Dokumentacja projektu: Aplikacja do tworzenia i sprawdzania uproszczonego podpisu cyfrowego

Vasili Karol Magdalena Kopyt Karolina Kosmala
Jakub Kościelecki

19.06.2023

Spis treści

1 Wprowadzenie

Aplikacja do tworzenia i sprawdzania podpisu cyfrowego jest narzędziem, które umożliwia generowanie, weryfikację i zarządzanie podpisami cyfrowymi. Podpis cyfrowy to zaawansowana technika kryptograficzna, która służy do potwierdzania autentyczności i integralności danych elektronicznych.

W dzisiejszym cyfrowym świecie, gdzie transmisja i przechowywanie informacji odbywa się głównie w postaci elektronicznej, istnieje duże zapotrzebowanie na metody potwierdzania, że przesyłane i przechowywane dane nie zostały zmienione lub sfałszowane. Właśnie w tym celu służy podpis cyfrowy.

Podpis cyfrowy oparty jest na asymetrycznych algorytmach kryptograficznych, takich jak RSA, które wykorzystują parę kluczy: klucz prywatny i klucz publiczny. Klucz prywatny jest znany tylko właścicielowi, natomiast klucz publiczny jest udostępniany innym osobom. Podpis cyfrowy generowany jest za pomocą klucza prywatnego, a weryfikowany za pomocą klucza publicznego.

Aplikacja do tworzenia i sprawdzania podpisu cyfrowego umożliwia użytkownikom wygodne tworzenie podpisów cyfrowych dla swoich dokumentów elektronicznych oraz weryfikację autentyczności podpisów innych osób. Umożliwia również przechowywanie i zarządzanie kluczami kryptograficznymi, co pozwala na bezpieczne korzystanie z podpisów cyfrowych.

Celem tego projektu jest stworzenie intuicyjnego i łatwego w obsłudze narzędzia, które dostarczy użytkownikom niezawodny sposób na potwierdzenie autentyczności i integralności ich elektronicznych dokumentów. Aplikacja ma na celu zwiększenie zaufania do przesyłanych danych oraz ułatwienie procesu weryfikacji podpisów cyfrowych.

W dalszej części dokumentacji przedstawimy szczegóły implementacyjne, opiszemy wykorzystane algorytmy kryptograficzne oraz przedstawimy diagramy i obrazki ilustrujące działanie aplikacji.

2 Podpis cyfrowy

2.1 Funkcja skrótu MD4

MD4 jest funkcją skrótu, która jest wykorzystywana w kryptografii[1]. Funkcją skrótu (nazywaną także funkcją haszującą) nazywamy funkcję, która dowolnej liczbie naturalnej N przypisuje mniejszą od N liczbę naturalną. Taka funkcja znajduje swoje zastosowanie przy obliczaniu tzw. sumy kontrolnej, czyli liczby służącej do upewnienia się, czy dane nie uległy zmianie. Funkcja MD4 opjera się na konstrukcji Merkle'a-Damgårda, która w pewnym skrócie polega na:

1. Uzupełniamy wiadomość m w taki sposób, aby miała liczbę bitów podzielną przez pewną ustaloną wartość.
2. Dzielimy naszą uzupełnioną wiadomość na n bloków m_i o równej liczbie bitów, $i = 1, \ldots, n$.
3. Ustalamy stan początkowy na s_0.
4. Definiujemy nowy stan $s_i = f_k(m_i, s_{i-1})$ dla $i = 1, \ldots, n$ oraz f_k jest funkcją kompresji,
5. Zwracamy stan s_n.

Ten schemat jest charakterystyczny dla funkcji skrótu tj. MD4, MD5, SHA-1, SHA-2. W naszej aplikacji korzystamy z funkcji MD4, która jest zbudowana w następujący sposób:

[1] Obecnie funkcja MD4 jest uznawana za złamaną, więc nie powinna być używana w zastosowaniach kryptograficznych

1. Dzielimy wiadomość m na bloki o długości 512 bitów. Jeśli ilość bitów n naszej wiadomości m nie jest podzielna przez 512, to postępujemy w następujący sposób:

   (a) Doczepiamy do wiadomości jeden bit ustawiony na 1, tak aby miała długość $n+1$

   (b) Wstawiamy v bitów zerowych, gdzie v obliczamy według wzoru:

   $$v = 512 - ((n+65) \mod 512)$$

   (c) Następnie do wiadomości dodajemy liczbę n zapisaną jako 64-bitowa liczba nieujemna.

   Po wykonaniu tych kroków mamy pewność, że długość jest podzielna przez 512.

2. Ustawiamy stan początkowy. Stan w funkcji MD4 to cztery 32-bitowe liczby całkowite zapisane w kolejności little-endian[2], dając łacznie 128 bity. Stan początkowy to ciąg wartości $s_0 = (A, B, C, D)$, gdzie stałe liczbowe[3] to:

$$A = 67452301_{16} = 1732584193,$$

$$B = EFCDAB89_{16} = 4023233417,$$

$$C = 98BADCFE_{16} = 2562383102,$$

$$D = 10325475_{16} = 271733878.$$

3. Aby zdefiniować nowy stan s_i potrzebujemy funkcje kompresji, które są wykonywane w trzech rundach. Do opisu trzech rund wykorzystywanych w MD4 potrzebujemy trzech pomocniczych funkcji. Każda z nich przyjmuje trzy argumenty będące 32-bitowymi liczbamicałkowitymi nieujemnymi i w wyniku daje jedną 32-bitową liczbę całkowitą nieujemną.

$$F(x,y,z) = (x \wedge y) \vee ((\neg x) \wedge z),$$

$$G(x,y,z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z),$$

$$H(x,y,z) = x \Delta y \Delta z,$$

   gdzie operacje $\neg, \vee, \wedge, \Delta$ to odpowiednio negacja, suma, iloczyn oraz różnica symetryczna wykonana na bitach. Do zapisu metody potrzebujemy

[2] Jest to forma zapisu danych, w której najmniej znaczący bajt umieszczony jest jako pierwszy

[3] Wartości te pochodzą z oficjalnej dokumentacji MD4: `https://datatracker.ietf.org/doc/html/rfc1320` z punktu 3.3

szeregu stałych[4] podzielonych na trzy sposoby ich ustalania, nazywana rundami, po 16 kroków każda. Podczas wyliczania wartości s_i na podstawie s_{i-1}, wyciągamy z s_{i-1} wartości A, B, C oraz D. W każdym z 48 kroków na podstawie stanu opisanego przez zmienne (A, B, C, D) oraz 512 bitów wiadomości m_i ułożonej w szesnaście fragmentów po 32 bity, oznaczonych $X_1, ..., X_15$, wyliczana jest nowa wartość stanu (A', B', C', D') jako, kolejno dla $j = 0, \ldots, 47$:

$$A' = D$$

$$B' = (A + f_i(B, C, D) + X_{z_j} + y_j) << w_j$$

$$C' = B$$

$$D' = C$$

Operacja $a << b$ oznacza przesunięcie liczby a w zapisie binarnym o b miejsc w lewo z obrotem Po wykonaniu wszystkich kroków podstawiamy $s_i = s_{i-1} + (A, B, C, D)$, gdzie (A, B, C, D) to 48 razy zaktualizowany stan s_{i-1} według procedury opisanej powyżej.

2.2 Algorytm RSA

Algorytm RSA to jeden z najstarszych i najpopularniejszych algorytmów kryptograficznych używających klucza publicznego. Powodem tak dużej popularnosci RSA jest faktoryzacja dużych liczb pierwszych, która stanowi trudny matematyczny problem obliczeniowy. Algorytm RSA składa się z dwóch głównych etapów: generowania kluczy oraz szyfrowania i deszyfrowania wiadomości. Generowanie kluczy w algorytmie RSA:

1. Wybieramy dwie losowe liczby pierwsze p oraz q (przykładowo większe niż 2^{64})

2. Wybieramy liczbę naturalną e spełniającą równanie:

   $$\gcd(e, (p-1)(q-1)) = 1,$$

   gdzie gcd to największy wspólny dzielnik.[5]

3. Na podstawie wybranych liczb, wyliczamy $N = pq$ oraz publikujemy parę (N, e) jako nasz klucz publiczny.

4. Aby wyliczyć klucz prywatny, stosujemy rozszerzony algorytm Euklidesa do liczby e oraz $(p-1)(q-1)$, wyznaczając stałą d spełniającą równanie

   $$e \cdot d = 1 \bmod (p-1)(q-1)$$

[4] Stałe znajdują się w doumentacji MD4 w punkcjie 3.4

[5] Częstymi wyborami są liczby 3, 17, 65537. Należy pamiętać, że im większa liczba e, tym bezpieczniejsza jest wiadomość, ale ceną za to będzie czas, w którym algorytm będzie działać.

Ponieważ e oraz $(p1)(q1)$ są względnie pierwsze (tak zostało wybrane e), rozwiązanie opiera się na tożsamości Bézout'a. W szczególności musimy rozwiązać równanie diofantyczne liniowe

$$d \cdot e + c \cdot (p-1)(q-1) = 1,$$

ze względu na d i c.[6] Trójka liczb (d, p, q) stanowi klucz prywatny.

Jeśli posiadamy już klucz publiczny i prywatny, to możemy zacząć szyfrowanie i deszyfrowanie wiadomości. Załóżmy, że mamy klucz publiczny (N, e) oraz wiadomość m $(0 \leq m \leq N)$, którą chcemy zaszyfrować. Szyfrowanie wiadomości polega na zwróceniu zaszyfrowanej wiadomości c, która spełnia rownanie:

$$c = m^e \bmod N. \tag{1}$$

Odszyfrowanie wiadomości c przy pomocy klucza prywatnego (d, p, q) polega na rozwiązaniu równania:

$$m = c^d \bmod pq. \tag{2}$$

3 Implementacja

3.1 MD4

- ```
          def __init__(self, x):
          if not isinstance(x, bytes):
              raise TypeError(f"Expected <class 'bytes'>,
              but {type(x)} were given.\nTry to use
              .from_string(x) or .from_file(x) instead!")
          self.input_text = x    ##.encode()
          self.block_list = self.bit_blocks()
          self.ABCD_list = [1732584193, 4023233417, 2562383102, 271733878]
          self.ABCD_old = self.ABCD_list.copy()
          self.output_hash = self.get_hash()
  ```

  Metoda `__init__(self, x)` to konstruktor klasy MD4. Krótko opisuje jej funkcjonalność:

  Sprawdza, czy argument x jest poprawnym typem bytes. W przypadku niezgodności podnosi wyjątek TypeError. Przypisuje wartość argumentu x do pola *input_text*. Wywołuje metodę *bit_blocks*() i przypisuje jej wynik do pola *block_list*. Inicjalizuje pola *ABCD_list* i *ABCD_old* wartościami początkowymi. Wywołuje metodę *get_hash*() i przypisuje jej wynik do pola *output_hash*.

- ```
      @classmethod
      def from_string(cls, x):
          if not isinstance(x, str):
              raise TypeError(f"Expected <class 'str'>,
              but {type(x)} were given.")
          x = x.encode('utf-8')
          return cls(x)
  ```

[6]Rozwiązanie to istnieje i interesuje nas jedynie jedna ze współrzędnych rozwiązania, ponieważ druga zniknie, gdy stronami na powyższe równanie zastosujemy operację modulo.

Metoda *from_string* to klasna metoda klasy MD4, która tworzy nową instancję klasy na podstawie przekazanego tekstu, kodując go jako sekwencję bajtów w formacie UTF-8.

- ```
  @classmethod
  def from_file(cls, x):
      if not isinstance(x, str):
          raise TypeError(f"Expected <class 'str'>,
          but {type(x)} were given.")
      with open(x, 'rb') as file:
          x = file.read()
      return cls(x)
  ```

Metoda *from_file* to metoda klasy MD4, która tworzy nową instancję klasy na podstawie zawartości pliku o podanej ścieżce, odczytując go w trybie binarnym.

- ```
   def __str__(self):
      return hex(self.output_hash)[2:]
  def __eq__(self, hash_2):
      return hex(self.output_hash)[2:] == hash_2
  ```

Metoda `__str__` zwraca wartość skrótu w formacie szesnastkowym, a metoda `__eq__` porównuje wartość skrótu obiektu z innym haszem.

- ```
  def bit_blocks(self):
      text = self.input_text
      n = len(text) *8  ## do bit w

      text += b"\x80"
      text += b"\x00" * (64- ( len(text)+len(b"\x80")*8 )% 64 )
      text += struct.pack("<Q", n)


      block_list = [text[i : i + 64] for i in range(0,
      len(text), 64)]  ## 1 bajt == 8 bit w !!
      unpacked_blocks = [ struct.unpack("<16I", block)
      for block in block_list ]
      return unpacked_blocks
  ```

Metoda *bit_blocks* dzieli tekst wejściowy na bloki o długości 64 bajtów (512 bitów) i zwraca te bloki w postaci listy. Wcześniej dodaje również odpowiednie bity wypełnienia do tekstu, aby dopasować go do wielokrotności 512 bitów. Każdy blok jest następnie konwertowany na listę 16 wartości całkowitych z rozpakowanymi wartościami 32-bitowych słów.

- ```
  def get_hash(self):
      ## Runda 1
      z_1_list = list( range(16) )
      w_1_list = [3, 7, 11, 19]*4


      for unpacked_block in self.block_list:
          for z, w in zip(z_1_list, w_1_list):
  ```

```
            A_prim = self.ABCD_list[3]
            new_X = self.ABCD_list[0] +
            self.F( self.ABCD_list[1], self.ABCD_list[2],
            self.ABCD_list[3] )+ unpacked_block[z]
            B_prim = self.rotate_left( new_X & 0xFFFFFFFF, w )


            C_prim = self.ABCD_list[1]
            D_prim = self.ABCD_list[2]
            self.ABCD_list = [A_prim, B_prim, C_prim, D_prim]
```

Metoda *get_hash* oblicza skrót MD4 dla bloków tekstu wejściowego. Wykorzystuje trzy rundy operacji na blokach danych. W pierwszej rundzie obliczane są nowe wartości dla zmiennych A, B, C i D na podstawie wartości z poprzedniej iteracji, funkcji F oraz danych z bloków. Wartości te są przekształcane poprzez obrót w lewo i zapisywane z powrotem do zmiennych *ABCD_list*, które reprezentują stan algorytmu MD4.

- ```
  ## Runda 2
  y_2 = 1518500249
  z_2_list = [ i for i_start in range(4)
  for i in range(i_start, 16, 4) ]
  w_2_list = [3, 5, 9, 13]*4


  for unpacked_block in self.block_list:
      for z, w in zip(z_2_list, w_2_list):
          A_prim = self.ABCD_list[3]
          new_X = self.ABCD_list[0]
          + self.G( self.ABCD_list[1],
          self.ABCD_list[2], self.ABCD_list[3] )
          + unpacked_block[z] + y_2
          B_prim = self.rotate_left( new_X & 0xFFFFFFFF, w )
          C_prim = self.ABCD_list[1]
          D_prim = self.ABCD_list[2]
          self.ABCD_list = [A_prim, B_prim, C_prim, D_prim]
  ```

  W drugiej rundzie operacji na blokach danych, metoda *get_hash* oblicza nowe wartości dla zmiennych A, B, C i D na podstawie wartości z poprzedniej rundy, funkcji G, stałej y_2 oraz danych z bloków.

- ```
  ## Runda 3
  y_3 = 1859775393
  z_3_list = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]
  w_3_list = [3, 9, 11, 15]*4

  for unpacked_block in self.block_list:
      for z, w in zip(z_3_list, w_3_list):
          A_prim = self.ABCD_list[3]
          new_X = self.ABCD_list[0] +
          self.H( self.ABCD_list[1], self.ABCD_list[2],
          self.ABCD_list[3] ) + unpacked_block[z] + y_3
          B_prim = self.rotate_left( new_X & 0xFFFFFFFF, w )
          C_prim = self.ABCD_list[1]
          D_prim = self.ABCD_list[2]
  ```

```
            self.ABCD_list = [A_prim, B_prim, C_prim, D_prim]
    ABCD_out = [ (x_list + x_old)
    & 0xFFFFFFFF for x_list, x_old in
    zip(self.ABCD_list, self.ABCD_old) ]


    hash = struct.pack("<4L", *ABCD_out)
    hash = int.from_bytes(hash, 'big')
    return hash
```

W trzeciej rundzie operacji na blokach danych, metoda *get_hash* oblicza nowe wartości dla zmiennych A, B, C i D na podstawie wartości z poprzedniej rundy, funkcji H, stałej y_3 oraz danych z bloków. Wartości te są przekształcane poprzez obrót w lewo i zapisywane z powrotem do zmiennych $ABCD\_list$. Na koniec, wartości $ABCD\_list$ są sumowane z odpowiadającymi wartościami z poprzedniego stanu ($ABCD\_old$) i wynik jest pakowany w strukturę bajtową, a następnie zamieniany na liczbę całkowitą i zwracany jako wynik funkcji skrótu MD4.

- ```
      @staticmethod
  def F(x, y, z):
      return (x & y) | (~x & z)
      @staticmethod
  def G(x, y, z):
      return (x & y) | (x & z) | (y & z)
      @staticmethod
  def H(x, y, z):
      return x ^ y ^ z
  ```

  Definiowanie funkcji kompresji F, G, H.

- ```
      @staticmethod
  def rotate_left(value, shift):
      return ((value << shift) | (value >> (32 - shift))) & 0xFFFFFFFF
  ```

  Definicja przesunięcia bitowego w lewo.

3.2 RSA

- ```
  def __init__(self):
  self.p, self.q = self.diff_primes()
  self.N = self.p * self.q
  self.gcd = self.extended_euclidean_algorithm(65537,
  (self.p - 1) * (self.q - 1))
  self.pub = (self.N, 65537)
  self.prv = self.solve_diofantine(65537,
  (self.p - 1) * (self.q - 1))
  ```

  __init__(self) jest konstruktorem klasy RSA. W tej metodzie: self.p i self.q są ustawiane na różne liczby pierwsze poprzez wywołanie metody $diff\_primes()$. self.N jest obliczane jako iloczyn self.p i self.q. self.gcd jest ustawiane na wynik rozszerzonego algorytmu Euklidesa dla wartości
  ```

65537 i $(self.p-1)*(self.q-1)$. self.pub jest ustawiane jako krotka zawierająca wartość self.N i 65537, tworząc klucz publiczny. self.prv jest ustawiane poprzez rozwiązanie równania diofantycznego dla wartości 65537 i $(self.p-1)*(self.q-1)$, tworząc klucz prywatny.

- ```
      def prime(self, min):
          prime = sympy.randprime(min, 10 * min)
          return prime
  ```

  Metoda prime(self, min) generuje losową liczbę pierwszą w zakresie od min do 10 * min. Wykorzystuje funkcję randprime() z biblioteki sympy.

- ```
  def diff_primes(self):
          min = 2 ** 64
          p = self.prime(min)
          q = self.prime(min)

          while p == q:
              q = self.prime(min)

          return p, q
  ```

  Metoda $diff\_primes(self)$ generuje dwie różne liczby pierwsze p i q, wykorzystując metodę prime(self, min). Zapewnia, że p i q są różne.

- ```
      def extended_euclidean_algorithm(self, a, b):
          if b == 0:
              return a, 1, 0
          gcd, x1, y1 = self.extended_euclidean_algorithm(b, a % b)
          x = y1
          y = x1 - (a // b) * y1
          return gcd, x, y
  ```

  Metoda $extended\_euclidean\_algorithm(self, a, b)$ implementuje algorytm Euklidesa, który oblicza największy wspólny dzielnik (gcd) dwóch liczb oraz współczynniki x i y.

- ```
  def solve_diofantine(self, a, b):
          gcd, x, y = self.extended_euclidean_algorithm(a, b)

          if gcd == 1:
              return x, self.p, self.q
          else:
              return None
  ```

  Metoda $solve\_diofantine(self, a, b)$ rozwiązuje równanie diofantyczne dla a i b za pomocą metody $extended\_euclidean\_algorithm(self, a, b)$. Jeśli największy wspólny dzielnik (gcd) wynosi 1, zwraca wartość (x, self.p, self.q), a w przeciwnym razie zwraca None.

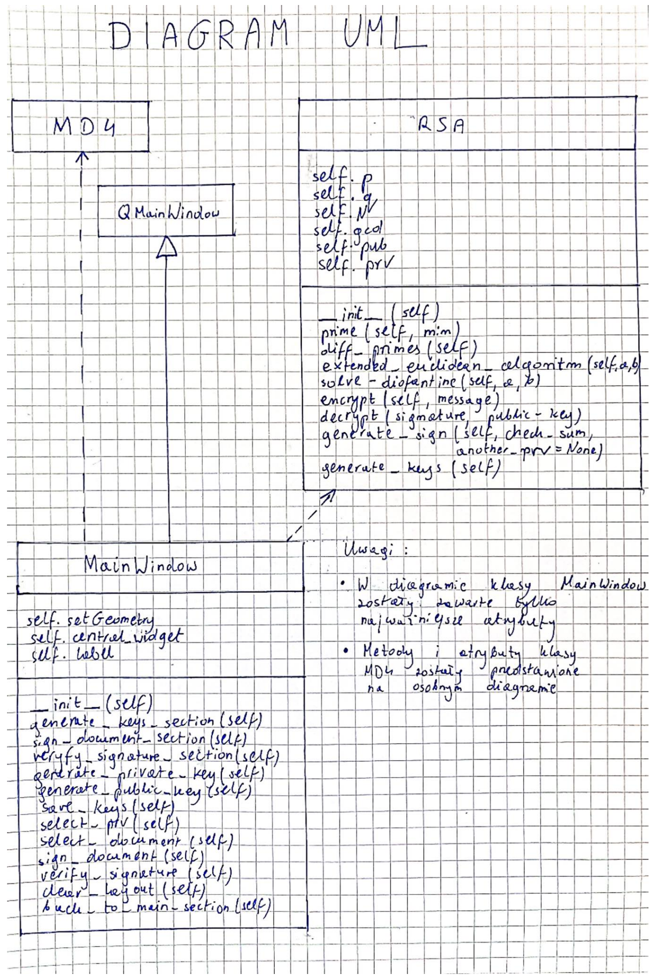
- ```
  def encrypt(self, message):
          c = (message ** self.pub[1]) % self.N
          return c
  ```
  ```

Metoda encrypt(self, message) szyfruje wiadomość z kluczem publicznym (self.pub[1]) i wartością self.N. Zwraca zaszyfrowaną wiadomość c.

- ```
  def decrypt(self, ciphertext):
          original_message = pow(ciphertext,
          self.prv[0], self.prv[1] * self.prv[2])
          return original_message
  ```

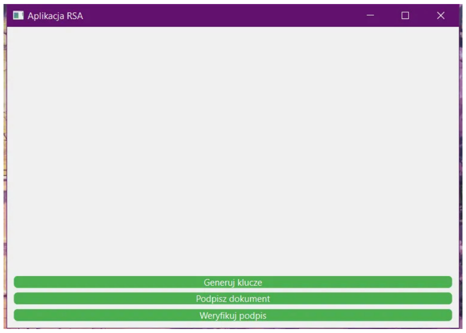
  Metoda decrypt(self, ciphertext) deszyfruje zaszyfrowaną wiadomość za pomocą operacji potęgowania modulo (pow()) z kluczem prywatnym (self.prv[0]) i wartością self.prv[1] * self.prv[2]. Zwraca oryginalną odszyfrowaną wiadomość.
  ```

4 Diagram UML



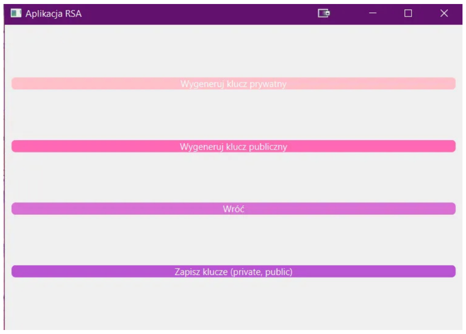
5 Instrukcja obsługi

Poniżej znajduje się prosta instrukcja obsługi wraz ze zdjęciami interfejsu.



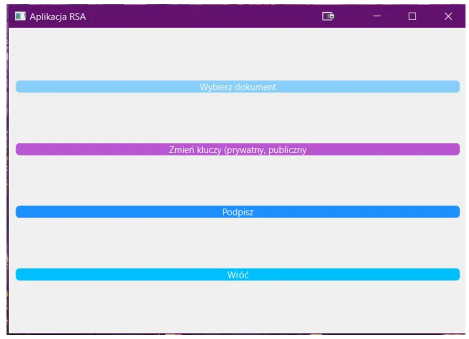
Rysunek 1: Ekran główny aplikacji

- Po odpaleniu aplikacji pojawia się ekran główny naszej aplikacji. Znajdują się na nim trzy przyciski: "Generuj klucze", "Podpisz dokument"oraz "Weryfikuj podpis". Po wciśnięciu przycisku "Generuj klucze"aplikacja przenosi się do ekranu generowania kluczy.



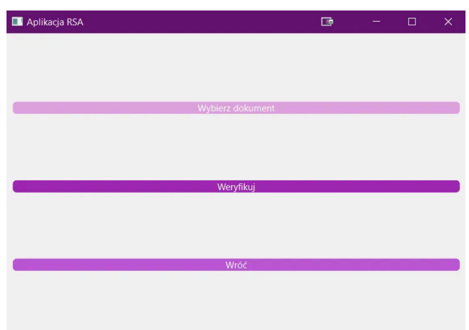
Rysunek 2: Ekran generowania kluczy

- Ekran generowania kluczy składa się z czterech przycisków: "Wygeneruj klucz prywatny", "Wygeneruj klucz publiczny", "Wróć"oraz "Zapisz klucze (private,public)". Pierwsze dwa przyciski odpowiadają za generowanie kluczy prywatnych i publicznego. Trzeci przycisk służy do powrotu do ekranu głównego. Czwarty przycisk służy do zapisania kluczy: prywatnego i publicznego. Po powrocie do ekranu głównego możemy skorzystać z przycisku "Podpisz dokument".



Rysunek 3: Ekran podpisywania dokumentu

- Po kliknięciu tego przycisku przenosimy się do ekranu podpisywania dokumentu. Znajdują się tam cztery przyciski: "Wybierz dokument", "Zmień klucze (prywatny,publiczny)", "Podpisz", "Wróć". Po kliknięciu pierwszego przycisku, pojawia się okno do wyboru dokumentu, który chcemy podpisać. Drugi przycisk służy do zmiany pary kluczy prywatny-publiczny. Trzeci przycisk umożliwia podpisać wybrany dokument. Po kliknieciu na czwarty przycisk, aplikacja cofa nas do ekranu głownego, w którym można wybrać trzeci przycisk: "Weryfikuj podpis".



Rysunek 4: Ekran weryfikacji podpisu

- Po kliknięciu "Weryfikuj podpis", zostajemy przeniesieni do ekranu weryfikacji podpisu. Znajdują się tam trzy przyciski: "Wybierz dokument", "Weryfikuj"oraz "Wróć". Pierwszy przycisk służy do wyboru dokumentu, któremy chcemy zweryfikować podpis. Drugi przycisk pozwala nam zweryfikować podpis dokumentu. Trzeci przycisk służy do wyjścia z ekranu weryfikacji podpisu do ekranu głównego. Aby wyjść z aplikacji, należy kliknąć przycisk X, który znajduje się po prawej stronie.

6 Bibliografia

Literatura

[1] dr inż. Andrzej Giniewicz, Wprowadzenie do kryptograficznych funkcji skrótu, 2023

[2] dr inż. Andrzej Giniewicz, Wprowadzenie do algorytmu RSA, szyfrowanie i podpis cyfrowy, 2023.