Implementacja uproszczonego mechanizmu Kyber

Kacper Müller

Opis Kyber

Crystals-Kyber to mechanizm enkapsulacji klucza (KEM), odporny na ataki przy użyciu komputerów kwantowych. Korzysta z uczenia z błędami: Module-LWE (MLWE), i krat. NIST uznał ten mechanizm za standard i został on oznaczony: FIPS 203, Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM). Opisany poniżej algorytm nie jest pełną implementacją FIPS 203, lecz korzysta z uproszczonego schematu. Przykładowo: pomija mechanizmy CCA i KDF.

Generacja kluczy

Zaimplementowany algorytm korzysta z macierzy złożonych z elementów pierścienia wielomianowego $Z_q [x] / (x^n + 1)$. Wszystkie operacje zatem wykonywane są modulo q.

Generacja kluczy polega na wylosowaniu kwadratowej macierzy **A**, a także wektorów **s** oraz **e** o niskich współczynnikach. Następnie oblicza wektor **t** na podstawie wzoru

$$t = As + e$$

Para **A**, **t** to klucz publiczny. Wektor **s** to klucz prywatny.

W rzeczywistej implementacji nie jest przesyłana cała macierz **A**, lecz jedynie ziarno do jej generacji, aby zmniejszyć ilość potrzebnej pamięci.

Szyfrowanie

Chcąc zaszyfrować wiadomość należy kolejne bity zamienić na współczynniki wielomianu i przemnożyć je przez ⌊q/2⌋, otrzymując $\mathbf{message_{poly}}$. Jeżeli liczba bitów wiadomości jest mniejsza od stopnia wielomianu, stosuje się uzupełnienie zerami. Liczba bitów nie może być większa niż stopień wielomianu.

Następnie należy wylosować wektory, również o niskich współczynnikach wielomianów: **r**, $\mathbf{e_1}$ oraz $\mathbf{e_2}$ i obliczyć wektory **u**, **v** według wzorów

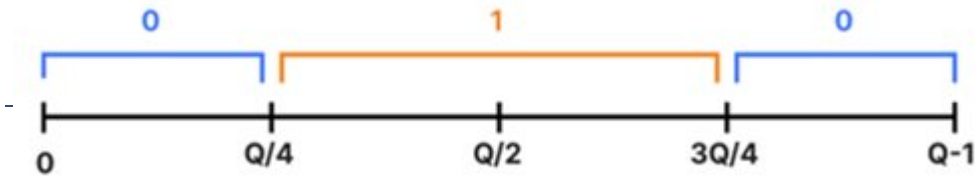
$$u = A^T r + e_1$$
$$v = t^T r + e_2 + message_{poly}$$

Odszyfrowywanie

Aby odszyfrować wiadomość należy skorzystać ze wzoru

$$message_{poly} = v - (s^T u)$$

a następnie odwrócić współczynniki wielomianu z powrotem na bity według rysunku 1. Współczynniki wielomianu to bity oryginalnej wiadomości.



Obraz 1. Wizualizacja zamiany współczynników wielomianu na bity.

Standardowe parametry ML-KEM-768 (dostępne są również wersje 512 i 1024) to:

- stopień wielomianu n: 256
- modulo q: 3329
- rozmiar macierzy k: 3
- rozrzut współczynników poszczególnych błędów: 2, 2, 10, 4

Parametry można dostosować, w zależności od oczekiwanego stopnia bezpieczeństwa.

Implementacja

Macierze

Główny problem stanowiła implementacja klasy macierzy wielomianowych. Implementacja znajduje się w pliku `polynomial_algebra.py`.

Aby stworzyć macierz z losowymi współczynnikami wielomianów, korzystamy z konstruktora:
`matrix = PolynomialMatrix(q: int, n_rows: int, n_cols: int, max_coefficient_value: int, polynomial degree: int, include_negative: bool)`

Parametry to kolejno:

- q - wartość modulo. Wszystkie operacje będą wykonywane modulo q
- n_ rows, n_cols - liczba wierszy i kolumn tworzonej macierzy. Dla wektorów n_cols ustawiamy jako 1
- max_coefficient_value - rozrzut wartości współczynników wielomianów. Przykładowo, ustalając wartość “3” współczynniki będą z przedziału [0, 3]. W przypadku argumentu `include_negative` jako True, współczynniki będą z przedziału [-3, 3].
- include_negative - jeżeli `True` - współczynniki wielomianów będą losowane z przedziału [-`max_coeff_val`, `max_coef_val`], w przeciwnym wypadku [0, `max_coef_val`]

