboru algorytmu

Rysunek ?? przedstawia ekran główny aplikacji CryptoLab Mobile, a rysunek ?? pokazuje listę dostępnych algorytmów kryptograficznych.

16.2 Implementacja szyfru Cezara

Listing 3: Szczegóły implementacji CaesarCipher

```
export default class CaesarCipher extends CryptographicAlgorithm {
  constructor() {
    super(
      'Szyfr Cezara',
      'Prosty szyfr substytucyjny z przesuni ciem',
      'Szyfry klasyczne'
    );
  }

  validateKey(key: string): { valid: boolean; error?: string } {
    const numKey = parseInt(key, 10);
    if (isNaN(numKey) || numKey < 1 || numKey > 25) {
      return {
        valid: false,
        error: 'Klucz musi by  liczb od 1 do 25'
      };
    }
    return { valid: true };
  }

  encrypt(plaintext: string, key: string): string {
    return this._process(plaintext, parseInt(key, 10));
  }
}
```

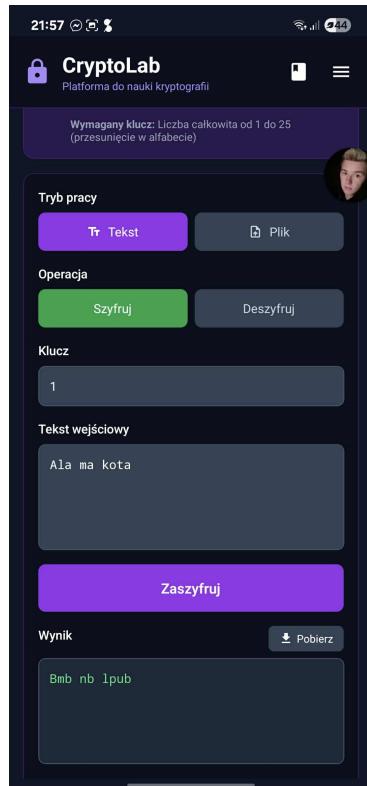
```

}

decrypt(ciphertext: string, key: string): string {
  const shift = 26 - (parseInt(key, 10) % 26);
  return this._process(ciphertext, shift);
}

private _process(text: string, shift: number): string {
  return text.split('').map(char => {
    if (/^[A-Za-z]$/.test(char)) {
      const base = char === char.toUpperCase() ? 65 : 97;
      return String.fromCharCode(
        (char.charCodeAt(0) - base + shift) % 26 + base
      );
    }
    return char;
  }).join('');
}
}

```



Rysunek 12: Test szyfru Cezara

Rysunek ?? przedstawia przykładowy test szyfru Cezara w aplikacji CryptoLab Mobile.

16.3 Implementacja szyfru Vigenère'a

Listing 4: Fragmenty klasy VigenereCipher

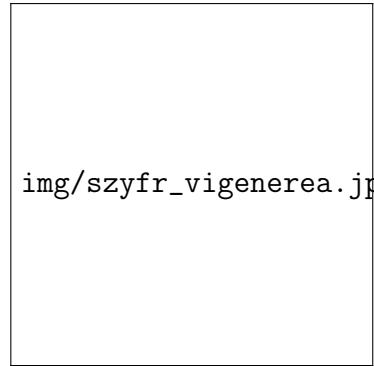
```
export default class VigenereCipher extends CryptographicAlgorithm {
    validateKey(key: string): { valid: boolean; error?: string } {
        if (!key || key.trim().length === 0) {
            return { valid: false, error: 'Klucz nie może być pusty' };
        }
        const hasOnlyLetters = /^[a-zA-Z]+$/;
        if (!hasOnlyLetters) {
            return { valid: false, error: 'Klucz może zawierać tylko litery' };
        }
        return { valid: true };
    }

    private _process(text: string, key: string, encrypt: boolean): string {
        let result = '';
        let keyIndex = 0;
        const normalizedKey = key.toUpperCase();

        for (let i = 0; i < text.length; i++) {
            const char = text[i];
            if (/[^A-Za-z]/.test(char)) {
                const base = char === char.toUpperCase() ? 65 : 97;
                const keyCode = normalizedKey.charCodeAt(keyIndex % normalizedKey.length) - 65;

                const resultCode = encrypt
                    ? (textCode + keyCode) % 26
                    : (textCode - keyCode + 26) % 26;

                result += String.fromCharCode(resultCode + base);
                keyIndex++;
            } else {
                result += char;
            }
        }
        return result;
    }
}
```



Rysunek 13: Ekran szyfru Vigenere'a

Rysunek ?? przedstawia interfejs użytkownika szyfru Vigenère'a w aplikacji CryptoLab Mobile.

16.4 Implementacja szyfru z kluczem bieżącym

Listing 5: Fragmenty klasy RunningKeyCipher

```
export default class RunningKeyCipher extends CryptographicAlgorithm {
    constructor() {
        super(
            'Szyfr z kluczem bieżącym',
            'Szyfr podobny do Vigenere'a, ale ułatwiający zmianę klucza o długosci tekstu',
            'Szyfry klasyczne'
        );
    }

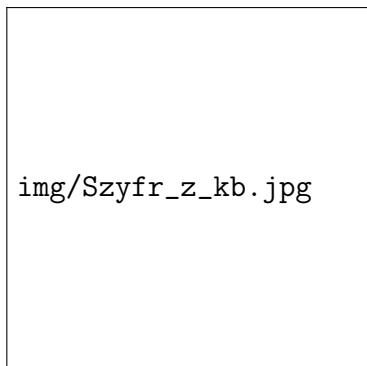
    validateKey(key: string): { valid: boolean; error?: string } {
        if (!key || key.trim().length === 0) {
            return { valid: false, error: 'Klucz nie może być pusty' };
        }

        // Sprawdza, czy klucz zawiera tylko litery
        const hasOnlyLetters = /^[a-zA-Z\s]+$/;
        if (!hasOnlyLetters) {
            return { valid: false, error: 'Klucz może zawierać tylko litery i spacje (A-Z, a-z)' };
        }

        // Policz tylko litery w kluczu
        const keyLettersCount = key.replace(/[^a-zA-Z]/g, '').length;
        if (keyLettersCount < 5) {
            return {
                valid: false,
                error: 'Klucz musi zawierać co najmniej 5 liter (może zawierać spacje)'
            };
        }

        return { valid: true };
    }
}
```

```
getKeyRequirements(): string {
    return 'Tekst (np. fragmentksi ki) - użyto generatora lorem
        ipsum do stworzenia klucza';
}
```



Rysunek 14: Ekran szyfru z kluczem bieżącym

Rysunek ?? przedstawia interfejs użytkownika szyfru z kluczem bieżącym w aplikacji CryptoLab Mobile.

