Metody kryptografii w analizie danych - Projekt

Kinga Sąkól, Michał Szczurek

19 czerwca 2023

1 Wstęp

W ramach projektu zaimplementowano, przetestowano i opisano 2 algorytmy kryptografii postkwantowej- Gravity SPHINCS i LEDAkem. Kod dostępny jest na `https://github.com/kingasakol/Post-quantum-cryptography`.

2 Gravity SPINCS

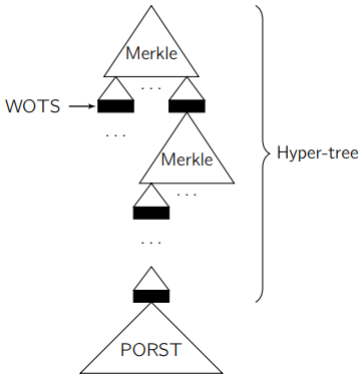
2.1 Cel algorytmu

Algorytm jest rozwinięciem algorytmu SPHINCS i umożliwia generowanie par kluczy (publiczny, prywatny), podpisywanie nimi wiadomości oraz weryfikację podpisanych wiadomości.

2.2 Wysokopoziomowy opis algorytmu

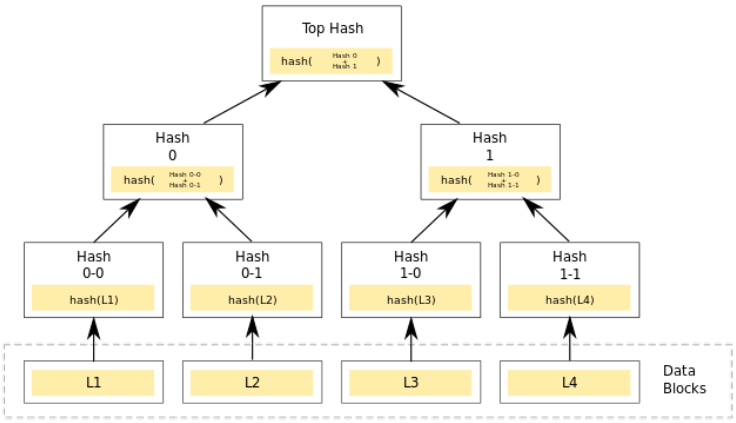
Schemat działania algorytmu jest bardzo zbliżony do działania oryginalnego SPHINCSa i różni się głównie drobnymi usprawnieniami, lub wymianą pewnych 'klocków' z których zbudowany był pierwowzór. Konstrukcję algorytmu można przedstawić jako hiperdzrzewo składające się z 3 rodzajów drzew, którego korzeniem jest klucz publiczny. W jego skład wchodzą następujące poddrzewa:

- Drzewa Merklego, których liśćmi są korzenie drzewa kompresji WOTS
- Drzewa kompresji WOTS, których liście są komponentami klucza publicznego do podpisu Winternitza. są podobne do drzew Merklego, ale nie muszą być kompletnymi drzewami binarnymi
- Drezewa kompresji PORST (Pseudo random number generator to Obtain Random Subset Tree) (w oryginalnym algorytmie SPHINCS - HORST(Hash to Obtain a Random Subset Tree)) - drzewa Merklego, których korzeń jest reprezentacją klucza publicznego schematu PORS



Rysunek 1: Gravity SPHINCS,
źródło: Supporting doccumentaion Gravity SPHINCS

Drzewa Merklego, to schemat używający dowolnej funkcji haszującej, o stałym rozmiarze hasza. Polega a tym, że wartość poszczególnego węzła jest haszem z konkatenacji dzieci węzła w drzewie.



Rysunek 2: Drzewo Merkle,*źródło: Wikipedia*

2.3 Krótki opis działania algorytmu SPHINCS

Gravity SPHINCS bardzo mocno bazuje na oryginalnym SPHINCS, wobec czego warto bardo wysokopoziomowo go poznać przed przystąpieniem do analizy Gravity SPHINCS.

Podpisywanie wiadomości w algorytmie SPHINCS polega na następujących krokach:

- W oparciu o wiadomość i klucz prywatny wylicza się indeks liścia. Liściem jest instancja drzewa HORST, która będzie użyta do podpisu wiadomości (wszystkich takich instancji jest 2^h, gdzie h - wysokość hiperdrzewa).
- Następnie generuje się instancję HORST z seedem uzyskanym w oparciu o klucz prywatny i indeks liścia. przy pomocy instancji HORST podpisuje się wiadomość - podpisem jest k kluczy i odpowiadających im ścieżek autentykacji. Klucze i ścieżki są częścią podpisu SPHINCS. Aby przejść do następnego kroku trzeba jeszcze obliczyć skompresowany klucz publiczny HORST - p.
- Dla każdej warstwy w drzewie podpisuje się klucz p uzyskany jako wynik z poprzedniej warstwy przy pomocy odpowiednich WOTS (obliczonych dzięki indeksowi). Podpis i ścieżkę autentykacji (w obrębie danej instacji WOTS) dodaje się do podpisu SPHINCS. Następnie do podpisu dodaje się jeszcze ścieżkę autentykacji obliczoną względem drzewa Merklego powiązanego z danym WOTS.

2.4 Usprawnienia względem oryginalnego SPHINCSa

- użycie PORS - bezpieczniejszego wariantu HORS z oryginalnego algorytmu
- cache'owanie wyników częściowych
- podpisy batchy wiadomości pozwalające zmniejszyć czas podpisywania i rozmiar podpisu
- użycie Mask-less hashing - oryginalny algorytm SPHINCS używał bitowych masek, które wchodziły w skład klucza publicznego. Maski były XORowane z konkatenacją węzłów przed ich zahaszowaniem Twórcy Gravity SPHINCS przedstawili dowód, że nie jest to potrzebne dzięki czemu uprościli schemat i zmniejszyli rozmiar klucza.
- Octopus autchentication pozwalający na zmniejszenie rozmiaru podpisu przez usunięcie redundancji

2.5 Prymitywy

Algorytm wykorzystuje 2 wymienialne komponenty:

- Funkcje haszujące: zmodyfikowane wersje algorytmów haszujących Haraka-v2-256 i Haraka-v2-512 wzbogacone o 6 nowych rund oraz SHA-256.
- AES w trybie counter, używany jako funkcja pseudolosowa

Za wyborem tych komponentów przemawia aspekt optymalizacyjny: Haraka była zaprojektowana, by wspierać obliczenia równoległe na procesorze, co pozwala znacząco przyspieszyć cały algorytm. Odpowiednie wywoływanie funkcji haszującej dla niezależnych łańcuchów w drzewie pozwala zrównoleglić i przyspieszyć Gravity SPHINCS. AES został wybrany ze względu na zestaw instrukcji zintegrowany z procesorami (AES-NI), co również przyspiesza tą, często wykonywaną w algorytmie, operację.

2.6 Zalety i wady algorytmu

Zalety:

- wysoki poziom zapewnienia bezpieczeństwa - bezpieczeństwo w algorytmie jest oparte o odporności na kolizję funkcji haszujących
- możliwość zmiany parametrów, pozwalająca na przyspieszenie algorytmu kosztem zwiększenia rozmiaru podpisu i na odwrót
- podpis batchy wiadomości

Wady:

- poziom skomplikowania algorytmu
- rozmiar podpisu (20-30 KiB)
- długi czas podpisu wiadomości - testy wydajnościowe twórców wskazywały, że w zależności o parametrów algorytmu może to być od 0.39s do 5.89s. Jest to ogromna wartość biorąc pod uwagę, że dotyczy optymalnej implementacji w C.