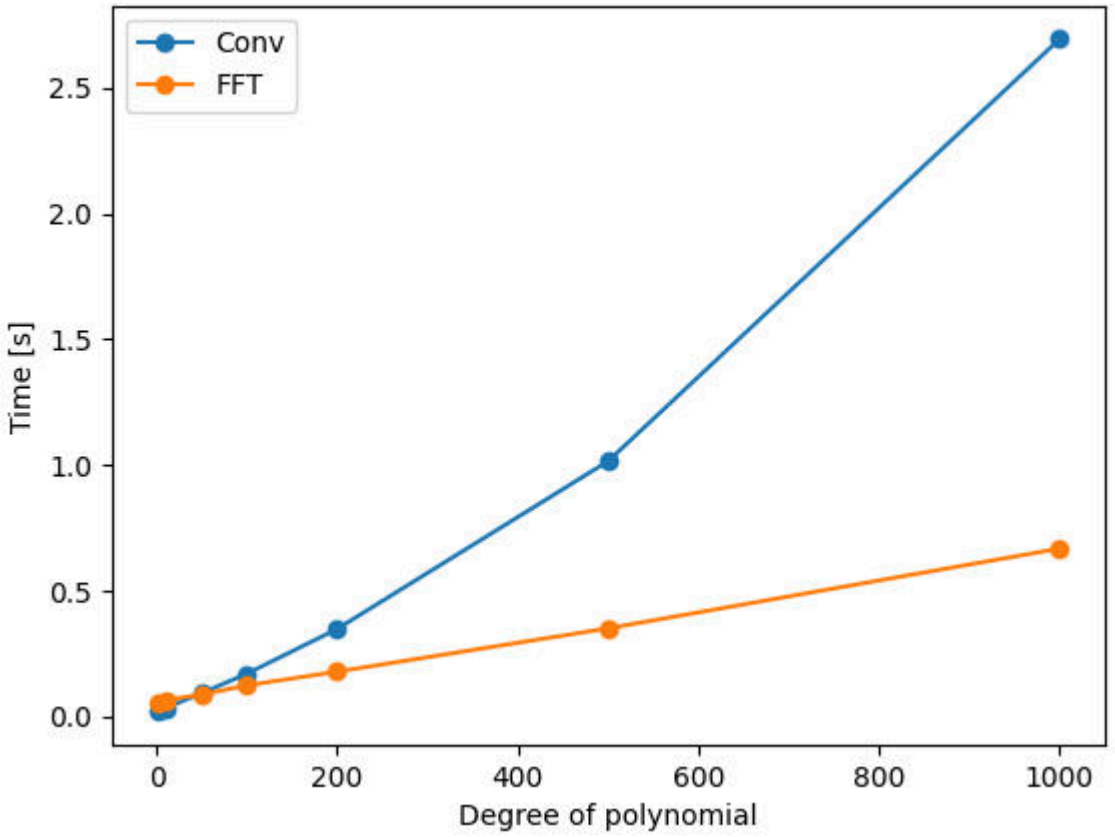
Zostały również zaimplementowane metody:

- __repr__ - umożliwia wyświetlenie macierzy i jej wielomianów w metodzie print()
- copy - kopiuje strukturę macierzy, lecz z współczynnikami wielomianów ustawionymi na 0
- __add__ - umożliwia dodawanie macierzy za pomocą operatora `+`. Przed operacją sprawdzane są wymiary macierzy.
- __sub__ - operacja analogiczna do __add__, lecz dla operatora `-`
- T (property) - zwraca transponowaną macierz, przykład użycia: `matrix_transposed = matrix.T`
- __mul__ - umożliwia mnożenie macierzy za pomocą operatora `*`. Przed operacją sprawdzane są wymiary macierzy.

Domyślnie mnożenie wykorzystuje szybką transformatę Fouriera (FFT). Może to być jednak zmienione na konwolucję, ustawiając zmienną `USE_DFT_INSTEAD_OF_CONVOLUTION` na `False`. W rzeczywistym ML-KEM Kyber wykrzystuje się NTT (Number Theoretic Transform).

Implementacja mnożenia z wykorzystaniem FFT znajduje się w pliku `polynomial_multiplication_fft.py`.

Została również zbadana różnica pomiędzy szybkością mnożenia z wykorzystaniem konwolucji, a FFT. Wykorzystane do tego zostały funkcje `check_times()` oraz `plot_times()` w `main.py`. Na wykresie 1 można łatwo zauważyć różnicę pomiędzy teoretyczną złożonością $O(n^2)$ konwolucji, a $O(n \cdot \log n)$ FFT.



Wykres 1. Porównanie czasu potrzebnego do wykonania mnożenia wielomianów o danych wielkościach, przez metody: konwolucja, FFT.

Operacje były testowane pod względem prawidłowości działań na małych macierzach i wielomianach.

Kyber

Po poprawnym zaimplementowaniu macierzy, implementacja mechanizmu Kyber nie jest skomplikowana. Znajduje się ona w pliku `kyber.py`.