16.5 Implementacja szyfru AES

Listing 6: Fragmenty klasy AESCipher

```
export default class AESCipher extends CryptographicAlgorithm {
    private mode: AESMode;
    public static readonly DEFAULT_KEY = '2b7e151628aed2a6abf7158809cf4f3c
    ';

    constructor() {
        super(
            'AES (Advanced Encryption Standard)',
            'Symetryczny szyfr blokowy z możliwością wyboru trybu (ECB, CBC
            , CTR)',
            'Szyfry symetryczne'
        );
        this.mode = 'ECB'; // Domyslny tryb
    }

    // Ustawia tryb pracy AES
    setMode(mode: AESMode): void {
        this.mode = mode;
    }

    validateKey(key: string): { valid: boolean; error?: string } {
        if (!key || key.trim().length === 0) {
            return { valid: false, error: 'Klucz nie może być pusty' };
        }

        // Sprawdza czy klucz jest w formacie hex
        const hexPattern = /^[0-9a-fA-F]+$/;
        if (!hexPattern.test(key)) {
            return {
                valid: false,
                error: 'Klucz musi być ciągiem znaków szesnastkowych (0-9, A-
                    F)'
            };
        }
    }

    // Klucz musi mieć dugość 32, 48 lub 64 znaków w hex
    if (key.length !== 32 && key.length !== 48 && key.length !== 64) {
        return {
            valid: false,
            error: 'Klucz musi mieć długość 32 (AES-128), 48 (AES-192)
                lub 64 (AES-256)'
        };
    }

    return { valid: true };
}

getKeyRequirements(): string {
    return `Tryb: ${this.mode} | Klucz hex: 32 znaki (AES-128),
        48 (AES-192) lub 64 (AES-256)';
}

// Główne metody szyfrowania wykorzystujące wybrany tryb
encrypt(plaintext: string, key: string): string {
```

```

    if (this.mode === 'ECB') {
      return this.encryptECB(plaintext, key);
    } else if (this.mode === 'CBC') {
      return this.encryptCBC(plaintext, key);
    } else if (this.mode === 'CTR') {
      return this.encryptCTR(plaintext, key);
    }
    throw new Error(`Tryb ${this.mode} nie jest obsugiwany`);
  }
}

```

16.6 Implementacja szyfru RSA

Listing 7: Fragmenty klasy RSACipher

```

export default class RSACipher extends CryptographicAlgorithm {
  private keyPair: RSAKeyPair | null = null;

  constructor() {
    super(
      'RSA (Rivest-Shamir-Adleman)',
      'Asymetryczny algorytm kryptograficzny z kluczem publicznym i
       prywatnym',
      'Kryptografia asymetryczna'
    );
  }

  // Generuje parę kluczy RSA
  generateKeyPair(bitSize: number = 512): RSAKeyPair {
    const min = bitSize === 512 ? 100 : 50;
    const max = bitSize === 512 ? 300 : 100;

    // Generuj dwie różne liczby pierwsze
    const p = generatePrime(min, max);
    let q = generatePrime(min, max);
    while (q === p) {
      q = generatePrime(min, max);
    }

    const n = p * q;    // Moduł
    const phi = (p - 1) * (q - 1);  // Funkcja Eulera

    // Wybierz e (wykładnik publiczny)
    let e = 65537;
    if (e >= phi) e = 17;
    while (gcd(e, phi) !== 1) {
      e++;
    }

    // Oblicz d (wykładnik prywatny)
    const d = modInverse(e, phi);

    this.keyPair = {
      publicKey: { e, n },
      privateKey: { d, n }
    };
  }
}

```

```

        return this.keyPair;
    }

    encrypt(plaintext: string, key: string): string {
        const [e, n] = key.split(',').map(p => parseInt(p.trim(), 10));

        const encrypted: number[] = [];
        for (let i = 0; i < plaintext.length; i++) {
            const charCode = plaintext.charCodeAt(i);

            // Szyfrowanie: c = m^e mod n
            const encryptedChar = Number(
                modPow(BigInt(charCode), BigInt(e), BigInt(n))
            );
            encrypted.push(encryptedChar);
        }

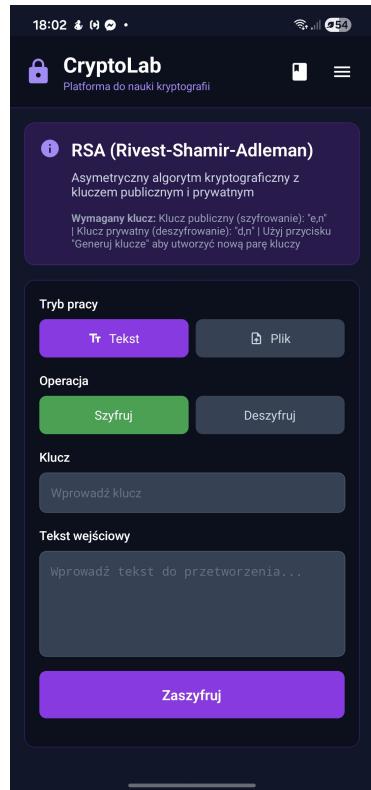
        return encrypted.join(' ');
    }

    decrypt(ciphertext: string, key: string): string {
        const [d, n] = key.split(',').map(p => parseInt(p.trim(), 10));
        const encryptedNumbers = ciphertext.trim().split(/\s+/)
            .map(s => parseInt(s, 10));

        const decrypted: string[] = [];
        for (const encryptedChar of encryptedNumbers) {
            // Deszyfrowanie: m = c^d mod n
            const decryptedChar = Number(
                modPow(BigInt(encryptedChar), BigInt(d), BigInt(n))
            );
            decrypted.push(String.fromCharCode(decryptedChar));
        }

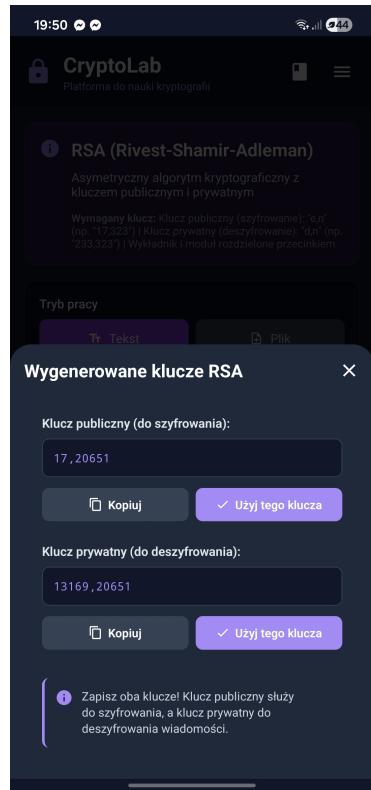
        return decrypted.join('');
    }
}

```



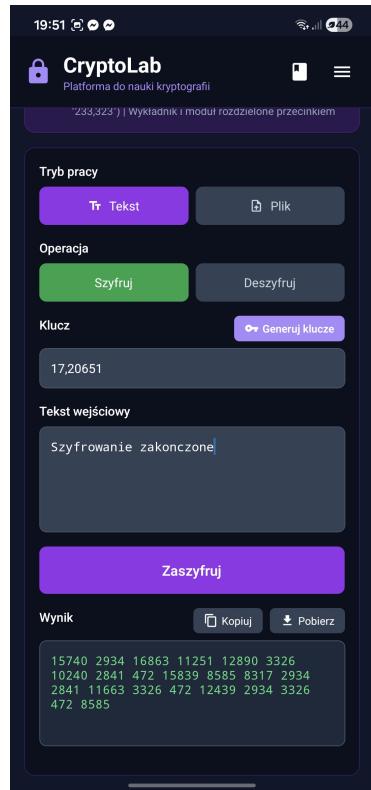
Rysunek 15: Ekran szyfru RSA

Rysunek ?? przedstawia interfejs użytkownika szyfru RSA w aplikacji CryptoLab Mobile.



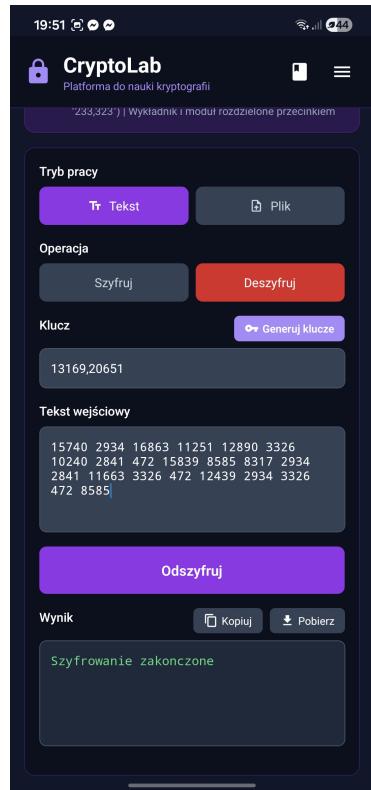
Rysunek 16: Generowanie kluczy RSA

Rysunek ?? przedstawia ekran generowania kluczy RSA w aplikacji CryptoLab Mobile.



Rysunek 17: Test szyfru RSA

Rysunek ?? przedstawia przykładowy test szyfru RSA w aplikacji CryptoLab Mobile.



Rysunek 18: Deszyfrowanie w szyfrze RSA

Rysunek ?? przedstawia ekran deszyfrowania w szyfrze RSA w aplikacji CryptoLab Mobile.

16.7 Implementacja algorytmu ElGamal

Listing 8: Fragmenty klasy ElGamalCipher

```
export default class ElGamalCipher extends CryptographicAlgorithm {
    // Generowanie kluczy
    generateKeyPair(): ElGamalKeyPair {
        // 1. Wybierz duza liczbe pierwsza p
        const p = generatePrime(300, 1000);

        // 2. Znajdz generator g
        const g = findPrimitiveRoot(p);

        // 3. Wybierz losowy klucz prywatny x
        const x = Math.floor(Math.random() * (p - 2)) + 2;

        // 4. Oblicz klucz publiczny y = g^x mod p
        const y = modPow(BigInt(g), BigInt(x), BigInt(p));

        return {
            publicKey: { p: p.toString(), g: g.toString(), y: y.toString() },
            privateKey: { x: x.toString(), p: p.toString() }
        };
    }

    encrypt(plaintext: string, key: string): string {
        const [p, g, y] = key.split(',').map(n => BigInt(n));
        const encryptedPairs: string[] = [];

        for (let i = 0; i < plaintext.length; i++) {
            const m = BigInt(plaintext.charCodeAt(i));

            // 1. Wybierz losowe k (klucz efemeryczny)
            let k: bigint;
            do {
                k = BigInt(Math.floor(Math.random() * (Number(p) - 2)) + 1);
            } while (gcd(k, p - 1n) !== 1n);

            // 2. Oblicz a = g^k mod p
            const a = modPow(g, k, p);

            // 3. Oblicz b = (y^k * m) mod p
            const b = (modPow(y, k, p) * m) % p;

            encryptedPairs.push(` ${a}: ${b} `);
        }
        return encryptedPairs.join(' ');
    }
}
```

16.8 Implementacja algorytmu ECDH

Listing 9: Fragmenty klasy ECDHAlgorithm

```
export default class ECDHAlgorithm extends CryptographicAlgorithm {
    // Dodawanie punktow na krzywej eliptycznej
    private addPoints(P: Point, Q: Point): Point {
```

```

    if (P.isInfinity) return Q;
    if (Q.isInfinity) return P;

    let m: bigint;
    if (P.x === Q.x && P.y === Q.y) {
        // Podwojenie punktu: m = (3x^2 + a) / (2y)
        const num = (3n * P.x * P.x + CURVE.a) % CURVE.p;
        const den = (2n * P.y) % CURVE.p;
        m = (num * this.modInverse(den, CURVE.p)) % CURVE.p;
    } else {
        // Dodawanie: m = (y2 - y1) / (x2 - x1)
        const num = (Q.y - P.y) % CURVE.p;
        const den = (Q.x - P.x) % CURVE.p;
        m = (num * this.modInverse(den, CURVE.p)) % CURVE.p;
    }

    // Obliczanie x3, y3...
    // ...
    return { x: x3, y: y3, isInfinity: false };
}

encrypt(plaintext: string, key: string): string {
    // key to klucz publiczny odbiorcy (Q)
    const [qx, qy] = key.split(',').map(c => BigInt(c));
    const Q: Point = { x: qx, y: qy, isInfinity: false };

    // 1. Wygeneruj tymczasowa pare kluczy (k, R)
    const k = BigInt(Math.floor(Math.random() * (Number(CURVE.p) - 2)) +
        1);
    const R = this.multiplyPoint(k, CURVE.G); // R = k * G

    // 2. Oblicz wspolny sekret S = k * Q
    const S = this.multiplyPoint(k, Q);

    // 3. Uzyj wspolrzednej x punktu S jako klucza symetrycznego
    const symmetricKey = Number(S.x);

    // 4. Szyfruj wiadomosc (XOR)
    // ...

    return `${R.x},${R.y}|${ciphertextBody}`;
}
}

```

16.9 Implementacja funkcji SHA-256

Listing 10: Fragmenty klasy SHA256Hash

```

export default class SHA256Hash extends CryptographicAlgorithm {
    // Stale K (pierwsze 32 bity ulamkowych czesci
    // szescianowych pierwiastkow pierwszych 64 liczb pierwszych)
    private K: number[] = [
        0x428a2f98, 0x71374491, 0xb5c0fbef, 0xe9b5dba5,
        // ... wiecej wartosci ...
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
        0x90beffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
    ];
}

```

```

// Wartosci poczatkowe H (pierwiastki kwadratowe
// pierwszych 8 liczb pierwszych)
private H: number[] = [
  0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
  0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
];

encrypt(plaintext: string, key: string = ''): string {
  this.logStep('SHA-256 Hash Process Started', plaintext);

  const bytes = this.stringToBytes(plaintext);
  const hash = this.sha256(bytes);
  const hexHash = this.bytesToHex(hash);

  this.logStep('SHA-256 Hash Generated', plaintext, hexHash);
  return hexHash;
}

// Glowny algorytm SHA-256
private sha256(message: number[]): number[] {
  const preprocessed = this.preprocess(message);
  const H = [...this.H];

  // Przetwarzanie kazdego bloku 512-bitowego
  for (let i = 0; i < preprocessed.length; i += 16) {
    const block = preprocessed.slice(i, i + 16);
    const W = this.prepareSchedule(block);
    let [A, B, C, D, E, F, G, H_temp] = [...H];

    // 64 rundy
    for (let t = 0; t < 64; t++) {
      const T1 = H_temp + this.Sigma1(E) +
        this.Ch(E, F, G) + this.K[t] + W[t];
      const T2 = this.Sigma0(A) + this.Maj(A, B, C);

      H_temp = G; G = F; F = E;
      E = (D + T1) >>> 0;
      D = C; C = B; B = A;
      A = (T1 + T2) >>> 0;
    }

    // Aktualizacja wartosci H
    H[0] = (H[0] + A) >>> 0;
    H[1] = (H[1] + B) >>> 0;
    // ... pozostałe ...
  }
  return this.hashToBytes(H);
}

// Funkcje logiczne SHA-256
private Ch(x: number, y: number, z: number): number {
  return (x & y) ^ (~x & z);
}

private Maj(x: number, y: number, z: number): number {
  return (x & y) ^ (x & z) ^ (y & z);
}

```

```

private Sigma0(x: number): number {
    return this.rightRotate(x, 2) ^
        this.rightRotate(x, 13) ^
        this.rightRotate(x, 22);
}

private Sigma1(x: number): number {
    return this.rightRotate(x, 6) ^
        this.rightRotate(x, 11) ^
        this.rightRotate(x, 25);
}
}

```

16.10 Implementacja podpisu elektronicznego

Listing 11: Fragmenty klasy DigitalSignature

```

export default class DigitalSignature extends CryptographicAlgorithm {
    private sha256: SHA256Hash;
    private keyPair: KeyPair | null = null;

    // Parametry RSA dla celow edukacyjnych
    private P: bigint = 61n;
    private Q: bigint = 53n;
    private N: bigint = this.P * this.Q; // 3233
    private PHI: bigint = (this.P - 1n) * (this.Q - 1n); // 3120
    private E: bigint = 17n; // Wyk adnik publiczny

    constructor() {
        super(
            'Podpis Elektroniczny',
            'Podpisywanie i weryfikacja dokumentow za pomoca RSA-SHA256',
            'Kryptografia asymetryczna'
        );
        this.sha256 = new SHA256Hash();
        this.sha256.setLogging(false); // Wylacz logowanie SHA256
        this.generateKeyPair();
    }

    // Generowanie pary kluczy RSA
    private generateKeyPair(): void {
        const d = this.modularInverse(this.E, this.PHI);
        this.keyPair = {
            publicKey: { n: this.N, e: this.E },
            privateKey: { n: this.N, d: d }
        };
    }

    // Podpisywanie dokumentu
    encrypt(plaintext: string, key: string = ''): string {
        // Krok 1: Oblicz hash SHA-256
        const documentHash = this.sha256.encrypt(plaintext);

        // Krok 2: Konwertuj hash na liczbe (pierwsze 3 znaki)
        const hashNumber = this.hashToNumber(documentHash);
    }
}

```

```

// Krok 3: Podpisz hash kluczem prywatnym
// signature = hash^d mod n
const signature = this.modularExponentiation(
    hashNumber,
    this.keyPair.privateKey.d,
    this.keyPair.privateKey.n
);

const signatureHex = signature.toString(16);

// Krok 4: Zwroc format: dokument|hash|podpis|klucz_publiczny
const publicKeyStr =
    '${this.keyPair.publicKey.n},${this.keyPair.publicKey.e}';
return `${plaintext}|${documentHash}|${signatureHex}|${publicKeyStr}
    `;
}

// Weryfikacja podpisu
decrypt(ciphertext: string, key: string = ''): string {
    // Parsowanie: dokument|hash|podpis|klucz_publiczny
    const [originalDoc, originalHash, signatureHex, publicKeyStr]
        = ciphertext.split('|');

    // Krok 1: Oblicz hash otrzymanego dokumentu
    const currentHash = this.sha256.encrypt(originalDoc);

    // Krok 2: Sprawdz integralnosc
    if (currentHash !== originalHash) {
        return 'PODPIS NIEWAZNY - Dokument zostal zmieniony!';
    }

    // Krok 3: Parsuj klucz publiczny
    const [nStr, eStr] = publicKeyStr.split(',');
    const publicKey = { n: BigInt(nStr), e: BigInt(eStr) };

    // Krok 4: Odszyfruj podpis kluczem publicznym
    const hashNumber = this.hashToNumber(currentHash);
    const signatureNumber = BigInt('0x' + signatureHex);
    const verifiedHash = this.modularExponentiation(
        signatureNumber,
        publicKey.e,
        publicKey.n
    );

    // Krok 5: Porownaj hashe
    if (hashNumber === verifiedHash) {
        return 'PODPIS WAZNY - Dokument autentyczny!';
    } else {
        return 'PODPIS NIEWAZNY - Podpis falszywy!';
    }
}

// Modulowe potegowanie (binary exponentiation)
private modularExponentiation(
    base: bigint,
    exp: bigint,
    modulus: bigint
): bigint {

```

```

if (modulus === 1n) return 0n;
let result = 1n;
base = base % modulus;

while (exp > 0n) {
    if (exp % 2n === 1n) {
        result = (result * base) % modulus;
    }
    exp = exp >> 1n;
    base = (base * base) % modulus;
}
return result;
}

// Odwrotnosc modulowa (Extended Euclidean Algorithm)
private modularInverse(a: bigint, m: bigint): bigint {
    let [old_r, r] = [a, m];
    let [old_s, s] = [1n, 0n];

    while (r !== 0n) {
        const quotient = old_r / r;
        [old_r, r] = [r, old_r - quotient * r];
        [old_s, s] = [s, old_s - quotient * s];
    }
    return old_s < 0n ? old_s + m : old_s;
}
}

```

17 Podsumowanie

17.1 Szyfr Cezara

Szyfr Cezara należy do najstarszych i najprostszych technik szyfrowania. Jego główna idea polega na przesuwaniu liter alfabetu o ustaloną liczbę pozycji. Mimo że w praktyce jest to jedynie przykład historyczny, implementacja szyfru pozwala lepiej zrozumieć podstawowe mechanizmy kryptografii, takie jak klucz, szyfrowanie i deszyfrowanie.

Zalety:

- bardzo prosta implementacja,
- szybkie działanie,
- dobre ćwiczenie dydaktyczne.

Wady:

- niska odporność na ataki kryptograficzne,
- atak brute-force łatwe przełamuje szyfr w sekundach,
- podatny na analizę częstotliwości.

17.2 Szyfr Vigenère'a

Szyfr Vigenère'a to znacznie bardziej zaawansowany szyfr polialfabetyczny. Przez wieki uważany był za niezniszczalny, ale ostatecznie został przełamany dzięki analizie częstotliwości długości okresu.

Zalety:

- znacznie bardziej bezpieczny niż szyfr Cezara,
- odporne na prostą analizę częstotliwości,
- wykorzystuje koncepcję słowa-klucza, co jest intuicyjne.

Wady:

- niska odporność na ataki kryptograficzne (możliwy atak siłowy poprzez sprawdzenie wszystkich przesunięć),
- brak zastosowania we współczesnych systemach bezpieczeństwa,
- szyfr działa jedynie na ograniczonym zbiorze znaków (najczęściej alfabet łaciński).

17.3 Szyfr z kluczem bieżącym

Szyfr z kluczem bieżącym to krok w kierunku szyfrowania jednorazowego.

Zalety:

- gdy klucz jest losowy i używany raz – teoretycznie nie do złamania,
- koncepcja zbliża się do rzeczywistego bezpieczeństwa informacyjnego,
- edukacyjnie pokazuje znaczenie losowości klucza.

Wady:

- wymaga przechowywania bardzo długich kluczy,
- wymaga absolutnej losowości i jednorazowego użycia,
- niepraktyczne w większości rzeczywistych zastosowań.

17.4 Szyfr AES

AES (Advanced Encryption Standard) to nowoczesny szyfr symetryczny, który stanowi podstawę współczesnej kriptografii. W przeciwieństwie do szyfrów klasycznych, AES jest używany w realnych systemach bezpieczeństwa na całym świecie.

Zalety:

- wysoki poziom bezpieczeństwa – odporny na wszystkie znane praktyczne ataki,
- elastyczność – obsługa trzech długości kluczy (128, 192, 256 bitów),
- różne tryby pracy (ECB, CBC, CTR) dostosowane do różnych zastosowań,
- szybkie działanie przy zachowaniu bezpieczeństwa,
- szeroko stosowany i przetestowany w praktyce,
- standaryzowany przez NIST i akceptowany globalnie.

Wady:

- znacznie bardziej złożona implementacja niż szyfry klasyczne,
- wymaga zrozumienia trybów pracy i ich właściwości,
- tryb ECB jest niebezpieczny i nie powinien być stosowany w praktyce,
- wymaga bezpiecznego zarządzania kluczami i wektorami inicjalizującymi (IV),
- jako szyfr symetryczny, wymaga bezpiecznego przekazania klucza obu stron komunikacji.

Zastosowanie edukacyjne:

- pokazuje różnicę między kriptografią klasyczną a nowoczesną,
- wprowadza pojęcia: tryby pracy, padding, wektor inicjalizujący (IV),
- demonstruje znaczenie wyboru odpowiedniego trybu pracy,
- ilustruje jak działa rzeczywiste szyfrowanie stosowane w praktyce.

17.5 Szyfr RSA

RSA to przełomowy algorytm kriptografii asymetrycznej, który rozwiązał fundamentalny problem bezpiecznej wymiany kluczy i wprowadził koncepcję kluczy publicznych.

Zalety:

- rozwiązuje problem wymiany kluczy – nie wymaga bezpiecznego kanału,
- umożliwia podpisy cyfrowe i uwierzytelnianie,
- bezpieczny przy odpowiednio dużych kluczach (2048+ bitów),
- szeroko stosowany i przetestowany w praktyce,
- fundamentalna technologia dla PKI (Public Key Infrastructure),
- umożliwia szyfrowanie hybrydowe w połączeniu z AES.

Wady:

- znacznie wolniejszy niż algorytmy symetryczne (100-1000x),
- wymaga dużych kluczy (2048-4096 bitów) dla bezpieczeństwa,
- zagrożenie ze strony komputerów kwantowych,
- złożona implementacja – łatwo popełnić błędy bezpieczeństwa,
- nie nadaje się do szyfrowania dużych ilości danych.

Zastosowanie edukacyjne:

- wprowadza fundamentalną różnicę między kriptografią symetryczną i asymetryczną,
- pokazuje matematyczne podstawy bezpieczeństwa (teoria liczb),
- ilustruje koncepcję klucza publicznego i prywatnego,
- demonstruje praktyczne zastosowania: wymiana kluczy, podpisy cyfrowe,
- pozwala zrozumieć jak działa HTTPS, SSH i inne protokoły bezpieczeństwa.

RSA w praktyce: W rzeczywistych systemach RSA rzadko jest używany do bezpośredniego szyfrowania danych. Zamiast tego stosuje się **szyfrowanie hybrydowe**:

1. Generowany jest losowy klucz AES (symetryczny),
2. Dane szyfrowane są szybkim algorytmem AES,
3. Klucz AES szyfrowany jest wolnym, ale bezpiecznym RSA,
4. Przesyłany jest zaszyfrowany klucz AES + zaszyfrowane dane.

To połączenie zapewnia zarówno bezpieczeństwo RSA, jak i szybkość AES.

17.6 Algorytm ElGamal

ElGamal to alternatywa dla RSA, oparta na innym problemie matematycznym (logarytm dyskretny).

Zalety:

- bezpieczeństwo oparte na dobrze zbadanym problemie matematycznym,
- niedeterministyczność szyfrowania (ten sam tekst szyfrowany jest inaczej za każdym razem), co utrudnia kryptoanalizę.

Wady:

- szyfrogram jest dwukrotnie dłuższy od wiadomości jawnej (para liczb dla każdego bloku),
- wolniejsze działanie niż RSA (więcej potęgowania).

17.7 Protokół ECDH

Kryptografia krzywych eliptycznych (ECC) to przyszłość bezpiecznej komunikacji.

Zalety:

- bardzo wysoki poziom bezpieczeństwa przy krótkich kluczach (256-bit ECC \approx 3072-bit RSA),
- szybsze obliczenia i mniejsze zużycie energii (ważne w urządzeniach mobilnych),
- mniejsze wymagania pamięciowe.

Wady:

- skomplikowana matematyka i trudniejsza implementacja,
- wrażliwość na słabe generatory liczb losowych.

17.8 Funkcja SHA-256

SHA-256 jest kryptograficzną funkcją skrótu o fundamentalnym znaczeniu dla bezpieczeństwa.

Zalety:

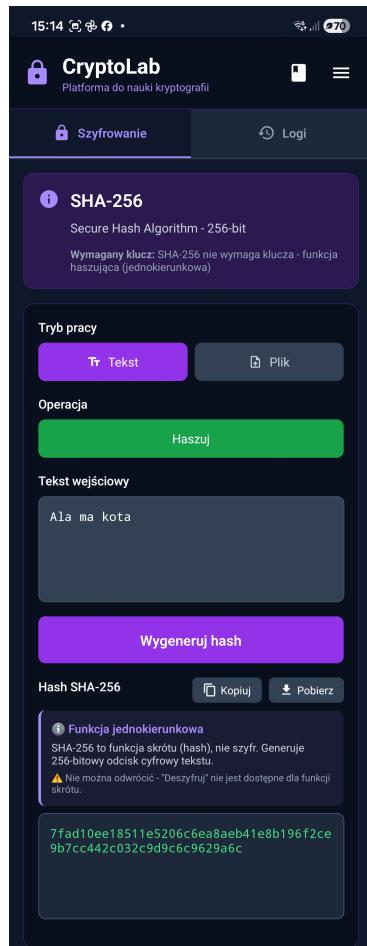
- **jednokierunkowość** – niemożliwe odtworzenie oryginalnej wiadomości,
- **deterministyczność** – ten sam wejście zawsze daje ten sam wynik,
- **efekt lawiny** – minimalna zmiana wejścia drastycznie zmienia skrót,
- **odporność na kolizje** – praktycznie niemożliwe znalezienie dwóch wiadomości o tym samym skrócie,
- **szybkość** – wydajne obliczenia na nowoczesnym sprzęcie,
- **wszechstronność** – używana w blockchain, podpisach cyfrowych, TLS, weryfikacji integralności.

Wady i ograniczenia:

- nie nadaje się do haszowania haseł bez dodatkowych mechanizmów (salt, iteracje),
- brak mechanizmu weryfikacji autentyczności (do tego służy HMAC),
- teoretycznie podatna na ataki kwantowe (w odległej przyszłości).

Zastosowania w praktyce:

- **Bitcoin i blockchain** – proof-of-work, identyfikacja bloków i transakcji,
- **Podpisy cyfrowe** – haszowanie dokumentów przed podpisaniem,
- **Certyfikaty SSL/TLS** – weryfikacja integralności i autentyczności,
- **Git** – identyfikacja commitów i obiektów (choć Git używa SHA-1),
- **Sumy kontrolne** – weryfikacja pobranych plików.



Rysunek 19: Ekran funkcji SHA-256

Rysunek ?? przedstawia interfejs użytkownika funkcji SHA-256 w aplikacji CryptoLab Mobile.

17.9 Podpis Elektroniczny

Podpis elektroniczny to kryptograficzny mechanizm łączący SHA-256 z RSA do autentykacji dokumentów.

Zalety:

- **Autentyczność** – potwierdza tożsamość nadawcy (tylko właściciel klucza prywatnego może podpisać),
- **Integralność** – wykrywa każdą modyfikację dokumentu,
- **Niezaprzecjalność** – nadawca nie może zaprzeczyć podpisaniu (non-repudiation),
- **Efektywność** – podpisywany jest tylko hash (256 bitów), nie cały dokument,
- **Uniwersalność** – działa dla dokumentów dowolnej wielkości,
- **Zgodność ze standardami** – RSA-SHA256 to powszechnie uznany standard (PKCS#1).

Wady i ograniczenia:

- **Zarządzanie kluczami** – wymaga bezpiecznego przechowywania klucza prywatnego,
- **Infrastruktura PKI** – w praktyce potrzebne są certyfikaty i zaufane centra certyfikacji,
- **Podatność kwantowa** – RSA może być złamany przez komputery kwantowe (w przyszłości),
- **Rozmiar klucza** – duże klucze (2048-4096 bitów) wymagają więcej pamięci.

Proces w CryptoLab:

1. **Podpiswanie:** Użytkownik wpisuje dokument → System oblicza SHA-256 → Podpisuje kluczem prywatnym → Zwraca: dokument—hash—podpis—klucz_publiczny
2. **Weryfikacja:** Użytkownik wkleja podpis → System oblicza hash dokumentu → Sprawdza integralność → Weryfikuje podpis kluczem publicznym → Wyświetla: WAŻNY lub NIEWAŻNY

Zastosowania praktyczne:

- **SSL/TLS** – certyfikaty serwerów WWW,
- **Podpisy kwalifikowane** – prawnie wiążące dokumenty (eIDAS),
- **Kod aplikacji** – weryfikacja pochodzenia oprogramowania,
- **Blockchain** – autoryzacja transakcji Bitcoin, Ethereum,
- **E-mail** – S/MIME, PGP dla bezpiecznej poczty,
- **Aktualizacje systemu** – weryfikacja integralności pakietów apt, yum.

Edukacyjne aspekty implementacji:

- Demonstracja związku między SHA-256 a RSA,
- Zrozumienie różnicy między szyfrowaniem a podpisywaniem,
- Wizualizacja procesu weryfikacji integralności,
- Obserwacja efektu modyfikacji dokumentu (podpis staje się nieważny),
- Praktyczne zastosowanie modular exponentiation i Extended Euclidean Algorithm.

18 System logowania operacji

Aplikacja CryptoLab Mobile zawiera zaawansowany system logowania, który rejestruje wszystkie operacje kryptograficzne wykonywane przez użytkownika. System ten służy celom edukacyjnym i analitycznym, umożliwiając śledzenie historii operacji oraz analizę kroków algorytmów.

18.1 Architektura systemu logowania

System logowania składa się z trzech głównych komponentów:

LogManager.ts Singleton zarządzający całym systemem logowania. Odpowiada za:

- rozpoczęwanie i kończenie operacji kryptograficznych,
- rejestrowanie kroków pośrednich w algorytmach,
- zapisywanie logów w pamięci trwałej (AsyncStorage),
- powiadamianie komponentów UI o zmianach w logach,
- zarządzanie liczbą przechowywanych logów (maksymalnie 100).

LogTypes.ts Definicje typów TypeScript dla systemu logowania:

- **LogStep** – pojedynczy krok w algorytmie,
- **CryptoLogEntry** – kompletny wpis logu operacji,
- **LogFilter** – filtry wyszukiwania logów,
- **LogStats** – statystyki użycia algorytmów.

LogsViewer.tsx Komponent React Native wyświetlający historię operacji. Funkcjonalności:

- wyświetlanie listy wszystkich operacji z timestampami,
- filtrowanie logów (wszystkie/szyfrowanie/deszyfrowanie),
- podgląd szczegółów operacji krok po kroku,
- eksport wyników do schowka,
- usuwanie pojedynczych logów lub czyszczenie całej historii,
- wyświetlanie statystyk (liczba operacji, najczęściej używany algorytm).

18.2 Rejestrowane informacje

Każdy wpis logu (CryptoLogEntry) zawiera:

- **Metadane:** ID, timestamp, nazwa algorytmu, typ operacji (encrypt/decrypt)
- **Dane operacji:** tekst wejściowy, tekst wyjściowy, klucz (maskowany dla bezpieczeństwa)
- **Parametry:** tryb pracy (dla AES), długość klucza
- **Status:** sukces/błąd, komunikat błędu (jeśli wystąpił)
- **Wydajność:** czas wykonania operacji w milisekundach
- **Kroki algorytmu:** szczegółowy przebieg operacji z danymi pośrednimi

18.3 Kroki algorytmu

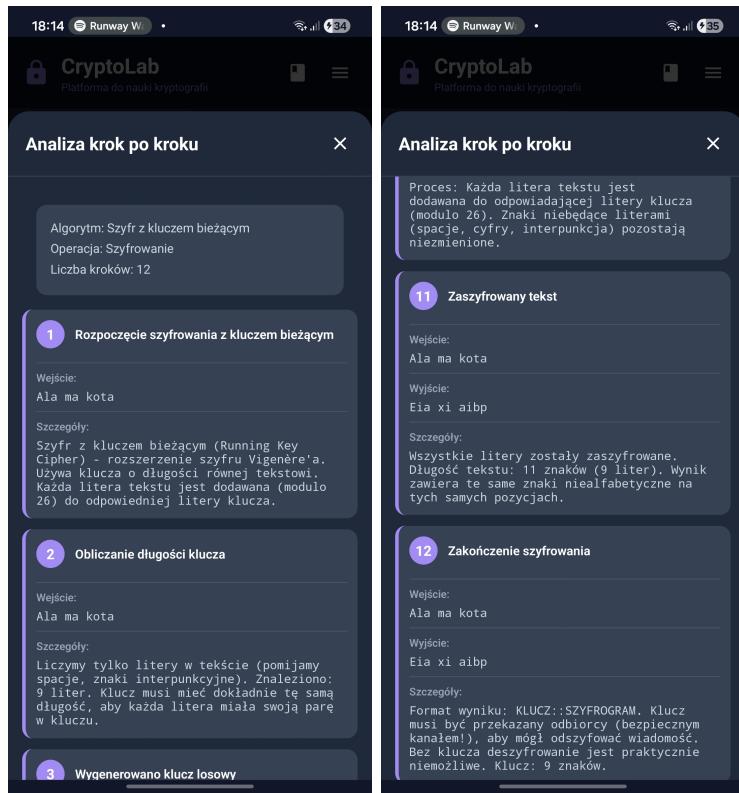
System loguje szczegółowe kroki wykonywania algorytmów (LogStep), co pozwala użytkownikom:

- zrozumieć jak działa algorytm krok po kroku,
- zobaczyć transformacje danych na każdym etapie,
- debugować problemy z szyfrowaniem/deszyfrowaniem,
- analizować różnice między algorytmami.



Rysunek 20: Ekran przeglądu logów

Rysunek ?? przedstawia ekran przeglądu logów w aplikacji CryptoLab Mobile.



Rysunek 21: Szczegółowe logi dla przykładowej operacji

Rysunek ?? przedstawia szczegółowe logi dla przykładowej operacji szyfrowania w aplikacji CryptoLab Mobile.

18.4 Bezpieczeństwo kluczy

System automatycznie maskuje klucze w logach:

- Krótkie klucze (do 10 znaków) – wyświetlane w całości
- Średnie klucze (11-30 znaków) – maskowane środkowe znaki
- Długie klucze (powyżej 30 znaków) – pokazane tylko pierwsze i ostatnie 10 znaków
- Klucze RSA – maskowane dla bezpieczeństwa

18.5 Statystyki

LogManager oblicza i udostępnia statystyki użycia:

- łączna liczba operacji szyfrowania i deszyfrowania,
- najczęściej używany algorytm,
- wskaźnik sukcesu operacji,
- średni czas wykonania operacji.

18.6 Zastosowanie edukacyjne

System logowania ma kluczowe znaczenie edukacyjne:

- pozwala studentom zobaczyć jak algorytmy działają *od środka*,
- umożliwia porównanie kroków różnych algorytmów,
- pomaga zrozumieć złożoność obliczeniową,
- wspiera analizę błędów i debugowanie,
- dostarcza danych do tworzenia raportów i prezentacji.

19 Changelog

- **14.10.2025** Implementacja szyfru Cezara (szyfrowanie, deszyfrowanie, walidacja klucza) oraz podstawowe GUI.
Ulepszenie interfejsu użytkownika.
Implementacja AlgorithmRegistry z wzorcem Singleton.
Ulepszenie walidacji kluczy z szczegółowymi komunikatami o błędach.
- **20.10.2025** Dodanie szyfru Vigenère'a i szyfru z kluczem bieżącym.
Współpraca z singletonem AlgorithmRegistry.
Pełna implementacja szyfrowania Vigenère'a.
- **28.10.2025** Implementacja szyfru AES (Advanced Encryption Standard) z obsługą trzech trybów pracy: ECB, CBC, CTR.
Wsparcie dla kluczy AES-128, AES-192 i AES-256.
Dodanie paddingu PKCS#7 i obsługi wektorów inicjalizujących (IV).
Pełna implementacja algorytmu AES bez użycia zewnętrznych bibliotek kryptograficznych.
- **16.11.2025** Implementacja algorytmu RSA (Rivest-Shamir-Adleman) – pierwszy algorytm kriptografii asymetrycznej w aplikacji.
Generowanie par kluczy publiczny/prywatny z losowymi liczbami pierwszymi.
Implementacja algorytmu Euklidesa, rozszerzonego algorytmu Euklidesa i szybkiego potęgowania modularnego.
Dodanie kategorii "Kriptografia asymetryczna" w rejestrze algorytmów.
Wprowadzenie koncepcji szyfrowania hybrydowego w dokumentacji.
Dodanie GUI do generowania kluczy RSA – przycisk "Generuj klucze" z modelem wyświetlającym klucz publiczny i prywatny, możliwość kopiowania i bezpośredniego użycia kluczy w aplikacji.
- **24.11.2025** Implementacja zaawansowanego systemu logowania operacji kryptograficznych.
Utworzenie LogManager z wzorcem Singleton do centralnego zarządzania logami.
Dodanie komponentu LogsViewer do wyświetlania historii operacji z interfejsem użytkownika.
Rejestrowanie szczegółowych kroków algorytmów dla celów edukacyjnych.
Przechowywanie logów w AsyncStorage z limitem 100 wpisów.
Funkcje filtrowania, usuwania i eksportu logów.
Automatyczne maskowanie kluczy dla bezpieczeństwa.

Statystyki użycia algorytmów (liczba operacji, najczęściej używany algorytm, czas wykonania).

Integracja systemu logowania ze wszystkimi algorytmami (Cezar, Vigenère, Running Key, AES, RSA).

- **01.12.2025** Implementacja algorytmu ElGamal oraz protokołu ECDH (Elliptic Curve Diffie-Hellman).

Dodanie obsługi problemu logarytmu dyskretnego w ElGamal.

Implementacja operacji na krzywych eliptycznych (dodawanie punktów, mnożenie skalarne) dla ECDH.

Demonstracja schematu ECIES (Elliptic Curve Integrated Encryption Scheme).

Rozszerzenie dokumentacji o nowe algorytmy asymetryczne.

- **07.12.2025** Implementacja funkcji skrótu SHA-256.

Rejestrowanie szczegółowych kroków funkcji skrótu w systemie logowania.

Dodanie zastosowań SHA-256 w praktyce (blockchain, podpisy cyfrowe).

Ulepszenie dokumentacji o funkcje skrótu i ich znaczenie w bezpieczeństwie.