2.7 Algorytmy składowe programu

Wykorzystane algorytmy wewnętrzne są dość wymagające konceptualnie, a twórcy Gravity SPHINCS opisali je poprzez pseudokod, wobec czego zbudowanie intuicji odnośnie działania programu nie jest proste. Poniżej znajduje się ich krótki opis:

- Operacje na adresach - każde poddrzewo posiada unikalny adres, dzięki czemu można szybko poznać generowany przez nie hasz. W algorytmie wykorzystuje się funkcje generujące adres dla poddrzewa i inkrementujące go.
- L-drzewo - jest implementacją drzewa podobnego do drzewa Merklego, która pozwala na wyliczenie hasza korzenia w oparciu o liście. Różni się od drzewz Merklego tym, że nie musi być pełnym drzewm binarnym.
- Suma kontrolna Winternitza - na wejściu przyjmuje hasz, a na wyjściu zwraca l liczb całkowitych, gdzie l jest parametrem algorytmu.

- Generowanie kluczy publicznych Winternitza - funkcja przyjmuje na wejściu sekretny seed i adres drzewa, a zwraca klucz publiczny Winternitza przy pomocy wielokrotnego użycia Haraka-v2-256.
- Podpis Winternitza - funkcja przyjmuje na wejściu sekretny seed, adres drzewa i hasz, oraz zwraca podpis hasza wykorzystując do tego wielokrotnie funkcję haszującą Haraka-v2-256 i sumę kontrolną Winternitza.
- Ekstrakcja klucza publicznego Winternitza - funkcja na wejściu przyjmuje hasz oraz podpis Winternitza i zwraca klucz publiczny wykorzystując do tego wielokrotnie funkcję haszującą Haraka-v2-256 i sumę kontrolną Winternitza oraz mechanizm L-drzewa.
- Wyliczenie Korzenia drzewa Merklego - wylicza wartość w korzeniu w oparciu o wartości liści.
- Autentykacja drzewa Merklego - przyjmuje na wejściu liście drzewa oraz indeks konkretnego liścia. Dla tego liścia zwracana jest ścieżka autentykacji zdefiniowana w oparciu o obliczenia w drzewie Merklego.
- Ekstrakcja korzenia drzewa Merklego - przyjmuje na wejściu liście drzewa, indeks, oraz ścieżkę autentykacji. Na wyjściu zwraca korzeń drzewa
- Autentykacja Octopus - jest to zbiorowa procedura autentykacji drzewa Merklego wykonana dla wielu liści na raz, usuwająca zbędne węzły autentykacji. Na wejściu przyjmuje liście, indeksy liści posortowane rosnąco i zwraca zbiorową ścieżkę autentykacji.
- Ekstrakcja korzenia Octopus - zwraca korzeń w oparciu o liście, ich indeksy oraz ścieżkę autentykacji. Procedura jest dość zwiła.
- PORS - przyjmuje na wejściu hasz oraz sól i zwraca losowe indeksy w hiperdrzewie.
- Podpis PORST - przyjmuje na wejściu sekretny seed, adres bazowy oraz posortowane indeksy liści i zwraca podpis generowany w oparciu o liczbę pseudolosowe i autentykację Octopus.
- Ekstrakcja klucza publicznego PORST - w oparciu o indeksy i podpis PORST zwraca klucz publiczny używając do tego ekstrakcji korzenia Octopus.

2.8 Schemat działania algorytmu

Używając algorytmów składowych można wykonać następujące operacje Gravity SPHINCS:

2.8.1 Generacja kluczy

Generacja kluczy polega na:

- wylosowaniu seeda i soli
- wygenerowaniu wielu kluczy publicznych Winternitza
- obliczeniu korzenia drzewa Merklego zbudowanego z tychże kluczy

2.8.2 Podpis

Podpis polega na

- generacji publicznej soli haszując sól prywatną
- wygenerowaniu losowych indeksów liści przy pomocy PORS
- wygenerowaniu podpisu PORST i powiązanego klucza publicznego
- obliczaniu kolejnych wartości kluczy generowanych przez WOTS i ścieżek autentykacji drzew Merklego
- Kluczem jest (sól publiczna, publiczna część podpisu Octopus, ścieżki autentykacji Merklego)

Weryfikacja podpisu polega na

- wygenerowaniu losowych indeksów liści przy pomocy PORS i publicznej soli
- wyekstraktowaniu klucza publicznego PORST
- wyekstraktowaniu kolejnych korzeni drzew WOTS i Merkle w oparciu i ścieżkę autentykacji
- porównaniu korzenia ostatniego drzewa Merklego i klucza publicznego

2.8.3 Funkcje batchowe

Istnieją jeszcze warianty batchowe podpisu i weryfikacji, które są lekko zmodyfikowaną wersją algorytmów opisanych powyżej, pozwalającą na pospisanie i weryfikację wielu wiadomości przy pomocy 1 hiperdrzewa.

2.9 Technologia

Do napisania algorytmu użyto języka Python. Dodatkowo wykorzystano biblioteki:

- cryptography - zawierającej implementację algorytmu AES
- hashlib - zawierającej implementację sha256

- secrets - zawierającej implementację zawierającej bezpieczny generator liczb pseudolosowych, używany do generowania soli i seeda

Zaimplementowany algorytm działa poprawnie, co zweryfikowano testami jednostkowymi. Niemniej jednak program działa znacznie wolniej niż ma to miejsce w oryginalnym algorytmie napisanym w języku C. Wynika to z powolnego charakteru języka Python, silnej typizacji, która wymaga konwersji typów w trakcie działania algorytmu (w C bajty można interpretować jako dowolny obiekt, o ile mają one odpowiednią strukturę) oraz braku optymalizacji specyficznych dla C.

3 LEDAkem

3.1 Cel algorytmu

LEDAkem jest modułem należącym do grupy KEM (Key Encapsulation Module) zbudowanym na bazie kryptosystemu Niederreiter w kodach liniowej korekcji błędów. Wykorzystuje on zalety polegania na Quasi-Cyclic Low-Density-Parity-Check (QC_LDPC), które zapewniają wysokie prędkości dekodowania. Algorytm wprowadza dwa podstawowe usprawnienia:

- używanie kluczy efemerycznych jest wykorzystywane do udaremniania ataków statystycznych
- algorytm zapewnia szybsze dekodowanie niż zwykłe przerzucanie bitów, zapisuje on ciężkie obliczenia na macierzach i pozwala na zmniejszenie wielkości klucza prywatnego

LEDAkem posiada zestaw parametrów wejściowych dzięki którym można dopasować poziom bezpieczeństwa.

3.2 Wysokopoziomowy opis algorytmu

Algorytm składa się z 3 głównych części:

- Generowanie klucza
- Enkapsulacja klucza
- Dekapsulacja klucza

3.2.1 Generowanie klucza

Generowanie klucza opiera się głównie na operacjach na macierzach. Kroki jakie należy wykonać, aby uzyskać wynik:

- Wygenerowanie losowej binarnej macierzy o rozmiarze r na n.

$$H = [H_0, ...H_{n_0-1}]$$

Macierz ta składa się z bloków wypełnionych bitami. Waga kolumny określana jest za pomocą sumy jedynek w kolumnie. Liczba jedynek jest określana jako:

$$n = n_0 p$$

gdzie p jest liczbą pierwszą

- Wygenerowanie losowej macierzy z n wierszami oraz n kolumnami. Macierz ta jest nazywana macierzą Q i składa się z binarnych bloków podobnie jak macierz H. Całkowita waga w kolumnie spełnia warunek:

$$m <= n$$

- Zapisanie prywatnego klucza H, Q
- Obliczenie L:

$$L = HQ = [L_0, ..., L_{n_0-1}]$$

- Zapisanie publiczengo klucza M:

$$M = (L_{n_0-1})^{-1}[L_0, ...L_{n0-2}]$$

3.3 Enkapsulacja klucza

Enkapsulacja klucza to również operacje na macierzach z tym, że operujemy tutaj w ciele Galois.

- Wygenerowanie n-bitowego wektora e z losowymi wartościami bitów o wadze t
- Obliczenie ciphertextu:

$$s = Me^T$$

- Dostarczenie klucza za pomocą KDF (Key Derivation Function). Funkcja ta jako parametr przyjmuje e i dostarcza klucz

3.4 Dekapsulacja klucza

Dekapsulacja klucza to operacja odwortna do enkapsulacji. Za jej pomocą możemy sprawdzić, to co wcześniej zostało zenkapsulowane.

- Wyciągnie wartości za pomocą Q_Decodera, który jako parametry przyjmuje s, H, Q. Q_Decoder korzysta z faktu, że cały algorytm opiera się na zbudowanych wcześniej macierzach H oraz Q
- Dostarczenie klucza poprzez ponowne użycie KDF

3.5 Parametry konfigurowalne dla algorytmu

Algorytm można skonfigurować podając zestaw parametrów. Zmiana parametrów powoduje wzrost lub spadek bezpieczeństwa algorytmu oraz czasu jego wykonania.

Parametry:

- category - zbiór wartości 1, 2, 3, 4, 5. Dla poszczególnych kategorii są ustawiane pozostałe parametry: byte_len oraz version_of_sha3. byte_len to wartość definiująca długość seeda, który jest podstawową wartością w algorytmie. version_of_sha3 to parametr definiujący konkretną wersję sha3, możemy wybrać spośród: sha3_256, sha3_384, sha3_512.
- n_0 - długość bloku w macierzy H
- p - długość wielomianu (pol)
- t - waga kolumn przy dekapsulacji (używane przy KDF dla e)
- m - schemat macierzy, zazwyczaj określany w bieżącej implementacji za pomocą biblioteki numpy jako: numpy.array([7, 6])
- dv - waga dla macierzy
- pol - macierz definiująca wielomian, zapisana za pomocą wektora, który na pierwszym miejscu ma 1, a na pozostałych zera, jego długość jest definiowana przez parametr p

3.6 Technologia implementacji

Algorytm zaimplementowano w języku Python. Oficjalna implementacja algorytmu została napisana w języku C. Python jest wysokopoziomowym językiem programowania, który w wielu aspektach usprawnił proces implementacji. Korzystano z podstawowych bibliotek języka oraz pakietu numpy, który wykorzystano do obliczeń na macierzach. Sporą część operacji wykonywano w ciele Galois, ale nawet te obliczenia wymagały zastosowania numpy'a. Oprócz tego w algorytmie skorzystano z biblioteki generującej losowe liczby oraz z biblioteki, która udostępniała zaimplementowane wersje algorytmu sha3 - hashlib. W algorytmie wykorzystano liczby "prawdziwie"losowe, które są uważane za bezpieczniejsze dla algorytmów kryptograficznych. Uzyskanie takich liczb było możliwe dzięki użyciu biblioteki secrets.

3.7 Różnice pomiędzy oficjalną implementacją a autorską

Oficjalna implementacja udostępniona w zdalnym repozytorium github.com to biblioteka, z której można skorzystać. Zawiera ona API, które nie do końca jest zgodne z opisem w dokumentacji. Mianowicie przy generowaniu klucza twórcy

proponują jako dodatkowy input wiadomość. Po głębszym prześledzeniu implementacji okazuje się, że generowanie klucza następuje poprzez losowe wygenerowanie takiej wiadomości. Nie ma to wpływu na efekt końcowy algorytmu ponieważ ogólny schemat zostaje zachowany.

Dzięki Pythonowi, dużą część operacji wykonała biblioteka numpy. Nie było konieczności implementacji funkcji operujących na macierzach. Co więcej sama implementacja działań w ciele Galois jest dużo przejrzystsza niż ta w języku C, ale jest to uwarunkowana charakterystyką języka.

Oprócz różnic czysto implementacyjnych, można zauważyć również różnicę w czasie wykonania algorytmu. Język Python jest zdecydowanie wolniejszy niż C.