

Dokumentacja Projektu CryptoLab Mobile

Agnieszka Ryś

14 grudnia 2025

Spis treści

1 Cel projektu	2
2 Podstawy kryptografii	2
3 Technologie wykorzystane w projekcie	3
4 Architektura systemu	4
4.1 Wzorzec projektowy	4
4.2 Komponenty główne	4
5 Struktura projektu	5
6 Implementacja szyfru Cezara	5
6.1 Podstawy	5
6.2 Model matematyczny	5
6.3 Cechy implementacji	5
7 Implementacja szyfru Vigenère'a	6
7.1 Historia i znaczenie	6
7.2 Podstawy	6
7.3 Model matematyczny	6
7.4 Przykład działania	6
7.5 Cechy implementacji	6
8 Implementacja szyfru z kluczem bieżącym	7
8.1 Historia i koncepcja	7
8.2 Algorytm szyfrowania i deszyfrowania	7
8.3 Przykład działania	7
8.4 Bezpieczeństwo	8
8.5 One-Time Pad (OTP)	8
8.6 Zastosowania historyczne	8
8.7 Cechy implementacji w CryptoLab	9

9 Implementacja szyfru AES	10
9.1 Podstawy	10
9.2 Warianty AES	10
9.3 Tryby pracy AES	11
9.4 Struktura algorytmu AES	13
9.5 Cechy implementacji	13
9.6 Bezpieczeństwo	13
9.7 Zastosowania	14
10 Kryptografia asymetryczna - RSA	15
10.1 Model matematyczny	15
10.2 Przykład działania	15
10.3 Cechy implementacji	16
10.4 Jak wygenerować klucze RSA	16
10.5 Bezpieczeństwo	17
10.6 Zastosowania	17
11 Algorytm ElGamal	18
11.1 Historia i znaczenie	18
11.2 Model matematyczny	18
11.3 Cechy implementacji	18
12 Protokół ECDH i Krzywe Eliptyczne	19
12.1 Wprowadzenie	19
12.2 Protokół ECDH	19
12.3 Implementacja ECIES	20
13 Funkcja skrótu SHA-256	20
13.1 Wprowadzenie	20
13.2 Algorytm	20
13.3 Zastosowania	21
13.4 Bezpieczeństwo	21
13.5 Implementacja	21
14 Podpis Elektroniczny (Digital Signature)	21
14.1 Wprowadzenie	21
14.2 Schemat podpisu RSA-SHA256	21
14.3 Format	22
14.4 Parametry RSA (edukacyjne)	22
14.5 Bezpieczeństwo	22
14.6 Zastosowania	22
14.7 Implementacja	22
15 Wybrane fragmenty kodu	25
15.1 Klasa bazowa algorytmu	25
15.2 Implementacja szyfru Cezara	27
15.3 Implementacja szyfru Vigenère'a	28
15.4 Implementacja szyfru z kluczem bieżącym	29
15.5 Implementacja szyfru AES	31

15.6 Implementacja szyfru RSA	31
15.7 Implementacja algorytmu ElGamal	34
15.8 Implementacja algorytmu ECDH	35
15.9 Implementacja funkcji SHA-256	36
15.10 Implementacja podpisu elektronicznego	37
16 Podsumowanie	38
16.1 Szyfr Cezara	38
16.2 Szyfr Vigenèrè'a	38
16.3 Szyfr z kluczem bieżącym	39
16.4 Szyfr AES	39
16.5 Szyfr RSA	40
16.6 Algorytm ElGamal	41
16.7 Protokół ECDH	41
16.8 Funkcja SHA-256	41
16.9 Podpis Elektroniczny	43
17 System logowania operacji	44
17.1 Architektura systemu logowania	44
17.2 Rejestrowane informacje	45
17.3 Kroki algorytmu	45
17.4 Bezpieczeństwo kluczy	46
17.5 Statystyki	46
17.6 Zastosowanie edukacyjne	47
18 Changelog	47

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

Kryptografia to ochrona informacji poprzez przekształcanie jej w formę nieczytelną. W kryptografii **symetrycznej** ten sam klucz służy do szyfrowania i deszyfrowania - cechuje się wysoką szybkością, ale wymaga bezpiecznej wymiany klucza.

Aplikacja implementuje szyfry klasyczne (Cezara, Vigenère'a, z kluczem bieżącym) oraz nowoczesne (AES, RSA, ElGamal, ECDH, SHA-256, podpis elektroniczny). Choć klasyczne szyfry nie zapewniają dziś bezpieczeństwa, stanowią doskonałe narzędzie dydaktyczne do zrozumienia podstawowych pojęć: klucza, przestrzeni kluczy, analizy częstości i ataków brute-force.

Ewolucja: od klasyki do nowoczesności

- **Szyfry klasyczne** - proste podstawienia (Cezar: przesunięcie liter o stałą wartość)
- **AES** - szyfr blokowy 128-bit z trybami ECB/CBC/CTR, odporny na znane ataki
- **RSA/ECC** - kryptografia asymetryczna z parą kluczy publiczny/prywatny
- **SHA-256** - funkcja skrótu jednokierunkowa do integralności danych
- **Podpis cyfrowy** - RSA+SHA256 do autentykacji i niezaprzeczalności

3 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.

4 Architektura systemu

4.1 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.

4.2 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.

5 Struktura projektu

```
crypto-lab-mobile/
    App.tsx                                (glowny komponent)
    package.json                            (zalezno ci projektu)
    tsconfig.json                           (konfiguracja TypeScriptu)
    app.json                                (konfiguracja Expo)
    src/
        algorithms/
            CryptographicAlgorithm.ts     (klasa bazowa)
            CaesarCipher.ts              (szynfr Cezara)
            VigenereCipher.ts           (szynfr Vigenere'a)
            RunningKeyCipher.ts        (szynfr z kluczem
                                         biezacym)
            AESCipher.ts                (szynfr AES)
            RSACipher.ts                (szynfr RSA)
            AlgorithmRegistry.ts       (rejestr algorytmow)
        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)
```

6 Implementacja szyfru Cezara

6.1 Podstawy

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

6.2 Model matematyczny

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

6.3 Cechy implementacji

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

7 Implementacja szyfru Vigenère'a

7.1 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.

7.2 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.

7.3 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.

7.4 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

7.5 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.

8 Implementacja szyfru z kluczem bieżącym

Szyfr z kluczem bieżącym (Running Key Cipher) to polialfabetyczny szyfr substytucyjny będący rozwinięciem szyfru Vigenère'a. Główna różnica polega na tym, że klucz ma długość co najmniej równą długości szyfrowanej wiadomości, co znaczowo zwiększa bezpieczeństwo.

8.1 Historia i koncepcja

Szyfr z kluczem bieżącym był używany już w XIX wieku. Klasycznie jako klucza używano fragmentów książek, gazet lub innych długich tekstów. Idea polega na tym, że klucz nie powtarza się (lub powtarza bardzo rzadko), co eliminuje główną słabość szyfru Vigenère'a - podatność na analizę Kasiskiego.

Gdy klucz jest:

- **tekstem z książki** - szyfr jest podatny na kryptoanalizę (klucz ma strukturę języka naturalnego),
- **losowy i używany jednorazowo** - otrzymujemy **One-Time Pad** (teoretycznie nie do złamania).

8.2 Algorytm szyfrowania i deszyfrowania

Model matematyczny jest identyczny jak w szyfrze Vigenère'a, ale klucz ma długość $|k| \geq |x|$:

Szyfrowanie:

- $E_k(x_i) = (x_i + k_i) \bmod 26$
- gdzie x_i - i -ty znak tekstu jawnego (0-25),
- k_i - i -ty znak klucza (0-25),
- wynik - zaszyfrowany znak.

Deszyfrowanie:

- $D_k(c_i) = (c_i - k_i + 26) \bmod 26$
- gdzie c_i - i -ty znak szyfrogramu.

8.3 Przykład działania

Załóżmy, że chcemy zaszyfrować słowo **ATTACK** używając klucza **LOREMIPSUM** (fragmentu tekstu Lorem Ipsum):

Tekst jawnny	A	T	T	A	C	K
Pozycja	0	19	19	0	2	10
Klucz	L	O	R	E	M	I
Pozycja klucza	11	14	17	4	12	8
Suma (mod 26)	11	7	10	4	14	18
Tekst zaszyfrowany	L	H	K	E	O	S

Wynik: **ATTACK** → **LHKEOS**

8.4 Bezpieczeństwo

Zalety:

- Eliminuje periodyczność klucza - ataki typu Kasiskiego są nieskuteczne,
- Przy losowym kluczu jednorazowym - teoretycznie bezpieczny (doskonała tajemnica),
- Długi klucz znaczco utrudnia kryptoanalizę statystyczną.

Wady:

- Wymaga bardzo długich kluczy (minimum długość tekstu),
- Klucz musi być przekazany bezpiecznym kanałem,
- Przy użyciu tekstu naturalnego jako klucza - podatny na kryptoanalizę (struktura języka),
- Praktycznie niepraktyczny w rzeczywistych zastosowaniach (poza OTP).

Atak na szyfr: Jeśli klucz jest tekstem w języku naturalnym, można zastosować:

- Analizę statystyczną struktur języka,
- Ataki słownikowe (próbowanie znanych tekstów jako kluczy),
- Analizę bigramów i trigramów.

8.5 One-Time Pad (OTP)

Gdy klucz spełnia warunki:

1. Jest **całkowicie losowy**,
2. Ma długość **co najmniej równą tekstowi**,
3. Jest używany **tylko raz**,
4. Jest znany **tylko nadawcy i odbiorcy**,

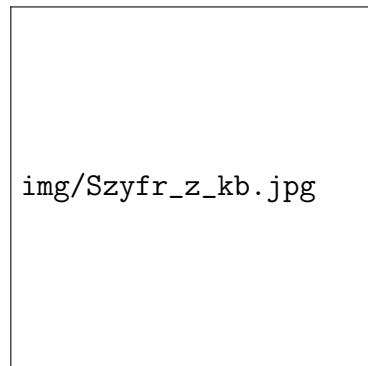
otrzymujemy One-Time Pad - jedyny szyfr z udowodnioną matematycznie **doskonałą tajemnicą** (Shannon, 1949).

8.6 Zastosowania historyczne

- **Diplomacja** - szyfrowanie tajnych depesz (XIX-XX wiek),
- **Wojskowość** - komunikacja wojskowa (WWII, Zimna Wojna),
- **Czerwony Telefon** - linia Waszyngton-Moskwa używała OTP,
- **Wywiad** - agenci używali książek jako kluczy.

8.7 Cechy implementacji w CryptoLab

- Klucz może zawierać litery (A-Z, a-z) i spacje,
- Minimalna długość klucza: 5 liter (bez liczenia spacji),
- Automatyczne generowanie klucza z Lorem Ipsum,
- Wynik w formacie: <klucz>:<szyfrogram>,
- Znaki niebędące literami w tekście źródłowym są przepisywane bez zmian,
- Obsługa wielkich i małych liter,
- Szczegółowe logowanie każdego kroku procesu.



Rysunek 1: Ekran szyfru z kluczem bieżącym w aplikacji CryptoLab

Rysunek 1 przedstawia interfejs użytkownika szyfru z kluczem bieżącym w aplikacji CryptoLab Mobile. Widoczny jest automatycznie wygenerowany klucz oraz proces szyfrowania wiadomości.

9 Implementacja szyfru AES

AES (Advanced Encryption Standard, 2001, Daemen/Rijmen) to najpopularniejszy szyfr symetryczny - chroni SSL/TLS, systemy bankowe, dyski. Następca DES, obecnie globalny standard bezpieczeństwa.

9.1 Podstawy

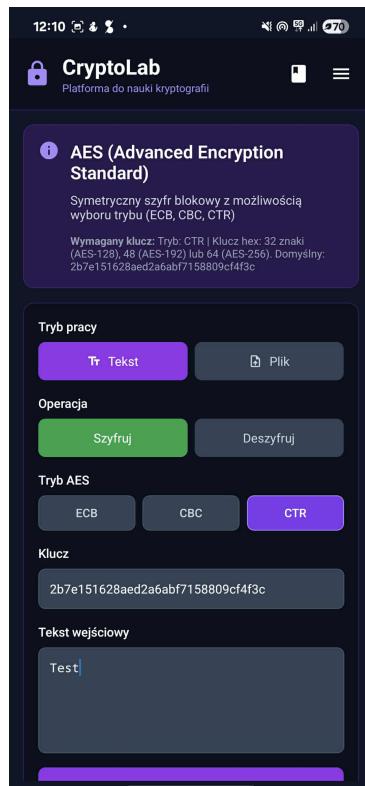
AES to szyfr **blokowy**, który operuje na blokach danych o długości 128 bitów (16 bajtów). W przeciwieństwie do szyfrów klasycznych, AES wykorzystuje skomplikowane operacje matematyczne na macierzach bajtów, w tym podstawienia (S-Box), permutacje, mieszanie kolumn i dodawanie klucza rundowego.

9.2 Warianty AES

AES występuje w trzech wariantach, różniących się długością klucza:

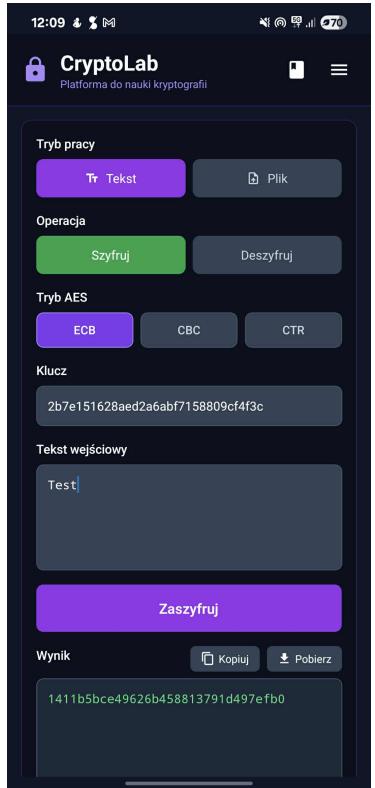
- **AES-128** - klucz 128-bitowy (32 znaki hex), 10 rund szyfrowania,
- **AES-192** - klucz 192-bitowy (48 znaków hex), 12 rund szyfrowania,
- **AES-256** - klucz 256-bitowy (64 znaki hex), 14 rund szyfrowania.

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



Rysunek 2: Strona główna szyfru AES w aplikacji CryptoLab

Rysunek 2 przedstawia interfejs użytkownika aplikacji CryptoLab Mobile dla szyfru AES.



Rysunek 3: Tryb ECB w szyfrze AES

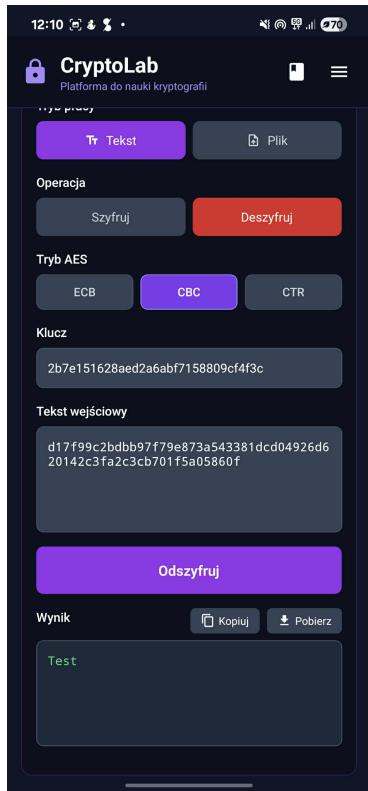
9.3 Tryby pracy AES

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

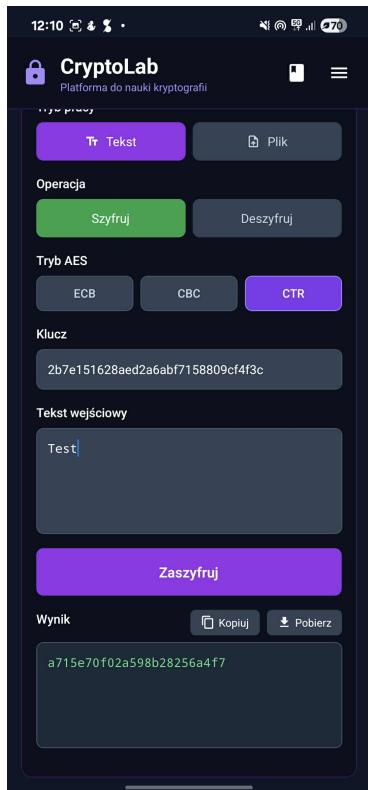
ECB (Electronic Codebook) Najprostszy tryb - każdy blok szyfrowany jest niezależnie tym samym kluczem. **Niezalecany** w praktyce, ponieważ identyczne bloki tekstu jawnego dają identyczne bloki szyfrogramu, co może ujawnić wzorce w danych. Rysunek 3 przedstawia przykładowe szyfrowanie tekstu w trybie 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 4 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 5 przedstawia przykładowe szyfrowanie tekstu w trybie CTR.



Rysunek 4: Tryb CBC w szyfrze AES



Rysunek 5: Tryb CTR w szyfrze AES

9.4 Struktura algorytmu AES

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

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.

9.5 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.

9.6 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.

9.7 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).

10 Kryptografia asymetryczna - RSA

RSA (Rivest-Shamir-Adleman, 1977 MIT) rozwiązał problem bezpiecznej wymiany kluczy. Para kluczy: **publiczny** (szyfrowanie, można udostępniać) i **prywatny** (deszyfrowanie, tajny). Bezpieczeństwo: trudność faktoryzacji dużych liczb złożonych ($n = p \cdot q$).

10.1 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

10.2 Przykład działania

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

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

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

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

Klucz publiczny: $(17, 3233)$

Klucz prywatny: $(2753, 3233)$

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

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

Deszyfrowanie:

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

10.3 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.

10.4 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: $(233, 323)$

10.5 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.

10.6 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 Algorytm ElGamal

11.1 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.

11.2 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) .

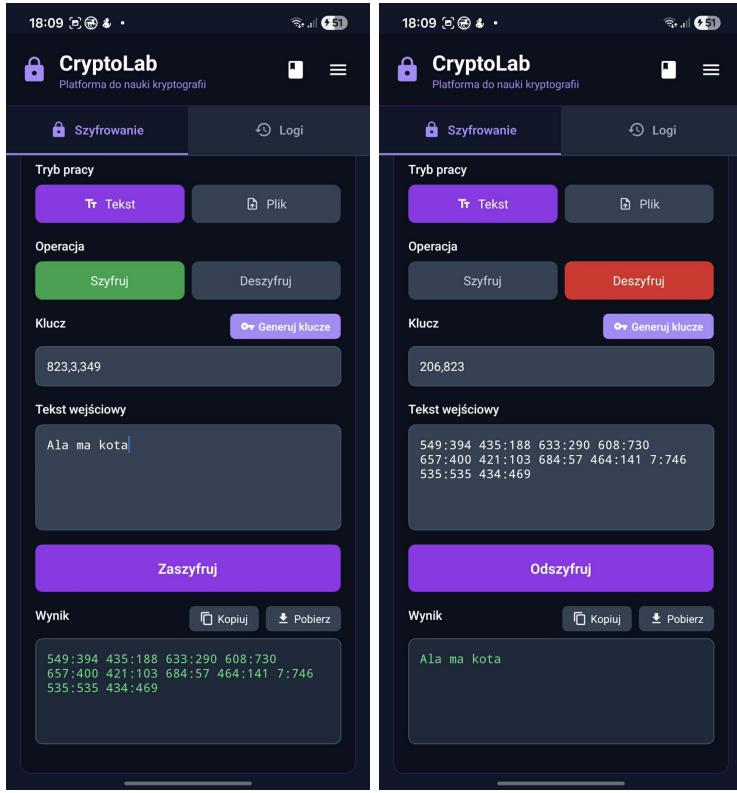
Deszyfrowanie: Aby odszyfrować parę (a, b) przy użyciu klucza prywatnego x :

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

11.3 Cechy implementacji

Implementacja wykorzystuje liczby pierwsze w zakresie 300-1000, szyfruje znaki osobno generując pary (a, b) z losowym kluczem k (niedeterministyczność).

Rysunek 6 przedstawia interfejs użytkownika aplikacji CryptoLab Mobile dla szyfru ElGamal.



Rysunek 6: Ekran szyfru i deszyfru ElGamal w aplikacji CryptoLab

12 Protokół ECDH i Krzywe Eliptyczne

12.1 Wprowadzenie

ECC zapewnia bezpieczeństwo porównywalne z RSA przy krótszych kluczach (256-bit ECC \approx 3072-bit RSA). Równanie krzywej: $y^2 = x^3 + ax + b \pmod{p}$.

12.2 Protokół ECDH

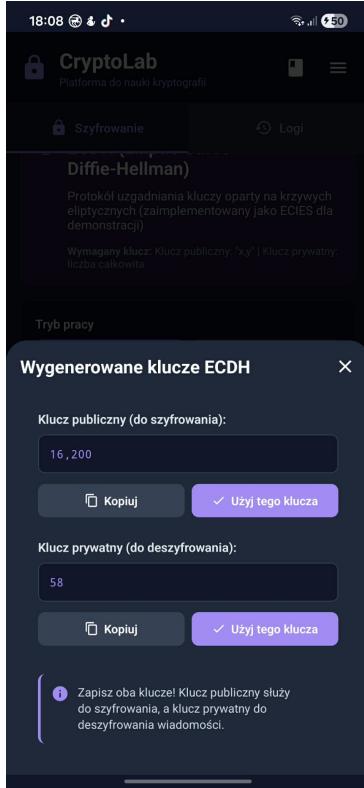
ECDH służy do uzgodnienia wspólnego sekretu przez niezabezpieczony kanał.

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 .

12.3 Implementacja ECIES

Aplikacja implementuje schemat ECIES: nadawca generuje klucz efemeryczny, oblicza wspólny sekret, używa współrzędnej X jako klucza XOR. Rysunek 7 przedstawia wyge-



Rysunek 7: Wygenerowane klucze ECDH w aplikacji CryptoLab

nerowane klucze ECDH w aplikacji CryptoLab Mobile. Rysunek 8 przedstawia proces szyfrowania i deszyfrowania ECDH w aplikacji CryptoLab Mobile.

13 Funkcja skrótu SHA-256

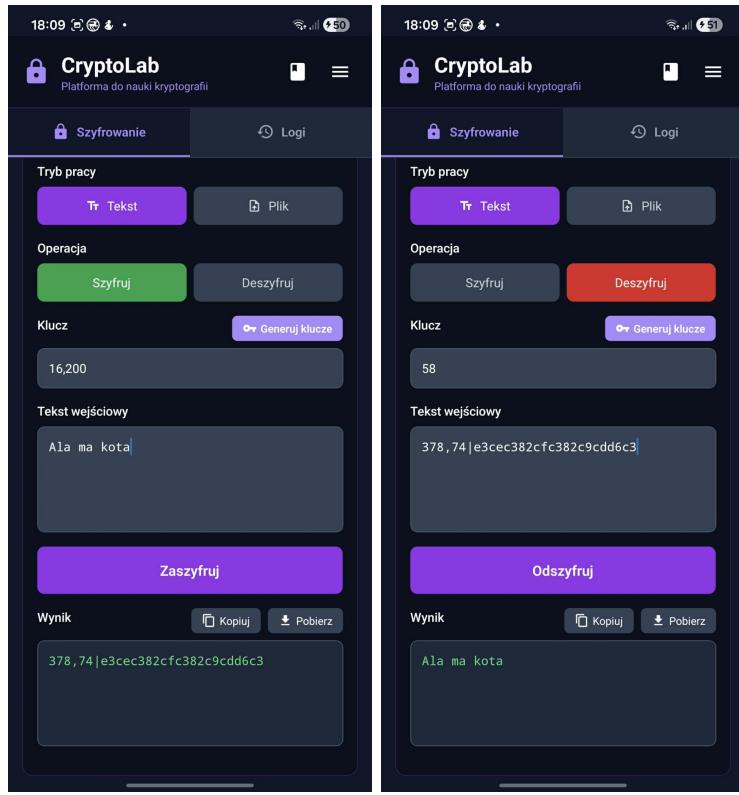
13.1 Wprowadzenie

Funkcje skrótu są jednokierunkowe i generują skrót stałej długości. Cechy: deterministyczność, efekt lawiny, odporność na kolizje, jednokierunkowość.

SHA-256 (SHA-2, NIST 2001): 256-bitowy skrót, 512-bitowy blok, 64 rundy, operacje bitowe.

13.2 Algorytm

1. **Preprocessing:** bit '1', zera do $\equiv 448 \pmod{512}$, 64-bit długości.
2. **Inicjalizacja:** Stałe $H_0 = 0x6a09e667, \dots, H_7 = 0x5be0cd19$.
3. **Kompresja 64 rund:** Harmonogram W_0, \dots, W_{63} , funkcje $Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$, $Maj(x, y, z)$, Σ , σ .
4. **Wynik:** Konkatenacja H_0 do H_7 .



Rysunek 8: Proces szyfrowania i deszyfrowania ECDH w aplikacji CryptoLab

13.3 Zastosowania

Podpisy cyfrowe, blockchain (Bitcoin), SSL/TLS, sumy kontrolne, HMAC, haszowanie haseł.

13.4 Bezpieczeństwo

Brak znanych ataków kolizyjnych. Złożoność brutalna: 2^{256} , urodzinowy: 2^{128} .

13.5 Implementacja

Pełna implementacja: preprocessing, funkcje $Ch/Maj/\Sigma/\sigma$, 64 rundy, UTF-8, szczegółowe logi. Funkcja jednokierunkowa - brak deszyfrowania.

14 Podpis Elektroniczny (Digital Signature)

14.1 Wprowadzenie

Podpis cyfrowy zapewnia: autentyczność, integralność, niezaprzecjalność (w odróżnieniu od szyfrowania dającego poufność).

14.2 Schemat podpisu RSA-SHA256

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

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

Proces weryfikacji:

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

14.3 Format

dokument|hash|podpis|klucz_publiczny

14.4 Parametry RSA (edukacyjne)

$p = 61, q = 53, n = 3233, e = 17, d = 2753$. W produkcji: 2048-4096 bitów.

14.5 Bezpieczeństwo

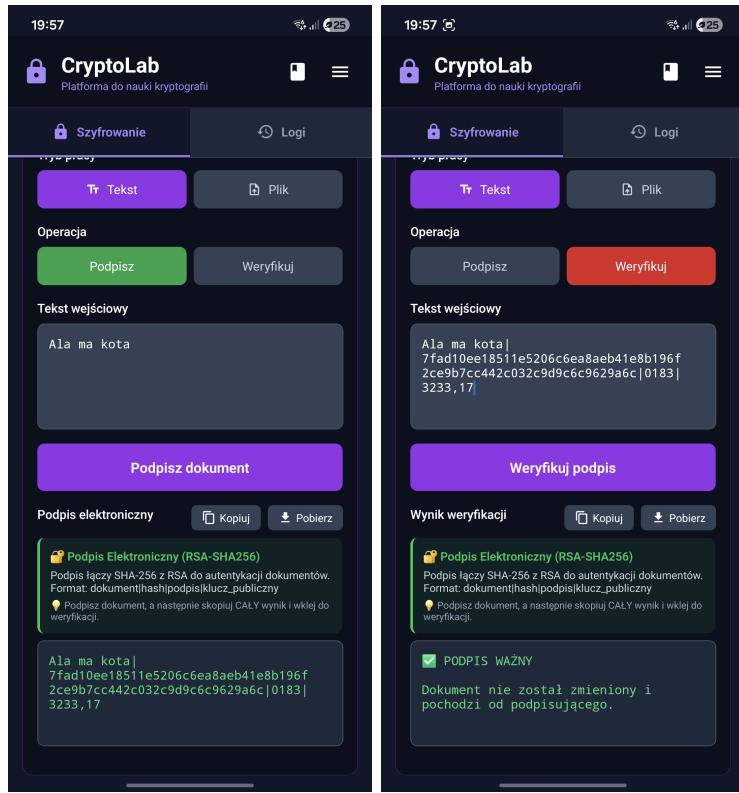
Wymaga: tajnego klucza prywatnego, silnych kluczy, bezpiecznej funkcji skrótu (SHA-256), paddingu (PSS).

14.6 Zastosowania

SSL/TLS, podpisywanie oprogramowania, dokumenty elektroniczne (eIDAS), transakcje finansowe, blockchain, e-mail (S/MIME, PGP), aktualizacje systemu.

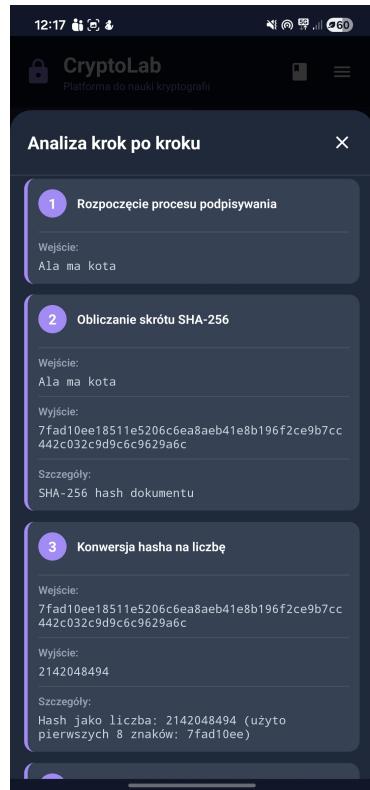
14.7 Implementacja

Pełna implementacja: generowanie kluczy RSA (12-bit edukacyjne), integracja SHA-256, modular exponentiation, rozszerzony algorytm Euklidesa, szczegółowe logi, wykrywanie modyfikacji, validacja WAŻNY/NIEWAŻNY.



Rysunek 9: Ekran podpisywania dokumentu w aplikacji CryptoLab i weryfikacja podpisu

Rysunek 9 przedstawia proces podpisywania dokumentu - system generuje pełny podpis cyfrowy (dokument—hash—podpis RSA—klucz publiczny). Natomiast rysunek obok, weryfikację podpisu - system sprawdza integralność (SHA-256) i autentyczność (klucz publiczny), wyświetlając PODPIS WAŻNY/NIEWAŻNY.



Rysunek 10: Szczegółowe logi procesu podpisywania

Rysunek 10 pokazuje szczegółowe logi procesu podpisywania: SHA-256, konwersja hasha, podpiswanie kluczem prywatnym.

15 Wybrane fragmenty kodu

15.1 Klasa bazowa algorytmu

Listing 2: Klasa abstrakcyjna CryptographicAlgorithm

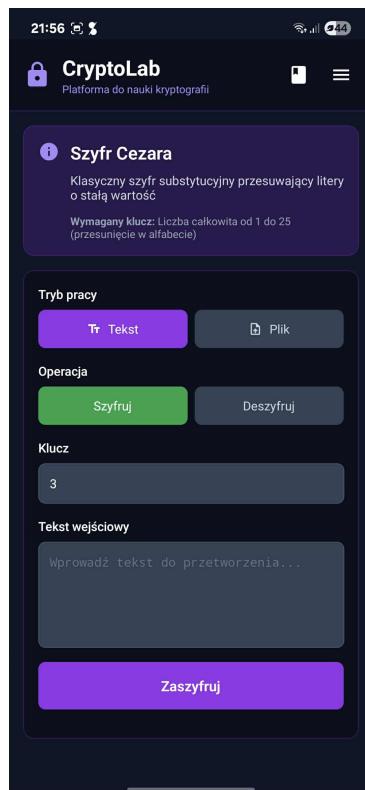
```
export default class CryptographicAlgorithm {
    name: string;
    description: string;
    category: string;

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

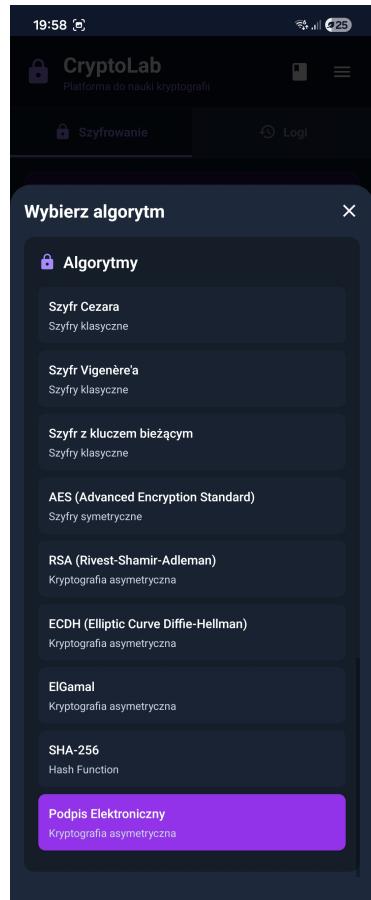
    decrypt(ciphertext: string, key: string): string {
        throw new Error('Metoda decrypt() musi być zaimplementowana');
    }

    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 11: Ekran główny aplikacji CryptoLab



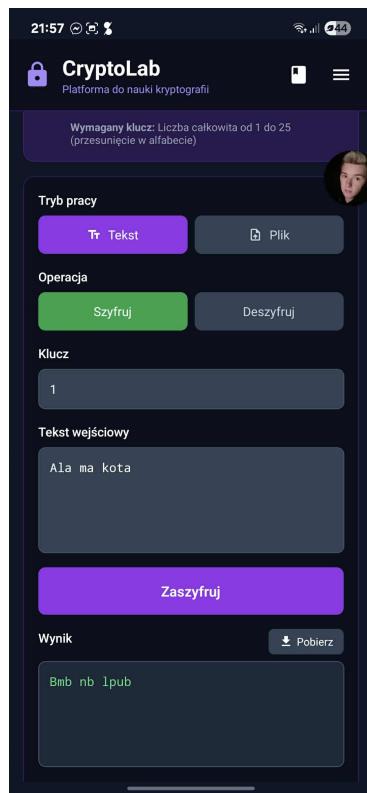
Rysunek 12: Lista z możliwością wyboru algorytmu

Rysunek 11 przedstawia ekran główny aplikacji CryptoLab Mobile, a rysunek 12 pokazuje listę dostępnych algorytmów kryptograficznych.

15.2 Implementacja szyfru Cezara

Listing 3: Fragment CaesarCipher

```
encrypt(plaintext: string, key: string): 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 13: Test szyfru Cezara

Rysunek 13 przedstawia przykładowy test szyfru Cezara w aplikacji CryptoLab Mobile.

15.3 Implementacja szyfru Vigenère'a

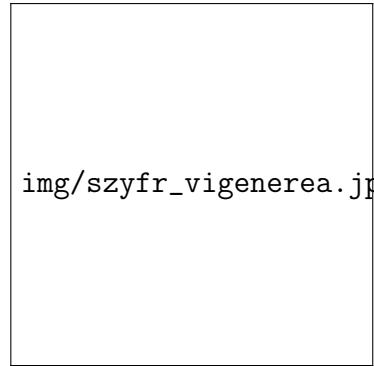
Listing 4: Kluczowy fragment VigenereCipher

```
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 = char.charCodeAt(0) - base;
            // Klucz powtarza się cyklicznie
            const resultCode = normalizedKey.charCodeAt(keyIndex % normalizedKey.length) - 65;

            // Operacja kryptograficzna Vigenere
            const resultCode = encrypt
                ? (resultCode + keyCode) % 26
                : (resultCode - keyCode + 26) % 26;

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



Rysunek 14: Ekran szyfru Vigenere'a

Rysunek 14 przedstawia interfejs użytkownika szyfru Vigenère'a w aplikacji CryptoLab Mobile.

15.4 Implementacja szyfru z kluczem bieżącym

Listing 5: Fragmenty klasy RunningKeyCipher

```
export default class RunningKeyCipher extends CryptographicAlgorithm {
    constructor() {
        super(
            'Szyfr z kluczem bie   cym',
            'Szyfr podobny do Vigen re\`a, ale u  ywaj  cy klucza o
             d ugo ci tekstu',
            'Szyfry klasyczne'
        );
    }

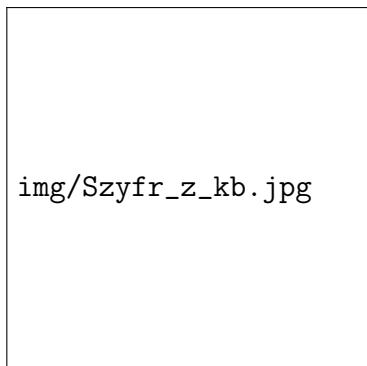
    validateKey(key: string): { valid: boolean; error?: string } {
        if (!key || key.trim().length === 0) {
            return { valid: false, error: 'Klucz nie mo  e by  pusty' };
        }

        // Sprawd  czy klucz zawiera tylko litery
        const hasOnlyLetters = /^[a-zA-Z\s]+$/;
        if (!hasOnlyLetters) {
            return { valid: false, error: 'Klucz mo  e zawiera  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 zawiera  co najmniej 5 liter (mo  e
                    zawiera  spacje)'
            };
        }

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

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



Rysunek 15: Ekran szyfru z kluczem bieżącym

Rysunek 15 przedstawia interfejs użytkownika szyfru z kluczem bieżącym w aplikacji CryptoLab Mobile.

15.5 Implementacja szyfru AES

Listing 6: Kluczowe metody AESCipher

```
// Szyfrowanie w zależności od trybu (ECB/CBC/CTR)
encrypt(plaintext: string, key: string): string {
    if (this.mode === 'ECB') return this.encryptECB(plaintext, key);
    if (this.mode === 'CBC') return this.encryptCBC(plaintext, key);
    if (this.mode === 'CTR') return this.encryptCTR(plaintext, key);
}

// ECB - każdy blok niezależnie (NIEZALECANY!)
private encryptECB(plaintext: string, key: string): string {
    const blocks = this.textToBlocks(plaintext);
    return blocks.map(block => this.encryptBlock(block, key))
        .join('');
}

// CBC - z wektorem IV, XOR z poprzednim blokiem
private encryptCBC(plaintext: string, key: string): string {
    const iv = this.generateIV();
    let previousBlock = iv;
    // XOR każdego bloku z poprzednim szyfrogramem
    // ...
}
```

15.6 Implementacja szyfru RSA

Listing 7: Kluczowe operacje RSA

```
// Generowanie kluczy
generateKeyPair(): RSAKeyPair {
    const p = generatePrime(100, 300);
    const q = generatePrime(100, 300);
    const n = p * q;                                // Moduł
    const phi = (p - 1) * (q - 1);                  // Funkcja Eulera
    const e = 65537;                                 // Wykładnik publiczny
    const d = modInverse(e, phi);                    // Wykładnik prywatny

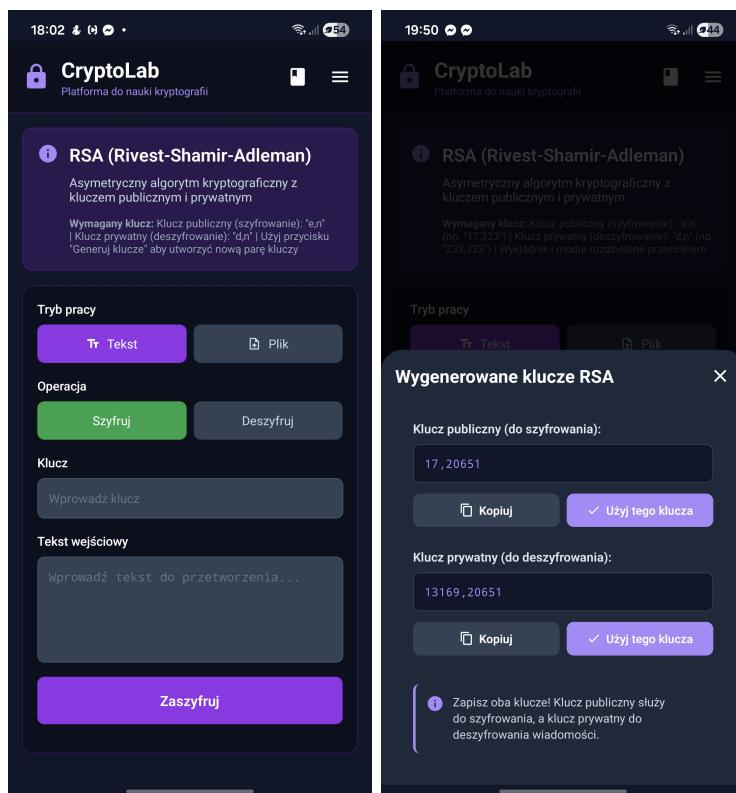
    return {
        publicKey: { e, n },
        privateKey: { d, n }
    };
}

// Szyfrowanie: c = m^e mod n
encrypt(plaintext: string, key: string): string {
    const [e, n] = key.split(',').map(p => parseInt(p, 10));
    const encrypted = [];

    for (const char of plaintext) {
        const m = char.charCodeAt(0);
        const c = modPow(BigInt(m), BigInt(e), BigInt(n));
        encrypted.push(Number(c));
    }
    return encrypted.join(' ');
}
```

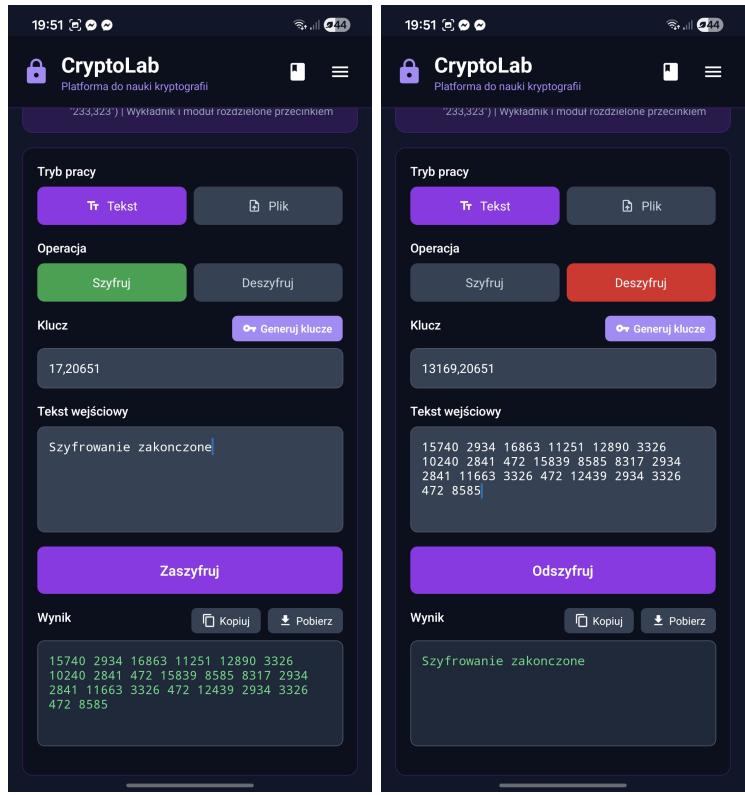
```
// Deszyfrowanie: m = c^d mod n
decrypt(ciphertext: string, key: string): string {
  const [d, n] = key.split(',').map(p => parseInt(p, 10));
  const numbers = ciphertext.split(' ').map(s => parseInt(s));

  return numbers.map(c => {
    const m = modPow(BigInt(c), BigInt(d), BigInt(n));
    return String.fromCharCode(Number(m));
  }).join('');
}
```



Rysunek 16: Ekran szyfrow RSA oraz generowanie kluczy

Rysunek 16 przedstawia interfejs użytkownika szyfrow RSA oraz generowania kluczy w aplikacji CryptoLab Mobile.



Rysunek 17: Test szyfru RSA i jego deszyfrowanie

Rysunek 17 przedstawia przykładowy test szyfru RSA oraz jego deszyfrowanie w aplikacji CryptoLab Mobile.

15.7 Implementacja algorytmu ElGamal

Listing 8: Kluczowe operacje ElGamal

```
// Generowanie kluczy
generateKeyPair(): ElGamalKeyPair {
    const p = generatePrime(300, 1000);           // Liczba pierwsza
    const g = findPrimitiveRoot(p);                // Generator
    const x = random(2, p-2);                     // Klucz prywatny
    const y = modPow(g, x, p);                    // Klucz publiczny: y = g^x
    mod p

    return { publicKey: {p, g, y}, privateKey: {x, p} };
}

// Szyfrowanie: (a, b) = (g^k mod p, y^k * m mod p)
encrypt(plaintext: string, key: string): string {
    const [p, g, y] = key.split(',').map(BigInt);
    const encrypted = [];

    for (const char of plaintext) {
        const m = BigInt(char.charCodeAt(0));
        const k = random(1, p-2);                  // Losowy klucz efemeryczny
        const a = modPow(g, k, p);                 // a = g^k mod p
        const b = (modPow(y, k, p) * m) % p; // b = (y^k * m) mod p
        encrypted.push(`${a}:${b}`);
    }
    return encrypted.join(' ');
}
```

15.8 Implementacja algorytmu ECDH

Listing 9: Kluczowe operacje ECDH

```
// Dodawanie punkt w na krzywej:  $y^2 = x^3 + ax + b \pmod{p}$ 
private addPoints(P: Point, Q: Point): Point {
    if (P.isInfinity) return Q;
    if (Q.isInfinity) return P;

    // Wsp czynnik nachylenia prostej
    const m = (P === Q)
        ? (3n * P.x * P.x + CURVE.a) / (2n * P.y) // Podw jenie
        : (Q.y - P.y) / (Q.x - P.x); // Dodawanie

    const x3 = (m * m - P.x - Q.x) % CURVE.p;
    const y3 = (m * (P.x - x3) - P.y) % CURVE.p;
    return { x: x3, y: y3 };
}

// Szyfrowanie ECIES
encrypt(plaintext: string, key: string): string {
    const Q = parsePoint(key); // Klucz publiczny odbiorcy
    const k = random(1, CURVE.p-1); // Losowy klucz efemeryczny
    const R = k * CURVE.G; // Punkt efemeryczny: R = k*G
    const S = k * Q; // Wsp lny sekret: S = k*Q
    const symmetricKey = S.x; // Klucz XOR
    const ciphertext = XOR(plaintext, symmetricKey);
    return `${R.x},${R.y}|${ciphertext}`; // (R, szyfrogram)
}
```

15.9 Implementacja funkcji SHA-256

Listing 10: Kluczowe operacje SHA-256

```
// Sta e i wartosci poczatkowe
private K: number[] = [0x428a2f98, /* ...63 wartosci... */, 0xc67178f2
];
private H: number[] = [0x6a09e667, 0xbb67ae85, /* ...6 wartosci... */];

encrypt(plaintext: string): string {
    const bytes = stringToBytes(plaintext);
    return bytesToHex(this.sha256(bytes));
}

// Gorny algorytm - 64 rundy kompresji
private sha256(message: number[]): number[] {
    const blocks = this.preprocess(message); // Padding do 512-bit
    let H = [...this.H]; // Kopia wartosci
    poczatkowych

    for (const block of blocks) {
        const W = this.prepareSchedule(block); // Harmonogram 64 rund w
        let [a, b, c, d, e, f, g, h] = H;

        // 64 rundy
        for (let t = 0; t < 64; t++) {
            const T1 = h + Sigma1(e) + Ch(e,f,g) + this.K[t] + W[t];
            const T2 = Sigma0(a) + Maj(a,b,c);
            [a,b,c,d,e,f,g,h] = [T1+T2, a, b, c, d+T1, e, f, g];
        }

        // Dodaj do H
        H = H.map((v, i) => (v + [a,b,c,d,e,f,g,h][i]) >>> 0);
    }
    return H;
}

// Funkcje logiczne
Ch(x,y,z) { return (x & y) ^ (~x & z); }
Maj(x,y,z) { return (x & y) ^ (x & z) ^ (y & z); }
Sigma0(x) { return ROTR(x,2) ^ ROTR(x,13) ^ ROTR(x,22); }
Sigma1(x) { return ROTR(x,6) ^ ROTR(x,11) ^ ROTR(x,25); }
```

15.10 Implementacja podpisu elektronicznego

Listing 11: Kluczowe operacje DigitalSignature

```
// Parametry RSA (edukacyjne): p=61, q=53, n=3233, e=17
private P = 61n, Q = 53n, N = 3233n, E = 17n;
private D = modularInverse(this.E, (P-1n)*(Q-1n)); // d=2753

// Podpisywanie: sign(doc) = (doc | hash | hash^d mod n | public_key)
encrypt(document: string): string {
    const hash = SHA256(document); // 1. Hash dokumentu
    const hashNum = hashToNumber(hash); // 2. Hash -> liczba (3 hex)
    const signature = modPow(hashNum, D, N); // 3. Podpis: hash^d mod n
    return `${document}|${hash}|${signature.toString(16)}|${N},${E}`;
}

// Weryfikacja: verify(signed_doc)
decrypt(signedDoc: string): string {
    const [doc, origHash, sigHex, pubKey] = signedDoc.split('|');
    const [n, e] = pubKey.split(',').map(BigInt);

    // 1. Sprawdza integralno\u0107a
    const currentHash = SHA256(doc);
    if (currentHash !== origHash)
        return 'PODPIS NIEWA\u017bNY - Dokument zmieniony!';

    // 2. Sprawdza autentyczno\u0107a
    const hashNum = hashToNumber(currentHash);
    const signature = BigInt('0x' + sigHex);
    const verifiedHash = modPow(signature, e, n); // signature^e mod n

    return (hashNum === verifiedHash)
        ? 'PODPIS WA\u017bNY - Dokument autentyczny!'
        : 'PODPIS NIEWA\u017bNY - Podpis fa\u0142szywy!';
}
```

16 Podsumowanie

16.1 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.

16.2 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).

16.3 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.

16.4 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.

16.5 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.

16.6 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).

16.7 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.

16.8 Funkcja SHA-256

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

Zalety:

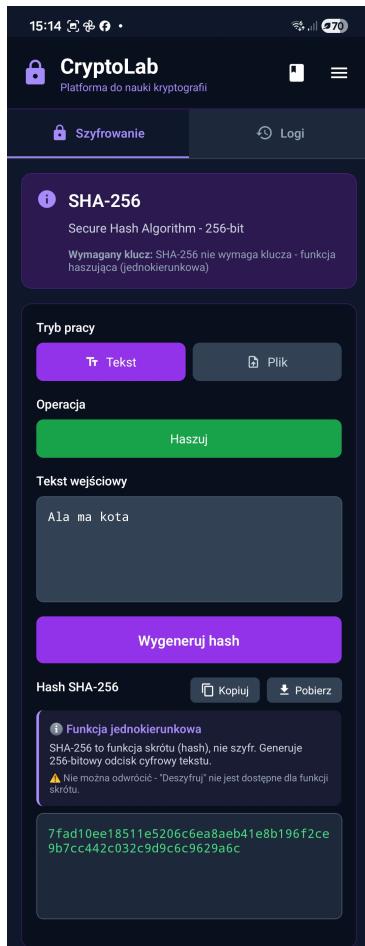
- **jednokierunkowość** - niemożliwe odtworzenie oryginalnej wiadomości,
- **deterministyczność** - ten sam wejście zawsze daje ten sam wynik,
- **efekt lawiny** - minimalna zmiana wejścia drastycznie zmienia skrót,
- **odporność na kolizje** - praktycznie niemożliwe znalezienie dwóch wiadomości o tym 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 18: Ekran funkcji SHA-256

Rysunek 18 przedstawia interfejs użytkownika funkcji SHA-256 w aplikacji CryptoLab Mobile.

16.9 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.

17 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.

17.1 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).

17.2 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

17.3 Kroki algorytmu

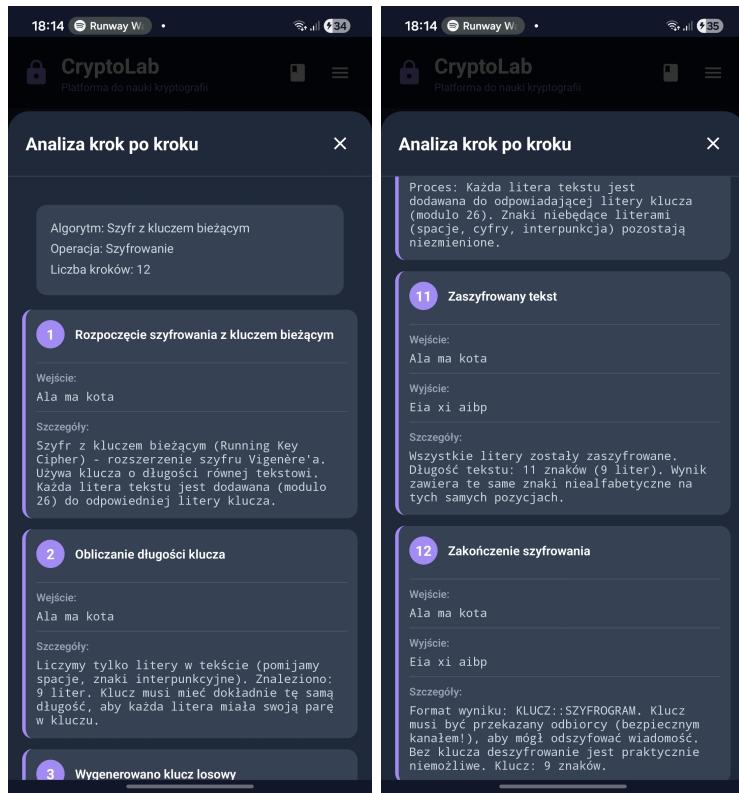
System loguje szczegółowe kroki wykonywania algorytmów (LogStep), co pozwala użytkownikom:

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



Rysunek 19: Ekran przeglądu logów

Rysunek 19 przedstawia ekran przeglądu logów w aplikacji CryptoLab Mobile.



Rysunek 20: Szczegółowe logi dla przykładowej operacji

Rysunek 20 przedstawia szczegółowe logi dla przykładowej operacji szyfrowania w aplikacji CryptoLab Mobile.

17.4 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

17.5 Statystyki

LogManager oblicza i udostępnia statystyki użycia:

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

17.6 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.

18 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 kryptografii asymetrycznej w aplikacji.
Generowanie par kluczy publiczny/prywatny z losowymi liczbami pierwszymi.
Implementacja algorytmu Euklidesa, rozszerzonego algorytmu Euklidesa i szybkiego potęgowania modularnego.
Dodanie kategorii "Kryptografia 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.

- **15.12.2025** Implementacja podpisu elektronicznego łączącego SHA-256 z RSA.

Proces podpisywania: obliczanie hash SHA-256 → podpisywanie kluczem prywatnym RSA → formatowanie wyniku.

Proces weryfikacji: obliczanie hash → sprawdzanie integralności → weryfikacja podpisu kluczem publicznym.

Rejestracja kroków podpisywania i weryfikacji w systemie logowania.