Korzystanie z mechanizmu zaczynamy od stworzenia obiektu klasy KyberPKE: `kyber = KyberPKE(n: int = 256, q: int = 3329, k: int = 3, eta1: int = 2, eta2: int = 2)`.
Domyślne parametry odpowiadają parametrom ML-KEM-768.

Po stworzeniu obiektu możemy korzystać z jego metod:

- generate_keys() - zwraca klucz publiczny i prywatny: macierz i wektor **(A, t)** oraz wektor **s**. Przykładowe użycie: `public_key, private_key = kyber.generate_keys()`
- encrypt_message(message: str, public_key: tuple) - zwraca zaszyfrowaną wiadomość: wektory **u** i **v**. Przykładowe użycie: `enc_message = kyber.encrypt_message('password', public_key)`, z możliwością rozbicia: `u, v = enc_message`
- decrypt_message(enc_message: tuple, private_key: PolynomialMatrix) - zwraca string z odszyfrowaną wiadomością. Przykładowe użycie:
  `message = kyber.decrypt_message(enc_message, private_key)`

W klasie znajdują się również prywatne metody pomocnicze:

- _transform_message_to_polynomial(message: string, scaling_factor: int) - zamienia wiadomość na wielomian, przemnażając współczynniki przez zadaną wartość. W przypadku Kyber jest to wartość ⌊q/2⌋ (patrz: obraz 1). Jeżeli wiadomość jest za krótka, stosowany jest padding.
- _transform_message_to_binary(message: str) - zamienia znaki wiadomości na bity 0/1 - jako string. Stosowana jest w metodzie opisanej powyżej.
- _transform_polynomial_to_binary_string(poly_message: PolynomialMatrix) - metoda odwrotna do _transform_message_to_polynomial. Zamienia wielomian na string bitów, korzystając z zamiany opisanej na obrazie 1.
- _center_round(x: int) - zamienia wartość x na 0/1, zgodnie z obrazem 1.
- _transform_binary_string_to_text(binary_string: str) - zamienia string bitów na string tekstu czytelnego dla człowieka.

Przykład działania opisany w rozdziale “Demonstracja działania”.

Demonstracja działania

Przesłanie wiadomości

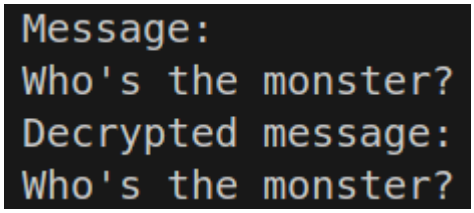
Demonstracja została przeprowadzona ze standardowymi dla ML-KEM-768 parametrami.

```
    kyber = KyberPKE()
    public_key, private_key = kyber.generate_keys()

    # Bob
    message = "Who's the monster?"
    print('Message:')
    print(message)
    encrypted_message = kyber.encrypt_message(message, public_key)

    # Alice
    decrypted_message = kyber.decrypt_message(encrypted_message, private_key)
    print('Decrypted message:')
    print(decrypted_message)
```

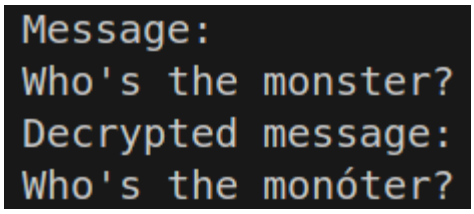
Program prawidłowo odszyfrowuje wiadomość



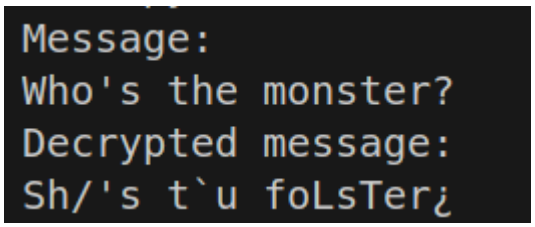
Badanie większych błędów

Ciekawym zjawiskiem do zbadania jest to jak zwiększanie rozrzutu współczynników wielomianów w wektorach błędów zmienia odszyfrowaną wiadomość.

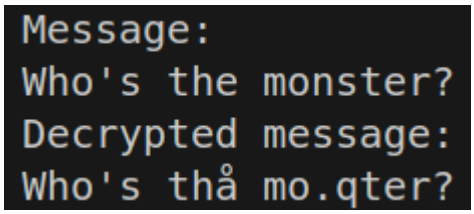
Przy zmianie jednego z błędów, dopiero przy wartościach około 15 zaczynają pojawiać się błędy:



Dla wartości 25 wiadomość jest kompletnie zmieniona:



Przy zmianie obu błędów jednak, już od wartości około 6 wiadomość staje się ciężka do zrozumienia:



Od wartości 10 dla obu błędów wiadomość jest zupełnie inna: