Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Implementacja i testy schematu podpisów cyfrowych opartych o hierarchię drzew Merkla o różnych wysokościach

Jan Sieradzki

Praca inżynierska napisana
pod kierunkiem
dr. Przemysława Kubiaka

Politechnika
Wrocławska

Wrocław 2020

Spis treści

Wstęp

Słowa klucze : podpisy cyfrowe, drzewa Merla, kryptografia post-quantum, funkcje haszujące, implementacja, XMSS, ASN.1

W pracy tej opisany jest schemat podpisów cyfrowych omawiany w publikacji [2], który jest oparty na hierarchii drzew Merkla o różnych wysokościach (mMSS bez podziału serwisu na kilka podmiotów). Przedstawione zostały wykorzystane algorytmy wraz z ich złożonością oraz implementacja aplikacji podpisującej i aplikacji weryfikującej. Opisana została notacja ASN.1, która jest użyta do komunikacji między aplikacjami. Na koniec prezentowane są wyniki przeprowadzonych testów wydajnościowych, które sprawdzają efektywność schematu i porównują go z wzorcem XMSS+, szczegółowo omówionym w pracy [3].

Od wieków ludzie wymieniają między sobą niezliczone wiadomości i poszukują sposobu na zapewnienie bezpieczeństwa przy wymianie informacji. Przykładowo, używaną od dawna metodą są pieczęcie, świadczące o wiarygodności oraz nienaruszalności zamkniętej informacji. Obecnie, powszechnie używanym narzędziem do nadawania mocy prawnej dokumentom, jest własnoręczny podpis, który opiera się na indywidualnym charakterze pisma każdego człowieka. Niestety ta metoda, oprócz takich problemów jak łatwość złożenia fałszywego podpisu, który może okazać się ciężki do rozpoznania bez profesjonalnego dochodzenia, nie nadaje się do podpisywania coraz liczniejszych dokumentów elektronicznych czy plików. Rozwiązaniem są podpisy cyfrowe, które współcześnie mają kluczowy wpływ na bezpieczeństwo w internecie i powoli wyrastają na następce tradycyjnych podpisów ręcznych.

Podpisy cyfrowe, czyli matematyczny sposób na zapewnienie danej wiadomości niżej wymienionych cech :

- autentyczności - pewność, iż wiadomość pochodzi od określonego autora,
- niezaprzeczalności - uniemożliwia wyparcie się autorstwa informacji (podpis jest w stanie złożyć tylko jeden podmiot),
- integralności - gwarancja, iż dokument dotarł w całości oraz, że nie był modyfikowany po złożeniu podpisu.

Podpisy cyfrowe zazwyczaj realizowane są za pomocą kryptografii asymetrycznej. Najczęściej spotykanym standardem jest X.509, który definiuje infrastrukturę klucza publicznego i jej składowe takie jak urzędy certyfikacji, które są zaufanym elementem schematu i potwierdzają powiązanie danego podmiotu z odpowiadającym mu kluczem publicznym. Oprócz tego ważną sprawą w podpisach cyfrowych są instrumenty prawne, które nadają tak uwierzytelnionym dokumentom moc sprawczą.

Istnieje wiele różnych wariantów algorytmów podpisu cyfrowego. Do najbardziej powszechnych należą ECDSA oraz te oparte na RSA. Są one najszybszymi wariantami zachowującymi odpowiedni poziom bezpieczeństwa, który w dużej mierze zależy od trudności

problemów logarytmu dyskretnego oraz faktoryzacji. Z biegiem lat algorytmy związane z tymi zagadnieniami stają się coraz lepsze, co skutkuje zwiększaniem długości kluczy. Oprócz tego, dużą groźbą są komputery kwantowe. Znane są rozwiązania, które w razie zbudowania takich komputerów, znajdują odpowiedź w czasie liniowym, co kompromituje powyższe schematy podpisów cyfrowych. Dlatego rozwijane są inne podejścia. Jednym z nich jest oparty na drzewach haszujących (drzewach Merkla) schemat `MSS` (ang. *Merkle Signature Scheme*) opisany w pracy [5], rozszerzony w `XMSS` [4], a następnie w `XMSS+` [3] i ostatecznie zmodyfikowany w `mMSS` [2]. Użyta funkcja haszująca `H` musi spełniać `Second preimage resistance` (tzn. dla wiadomości m_1, trudno znaleźć m_2, t. że $H(m_1) = H(m_2)$), ponieważ w przeciwnym wypadku, po wykorzystaniu kolidującego klucza, adwersarz jest w stanie podpisać dowolną, wybraną przez siebie wiadomość (dokładniejszy opis ataku w [2]). Zaletą rodziny `MSS` jest to, iż jej bezpieczeństwo zależy od bezpieczeństwa użytej funkcji haszującej, zatem w razie złamania, można ją łatwo wymienić na inną. Co jest ważne w kontekście czasu ważności podpisu, w przeciwieństwie do kluczy `ECDSA/RSA`, klucze `MSS` są uznawane za długotrwałe, tzn. nie powinny znacząco się wydłużyć w najbliższej przyszłości, a najlepszy znany algorytm kwantowy na znalezienie kolizji funkcji haszującej, daje tylko logarytmiczne przyśpieszenie w stosunku do ataku urodzinowego (atak siłowy). Ponadto według badań autorów [3], czasy generowania kluczy, podpisów i weryfikowania sygnatur w `XMSS+` nie odbiegają znacząco od czasów `ECDSA/RSA`.

Podpisy cyfrowe często są implementowane na kartach chipowych. Rozważmy najpierw takie urządzenie, które generuje podpisy (przykładowo po podłączeniu do USB, podpisuje dokument elektroniczny). Schemat podpisów, aby działał w takim środowisku, które z racji niewielkich wymiarów ma ograniczone parametry, musi być zoptymalizowany. W takim urządzeniu podpisującym, pewnym wymogiem bezpieczeństwa jest generowanie pary kluczy schematu wewnątrz niego samego. Dzięki temu klucz prywatny nie opuszcza bezpiecznego środowiska przyrządu, który jest tak zaprojektowany, iż w przypadku jakakolwiek próby modyfikacji, urządzenie zostaje trwale uszkodzone co uniemożliwia wydobycie klucza prywatnego. Stąd w `XMSS+` oraz w algorytmie opisywanym w tej pracy głównym celem jest zminimalizowanie czasu generowania pary kluczy schematu. Analogicznie jest w urządzeniach weryfikujących podpisy. Klucz publiczny jest tam zamieszczany fizycznie (w hardware), co jednak jest problematyczne w przypadku zmiany takiego klucza publicznego. Dlatego też, fakt że podpisy `MSS` są bardziej długotrwałe, niż te oparte na `ECDSA/RSA`, działa na korzyść, gdyż zmniejsza szansę na zmianę klucza publicznego. Problemem jest sytuacja, w którym klucz prywatny zostaje wykradziony/unieważniony. Jako, że miejsce na karcie chipowej jest ograniczone, takie środki jak protokół `OCSP` są trudne do zaimplementowania. Rozwiązane jest to w `mMSS`, który pozwala na podpisanie nowego klucza prywatnego i zastąpienie nim starego, dzięki czemu mimo zmiany pary kluczy schematu, urządzenia weryfikujące mogą dalej uwierzytelniać wiadomości za pomocą starego klucza publicznego. Więcej szczegółów o `mMSS` w publikacji [2].

Ważną cechą wprowadzoną przez autorów `XMSS` [3], jest `utajnienie z wyprzedzeniem` (ang. *forward secure*). To oznacza, iż w przypadku wycieknięcia klucza prywatnego, podpisy wykonane wcześniej są niezagrożone. Aby osiągnąć taką właściwość, klucz prywatny musi się zmieniać w sposób jednokierunkowy wraz z każdym złożonym podpisem. Zazwyczaj w innych schematach, nie będących `utajnionymi z wyprzedzeniem`, jest to sztucznie osiągnięte dołączaniem do podpisu znacznika czasu (ang. *timestamp*). Jest to istotna właściwość, zakładając, że podpisane dokumenty muszą być wiarygodne nawet przez kilkadziesiąt lat.

Opis schematu

W tym rozdziale opisany jest schemat podpisów cyfrowych, oparty o hierarchię drzew Merkla o różnych wysokościach, przedstawiony w publikacji [2]. Rozwiązanie to opiera się na generowaniu jednorazowych podpisów Winternitza [W-OTS] (ang. *Winternitz one time signature*), które są tworzone przy użyciu jednorazowej pary kluczy W-OTS. W przypadku, gdyby dwie wiadomości zostały podpisane tym samym zestawem kluczy, adwersarz byłby w stanie uwierzytelnić własną wiadomość (prawdopodobnie bezsensowną).

W celu opisania W-OTS, konieczne jest zapoznanie się z parametrami i funkcjami, które zostaną użyte. W całej pracy *log* oznacza logarytm o podstawie dwa. Parametr $n \in N$ oznacza długość bloku bitów, na których będzie przeprowadzona większość operacji. Współczynnik Winternitza $w \in N$ definiuje bazę systemu liczbowego, w którym wiadomość jest przeliczana na sygnaturę. Pozwala to na wymianę pomiędzy czasem generowania podpisu, a pamięcią zajmowaną przez podpis, gdzie mniejsze w oznacza zysk czasowy, a większe w zysk pamięciowy.

Niech $F = \{F_k : \{0,1\}^n \to \{0,1\}^n \mid k \in \{0,1\}^n\}$ będzie rodziną funkcji pseudolosowej, a $X \in \{0,1\}^n$ losowo wybraną wartością, która jest trzymana w kluczu publicznym schematu. Wtedy, dla $e \in N$, jest zdefiniowana rekurencyjnie konstrukcja $F_k^e(X)$:

$$\begin{cases} F_k^0(X) = k \\ F_k^{i-1}(X) = k' & dla\ i > 0 \\ F_k^i(X) = F_{k'}(X) & dla\ i > 0 \end{cases}$$

Oprócz tego, są długości l_1, l_2 oraz l, wyliczone według wzorów poniżej :

$$l_1 = \left\lceil \frac{m}{log(w)} \right\rceil,\ l_2 = \left\lfloor \frac{log(l_1(w-1))}{log(w)} \right\rfloor + 1,\ l = l_1 + l_2$$

l_1 oznacza ilość ciągów $M_i \in \{0,\ldots,w-1\}$ w $M = (M_1,\ldots,M_{l_1})$, gdzie M jest otrzymane po przekształceniu binarnej wiadomości podanej na wejściu (o długości m) na system liczbowy w. l_2 oznacza ilość ciągów $C_i \in \{0,\ldots,w-1\}$ w $C = (C_1,\ldots,C_{l_2})$, gdzie C jest otrzymane po wyliczeniu sumy kontrolnej następującym wzorem :

$$C = \sum_{i=1}^{l_1} (w - 1 - M_i)$$

Po wyliczeniu M i C tworzone jest $T = (M_1,\ldots,M_{l_1},C_1,\ldots,C_{l_2}) = (T_1,\ldots,T_l)$. Zatem l oznacza długość ciągu T, a oprócz tego długość kluczy W-OTS prywatnego K_{priv} i publicznego K_{pub} oraz wartości δ. Sposób otrzymywania K_{priv} zostanie opisany później, natomiast K_{pub} wyliczany jest dla $K_{priv} = (K_{priv_1},K_{priv_2},\ldots,K_{priv_l})$, w, X, $oraz$ $F_k^e(X)$ według poniższego wzoru :

$$K_{pub} = (F^{w-1}_{K_{priv_1}}(X),F^{w-1}_{K_{priv_2}}(X),\ldots,F^{w-1}_{K_{priv_l}}(X))$$

Dzięki tej wiedzy, można teraz przejść do opisu generowania i weryfikowania wartości δ, która wyliczana jest za pomocą T i K_{priv} :

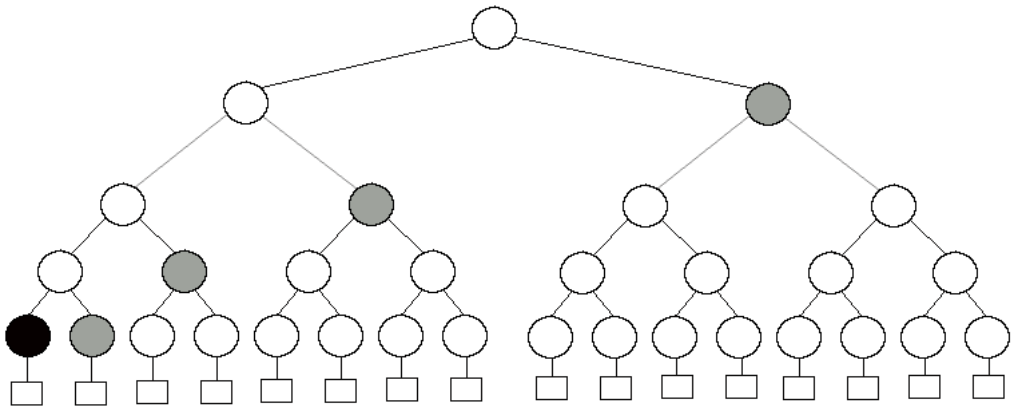
$$\delta = (\delta_1,\delta_2,...,\delta_l) = (F^{T_1}_{K_{priv_1}}(X),F^{T_2}_{K_{priv_2}}(X),...,F^{T_l}_{K_{priv_l}}(X))$$

Weryfikacja zaś polega, na wyliczeniu klucza publicznego `W-OTS`, z użyciem, danej w podpisie cyfrowym, wartości δ, parametru w i wyliczonego (na podstawie wiadomości M) T :

$$(F^{w-1-T_1}_{\delta_1}(X),F^{w-1-T_2}_{\delta_2}(X),...,F^{w-1-T_l}_{\delta_l}(X)) = K_{pub}$$

Powyższa równość jest prawdziwa tylko wtedy, kiedy użyta została wiadomość wraz z pasującą do niej sygnaturą. Wtedy otrzymany K_{pub} jest prawidłowy i podpis zostanie zweryfikowany jako prawidłowy, po wykonaniu pozostałej części schematu opisanej w sekcji 3.6.

Po zapoznaniu się ze schematem `W-OTS`, może zastanawiać sens jednorazowych par kluczy, gdyż trzymanie osobnego klucza publicznego dla każdego podpisu jest mało praktyczne. Dlatego w schematach `MSS` korzysta się z drzew Merkla.



Rysunek 2.1: XMSS, drzewo Merkla o wysokości H = 4

Na powyższym rysunku 2.1, widnieje przykładowe drzewo Merkla o wysokości H = 4. W dalszej części pracy poszczególne węzły oznaczone będą poprzez $W_h[i]$, gdzie h oznacza wysokość węzła (liście mają $h = 0$, korzeń $h = H$), a i indeks liścia (dla każdego h $i \in \{0,\ldots,2^{H-h} - 1\}$ numerowane od lewej do prawej). Kolejne, coraz wyższe, węzły drzewa Merkla są wyliczane rekurencyjnie na podstawie potomnych węzłów i maski $Mask_h$ dla $h \in \{1,\ldots,H\}$ (maska jest częścią klucza publicznego):

$$W_{h+1}[i] = H((W_h[2 \cdot i]||W_h[2 \cdot i + 1])\ xor\ Mask_{h+1})$$

W tym równaniu oraz w pozostałej części pracy niech $H : \{0,1\}^{2 \cdot n} \rightarrow \{0,1\}^n$ oznacza losowo wybraną funkcją haszującą, która spełnia wymagania `second preimage resistance`.

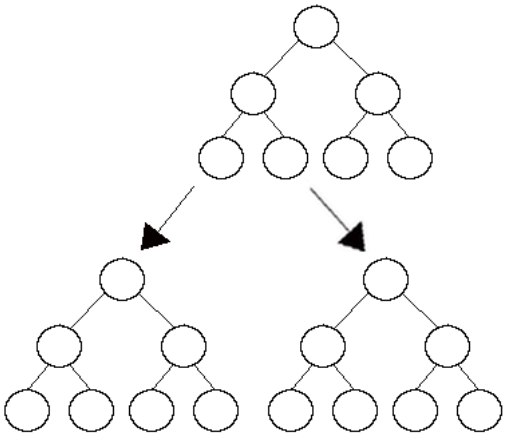
W przypadku `XMSS` korzeń drzewa $W_H[0]$ jest kluczem publicznym schematu. Wyliczany jest on na podstawie liści, które są otrzymane z kluczy publicznych `W-OTS` (algorytm $Treehash$ w sekcji 3.2). Dzięki takiej konstrukcji, weryfikacja podpisu cyfrowego sprowadza się do wyliczenia klucza publicznego `W-OTS` na podstawie wiadomości M i wartości δ (zgodnie z wcześniejszym opisem), a następnie sprawdzeniu, czy otrzymany klucz należy do drzewa Merkla schematu. Osiąga się to poprzez obliczenie korzenia, za pomocą ścieżki uwierzytelniającej $Auth_i$ dla $i \in \{0,\ldots,H - 1\}$, tzn. H węzłów potrzebnych do

otrzymania korzenia struktury dla danego liścia. Przykładowo, na rysunku 2.1 węzły szare są ścieżką uwierzytelniającą dla czarnego liścia. Rekurencyjny wzór na korzeń drzewa, z daną ścieżką uwierzytelniającą, indeksem liścia i (te dane są częścią podpisu cyfrowego), oraz z wyliczonym liściem $W_0[i]$ wygląda następująco:

$$\begin{cases} W_0 = W_0[i] & dla\ i = 0 \\ W_j = H((W_{j-1}||Auth_{j-1})\ xor\ Mask_j) & dla\ \left\lfloor \frac{i}{2^j} \right\rfloor\ mod\ 2 == 0 \\ W_j = H((Auth_{j-1}||W_{j-1})\ xor\ Mask_j) & dla\ \left\lfloor \frac{i}{2^j} \right\rfloor\ mod\ 2 == 1 \end{cases}$$

Jak wspomniano, liście oblicza się na podstawie klucza publicznego `W-OTS`. W tym celu używa się tak zwanego `L-drzewa`. L-drzewo jest to drzewo Merkla, którego liście są kolejnymi fragmentami klucza $K_{pub} = (K_{pub_1}, \ldots, K_{pub_l})$, $K_{pub_i} \in \{0,1\}^n$. Jako, że l niekoniecznie musi być potęgą dwójki, aby umożliwić obliczenie korzenia, po wykonaniu wszystkich możliwych rachunków (szczegóły wykonywanych operacji w sekcji 3.1), lewy liść który nie ma prawego brata, jest podnoszony do momentu, aż nie stanie się prawym bratem innego węzła. Korzeń L-drzewa jest liściem głównego drzewa. W L-drzewie stosuje się odrębną maskę : $LMask_h$ dla $h \in \{1, \ldots, \lceil log(l) \rceil\}$.

To był zarys działania `XMSS`. Schemat `XMSS+` wprowadza pewną modyfikację. Pojedyncze drzewo o wysokości H, rozbija się na dwa mniejsze, o wysokości $H' = \frac{H}{2}$. Tworzą one hierarchię drzew. Górne drzewo służy do podpisywania dolnego drzewa, a dolne służy do podpisywania wiadomości. Kiedy dolne drzewo podpisze $2^{H'}$ wiadomości i zużyje tym samym wszystkie liście, jest zastępowane nowym dolnym drzewem o wysokości H', które jest podpisywane przez następny liść górnego drzewa. W ten sposób struktura ma do dyspozycji $2^{H'}$ dolnych drzew o pojemności $2^{H'}$ podpisów. Na początku jest generowane tylko jedno drzewo górne i dolne. Następne dolne drzewa są generowane, w sposób zbalansowany, w trakcie podpisywania kolejnych wiadomości. Kluczem publicznym schematu jest korzeń górnego drzewa.



Rysunek 2.2: XMSS+, hierarchia drzew Merkla o wysokości H' = 2

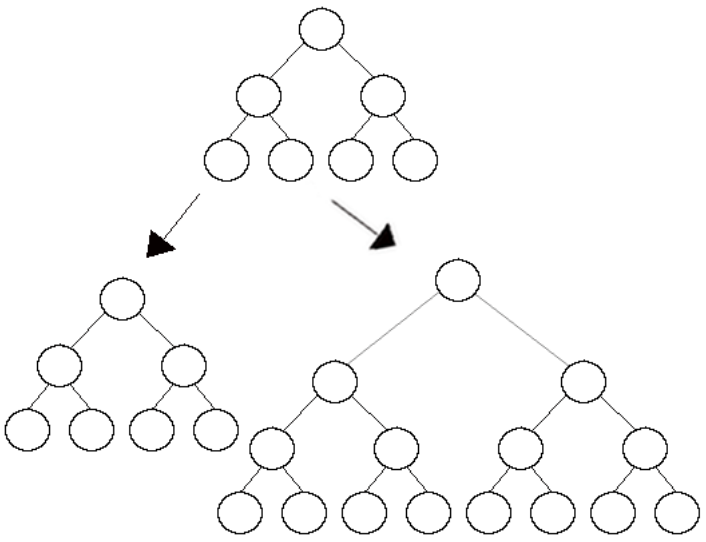
Taka modyfikacja, zachowując tą samą pojemność (2^H) podpisów cyfrowych, zmniejsza czas generowania struktury z $O(2^H)$ do $O(2^{\frac{H}{2}})$ (na początku generowane jest tylko jedno drzewo górne i jedno dolne), co jest znaczącym przyśpieszeniem. Okupione jest to wzrostem zajmowanej pamięci przez podpis cyfrowy, z powodu potrzeby dodania do niego

podpisu dolnego drzewa, złożonego przez górne. Klucz publiczny, który przechowuje parametr X, korzeń górnego drzewa, maskę L-Drzewa i maskę główną, nieznacznie zyskuje pamięciowo, ponieważ maska główna, stosowana zarówno do dolnego, jak i górnego drzewa, ma zmniejszoną długość z H (`XMSS`) do $\frac{H}{2}$ (`XMSS+`). Występuje również niewielki wzrost czasu składania podpisu, gdyż równolegle podpisywane i generowane jest następne dolne drzewo, które zastąpi aktualne, w momencie gdy dolne drzewo zużyje wszystkie swoje liście.

W celu ułatwienia dalszych opisów, zostaje wprowadzona notacja : H_u oznaczająca wysokość drzewa górnego, oraz H_l oznaczająca wysokość drzewa dolnego.

Nadszedł czas na przedstawienie schematu o rosnących poddrzewach, którego dotyczy ta praca. Jest to zmodyfikowana wersja `XMSS+`, w której H_u może się różnić od H_l (w `XMSS+` $H_u = H_l = \frac{H}{2}$), a wysokość każdego kolejnego dolnego drzewa, które jest generowane w trakcie działania schematu, zwiększa się o 1 względem poprzedniego. W ten sposób, dla struktury wygenerowanej z wysokościami $H_u = 3$ i $H_l = 4$, powstanie łącznie 7 drzew, o wysokościach (4,5,6,7,8,9,10). Wielkość maski głównej zmieniła się w stosunku do `XMSS+` i jej długość to $max(H_u,H_l)$.

Ostatni liść górnego drzewa jest zarezerwowany, na podpisanie nowej struktury. Jest to rozszerzenie względem `XMSS+`, które powoduje, że schemat jest w stanie podpisywać kolejne wiadomości, nawet po zapełnieniu całej struktury, gdyż może zastąpić ją nową, zostawiając przy tym stary klucz publiczny. Jest to niestety kosztowne, gdyż każde dołączenie nowej struktury, implikuje dodatkowy podpis cyfrowy, który należy zweryfikować podczas weryfikacji każdej kolejnej wiadomości, co wydłuża czas tej operacji. Ponadto, generowanie nowej struktury (sekcja 3.4) ma największą złożoność spośród wszystkich algorytmów użytych w tym schemacie. Z tych powodów, dobrze jest "wymierzyć" strukturę do przewidywanej liczby złożonych podpisów, choć niewątpliwą zaletą i zabezpieczeniem jest możliwość dalszego działania schematu w razie przepełnienia, szczególnie w przypadku, gdy ciężko określić liczbę wiadomości do uwierzytelnienia.



Rysunek 2.3: hierarchia drzew Merkla o różnych wysokościach, $H_u = 3$, $H_l = 3$

Wielką przewagą tego rozwiązania nad `XMSS+`, jest stosunek czasu generacji struktury do liczby podpisów. Podobnie jak w `XMSS+`, na początku tworzone jest górne drzewo i tylko jedno dolne. Różnice w pojemności struktur pomiędzy schematami można zaobserwować w poniższych tabelach 2.1 i 2.2.

Tablica 2.1: Pojemności struktur dla XMSS+

H górne	H dolne	pojemność struktury
2	2	$(2^2 - 1) * 2^2$
3	3	$(2^3 - 1) * 2^3$
4	4	$(2^4 - 1) * 2^4$
5	5	$(2^5 - 1) * 2^5$
6	6	$(2^6 - 1) * 2^6$
7	7	$(2^7 - 1) * 2^7$
8	8	$(2^8 - 1) * 2^8$
9	9	$(2^9 - 1) * 2^9$
10	10	$(2^{10} - 1) * 2^{10}$
11	11	$(2^{11} - 1) * 2^{11}$
12	12	$(2^{12} - 1) * 2^{12}$
13	13	$(2^{13} - 1) * 2^{13}$
14	14	$(2^{14} - 1) * 2^{14}$
15	15	$(2^{15} - 1) * 2^{15}$
16	16	$(2^{16} - 1) * 2^{16}$
17	17	$(2^{17} - 1) * 2^{17}$
18	18	$(2^{18} - 1) * 2^{18}$
19	19	$(2^{19} - 1) * 2^{19}$
20	20	$(2^{20} - 1) * 2^{20}$

Tablica 2.2: Pojemności struktur dla drzew rosnących

	H górne = 2	H górne = 3	H górne = 4	H górne = 5	H górne = 6
H dolne = 2	$2^5 - 2^2$	$2^9 - 2^2$	$2^{17} - 2^2$	$2^{33} - 2^2$	$2^{65} - 2^2$
H dolne = 3	$2^6 - 2^3$	$2^{10} - 2^3$	$2^{18} - 2^3$	$2^{34} - 2^3$	$2^{66} - 2^3$
H dolne = 4	$2^7 - 2^4$	$2^{11} - 2^4$	$2^{19} - 2^4$	$2^{35} - 2^4$	$2^{67} - 2^4$
H dolne = 5	$2^8 - 2^5$	$2^{12} - 2^5$	$2^{20} - 2^5$	$2^{36} - 2^5$	$2^{68} - 2^5$
H dolne = 6	$2^9 - 2^6$	$2^{13} - 2^6$	$2^{21} - 2^6$	$2^{37} - 2^6$	$2^{69} - 2^6$
H dolne = 7	$2^{10} - 2^7$	$2^{14} - 2^7$	$2^{22} - 2^7$	$2^{38} - 2^7$	$2^{70} - 2^7$
H dolne = 8	$2^{11} - 2^8$	$2^{15} - 2^8$	$2^{23} - 2^8$	$2^{39} - 2^8$	$2^{71} - 2^8$
H dolne = 9	$2^{12} - 2^9$	$2^{16} - 2^9$	$2^{24} - 2^9$	$2^{40} - 2^9$	$2^{72} - 2^9$
H dolne = 10	$2^{13} - 2^{10}$	$2^{17} - 2^{10}$	$2^{25} - 2^{10}$	$2^{41} - 2^{10}$	$2^{73} - 2^{10}$
H dolne = 11	$2^{14} - 2^{11}$	$2^{18} - 2^{11}$	$2^{26} - 2^{11}$	$2^{42} - 2^{11}$	$2^{74} - 2^{11}$
H dolne = 12	$2^{15} - 2^{12}$	$2^{19} - 2^{12}$	$2^{27} - 2^{12}$	$2^{43} - 2^{12}$	$2^{75} - 2^{12}$
H dolne = 13	$2^{16} - 2^{13}$	$2^{20} - 2^{13}$	$2^{28} - 2^{13}$	$2^{44} - 2^{13}$	$2^{76} - 2^{13}$
H dolne = 14	$2^{17} - 2^{14}$	$2^{21} - 2^{14}$	$2^{29} - 2^{14}$	$2^{45} - 2^{14}$	$2^{77} - 2^{14}$
H dolne = 15	$2^{18} - 2^{15}$	$2^{22} - 2^{15}$	$2^{30} - 2^{15}$	$2^{46} - 2^{15}$	$2^{78} - 2^{15}$
H dolne = 16	$2^{19} - 2^{16}$	$2^{23} - 2^{16}$	$2^{31} - 2^{16}$	$2^{47} - 2^{16}$	$2^{79} - 2^{16}$
H dolne = 17	$2^{20} - 2^{17}$	$2^{24} - 2^{17}$	$2^{32} - 2^{17}$	$2^{48} - 2^{17}$	$2^{80} - 2^{17}$
H dolne = 18	$2^{21} - 2^{18}$	$2^{25} - 2^{18}$	$2^{33} - 2^{18}$	$2^{49} - 2^{18}$	$2^{81} - 2^{18}$
H dolne = 19	$2^{22} - 2^{19}$	$2^{26} - 2^{19}$	$2^{34} - 2^{19}$	$2^{50} - 2^{19}$	$2^{82} - 2^{19}$
H dolne = 20	$2^{23} - 2^{20}$	$2^{27} - 2^{20}$	$2^{35} - 2^{20}$	$2^{51} - 2^{20}$	$2^{83} - 2^{20}$
	H górne = 7	H górne = 8	H górne = 9	H górne = 10	H górne = 11
H dolne = 2	$2^{129} - 2^2$	$2^{257} - 2^2$	$2^{513} - 2^2$	$2^{1025} - 2^2$	$2^{2049} - 2^2$
H dolne = 3	$2^{130} - 2^3$	$2^{258} - 2^3$	$2^{514} - 2^3$	$2^{1026} - 2^3$	$2^{2050} - 2^3$
H dolne = 4	$2^{131} - 2^4$	$2^{259} - 2^4$	$2^{515} - 2^4$	$2^{1027} - 2^4$	$2^{2051} - 2^4$
H dolne = 5	$2^{132} - 2^5$	$2^{260} - 2^5$	$2^{516} - 2^5$	$2^{1028} - 2^5$	$2^{2052} - 2^5$
H dolne = 6	$2^{133} - 2^6$	$2^{261} - 2^6$	$2^{517} - 2^6$	$2^{1029} - 2^6$	$2^{2053} - 2^6$
H dolne = 7	$2^{134} - 2^7$	$2^{262} - 2^7$	$2^{518} - 2^7$	$2^{1030} - 2^7$	$2^{2054} - 2^7$
H dolne = 8	$2^{135} - 2^8$	$2^{263} - 2^8$	$2^{519} - 2^8$	$2^{1031} - 2^8$	$2^{2055} - 2^8$
H dolne = 9	$2^{136} - 2^9$	$2^{264} - 2^9$	$2^{520} - 2^9$	$2^{1032} - 2^9$	$2^{2056} - 2^9$
H dolne = 10	$2^{137} - 2^{10}$	$2^{265} - 2^{10}$	$2^{521} - 2^{10}$	$2^{1033} - 2^{10}$	$2^{2057} - 2^{10}$
H dolne = 11	$2^{138} - 2^{11}$	$2^{266} - 2^{11}$	$2^{522} - 2^{11}$	$2^{1034} - 2^{11}$	$2^{2058} - 2^{11}$
H dolne = 12	$2^{139} - 2^{12}$	$2^{267} - 2^{12}$	$2^{523} - 2^{12}$	$2^{1035} - 2^{12}$	$2^{2059} - 2^{12}$
H dolne = 13	$2^{140} - 2^{13}$	$2^{268} - 2^{13}$	$2^{524} - 2^{13}$	$2^{1036} - 2^{13}$	$2^{2060} - 2^{13}$
H dolne = 14	$2^{141} - 2^{14}$	$2^{269} - 2^{14}$	$2^{525} - 2^{14}$	$2^{1037} - 2^{14}$	$2^{2061} - 2^{14}$
H dolne = 15	$2^{142} - 2^{15}$	$2^{270} - 2^{15}$	$2^{526} - 2^{15}$	$2^{1038} - 2^{15}$	$2^{2062} - 2^{15}$
H dolne = 16	$2^{143} - 2^{16}$	$2^{271} - 2^{16}$	$2^{527} - 2^{16}$	$2^{1039} - 2^{16}$	$2^{2063} - 2^{16}$
H dolne = 17	$2^{144} - 2^{17}$	$2^{272} - 2^{17}$	$2^{528} - 2^{17}$	$2^{1040} - 2^{17}$	$2^{2064} - 2^{17}$
H dolne = 18	$2^{145} - 2^{18}$	$2^{273} - 2^{18}$	$2^{529} - 2^{18}$	$2^{1041} - 2^{18}$	$2^{2065} - 2^{18}$
H dolne = 19	$2^{146} - 2^{19}$	$2^{274} - 2^{19}$	$2^{530} - 2^{19}$	$2^{1042} - 2^{19}$	$2^{2066} - 2^{19}$
H dolne = 20	$2^{147} - 2^{20}$	$2^{275} - 2^{20}$	$2^{531} - 2^{20}$	$2^{1043} - 2^{20}$	$2^{2067} - 2^{20}$

Ogromna pojemność, osiągnięta małym czasem generacji, jest okupiona czasem składania podpisów, który się stopniowo wydłuża i mimo początkowych mniejszych wartości niż w `XMSS+`, po pewnej ilości wystawionych sygnatur czasy stają się większe. Schemat opisywany w tej pracy jest rozwiązaniem mniej stałym niż `XMSS+`, jednak bardziej elastycznym, w kontekście ograniczonej możliwości przewidzenia ilości podpisów do złożenia.

Klucz prywatny w implementacji omawianej w tej pracy, zawiera między innymi ziarna do generatorów pseudolosowych. Taki generator z każdym wywołaniem zwraca inne ziarno, z którego otrzymywane jest l n-bitowych ciągów, składających się na klucz prywatny `W-OTS`. Oprócz tego, generator zwraca nowe ziarno dla samego siebie. W ten sposób aktualizuje się z każdym złożonym podpisem, wypełniając warunek `utajnienia z wyprzedzeniem` (ang. `forward secure`). Generator jest skonstruowany z dwóch funkcji pseudolosowych z rodziny F w następujący sposób :

$$FSGen(S_i) = (S_{i+1}||R_i) = (F_{S_i}(0)||F_{S_i}(1))$$

Gdzie R_i jest wspomnianym ziarnem, które służy do wygenerowania $K_{priv} = (K_{priv_1}, K_{priv_2}, ..., K_{priv_l})$:

$$K_{priv_j} = F_{R_i}(j-1),\ 1 \leqslant j \leqslant l$$

We wszystkich omówionych schematach, podstawowymi elementami schematu są: klucz publiczny, klucz prywatny oraz podpis. Poniżej przedstawione są zawartości tych struktur danych.

`Klucz prywatny zawiera następujące dane` :

* ziarno generatora dla górnego drzewa
* ziarno generatora dla dolnego drzewa
* ziarno generatora dla następnego dolnego drzewa
* stan algorytmu obliczania węzłów uwierzyt. dla górnego drzewa (więcej w sekcji 3.3)
* stan algorytmu obliczania węzłów uwierzyt. dla dolnego drzewa (więcej w sekcji 3.3)
* stan algorytmu *Treeash* (buduje następne drzewo, więcej w sekcji 3.2)
* podpis cyfrowy dolnego drzewa (złożony przez górne drzewo)

`W kluczu publicznym schematu znajdują się` :

* korzeń górnego drzewa
* maska główna
* maska L-drzewa
* parametr X

`Oraz na koniec zawartość podpisu wiadomości` :

* ścieżka uwierzytelniająca dla dolnego drzewa
* wartość δ otrzymana z wiadomości
* indeks użytego liścia w dolnym drzewie
* ścieżka uwierzytelniająca dla górnego drzewa [część podpisu dolnego drzewa]
* wartość δ otrzymana z korzenia dolnego drzewa [część podpisu dolnego drzewa]
* indeks użytego liścia w górnym drzewie [część podpisu dolnego drzewa]
* tablica podpisów starszych struktur (zawierająca elementy w przypadku, gdy wcześniejsze struktury zostały wypełnione i w ich miejsce wygenerowane zostały nowe, z nowymi kluczami schematu)

W tym rozdziale został przedstawiony ogólna koncepcja schematów, którymi zajmuje się ta praca. W następnym rozdziale dokładniej będą opisane i zbadane algorytmy użyte do realizacji omówionych tu operacji oraz precyzyjniej wyjaśnione zastosowanie danych znajdujących się w powyższych strukturach.

Opis algorytmów oraz ich złożoności

W tym rozdziale dokładniej opisane są algorytmy, użyte w implementacji schematu opartego o hierarchię drzew Merkla o różnych wysokościach. Omówione są algorytmy: $leafcalc$ - obliczający liście, $treehash$ - służący do efektywnego wyliczania poszczególnych węzłów w drzewie oraz algorytm obliczania ścieżek uwierzytelniających. Ponadto opisane zostało generowanie struktury i składanie/weryfikacja podpisów.

3.1 Algorytm obliczania liści (leafcalc)

implementacja

Algorytm $leafcalc$ służy do wyliczania liści na podstawie kluczy publicznych `W-OTS`. Zgodnie z propozycją w pracy [3], użyte zostało do tego `L-drzewo`, tzn. drzewo Merkla, które ma l liści. Tymi liśćmi są kolejne fragmenty klucza publicznego $K_{pub} = (K_{pub_1}, \ldots, K_{pub_l})$, $K_{pub_i} \in \{0,1\}^n$. `L-drzewo` ma wysokość $H = \lceil log(l) \rceil$. Poniżej, w pseudokodzie 3.1 przedstawiony jest algorytm :

Pseudokod 3.1: Algorytm leafcalc

Input: Indeks liścia α, ziarno R_α
Output: Liść o indeksie α

$STACK \leftarrow$ inicjacja pustego stosu $STACK$
for ($i = 1$ **to** l) **do**
    $K_{priv_i} \leftarrow F_{R_\alpha}(i-1)$
    $LEAF \leftarrow F^{w-1}_{K_{priv_i}}(X)$
    $h \leftarrow$ wysokość $LEAF$ (liście zawsze mają $h = 0$)
    while (h jest równe wysokości węzła na wierzchu stosu $STACK$ (i $STACK \neq \emptyset$)) **do**
        $h \leftarrow h + 1$
        $TOP \leftarrow STACK.pop()$
        $LEAF \leftarrow$ H($(TOP \mid\mid LEAF)$ xor $LMASK_h$)
    $STACK.push(LEAF)$
if ($STACK.size() \neq 1$) **then**
    $LEAF \leftarrow$ stack.pop()
    $h \leftarrow$ wysokość węzła na szczycie stosu ($STACK.pop().height$)
    while (h jest większe od wysokości węzła w $LEAF$) **do**
        $LEAF.height = LEAF.height + 1$
    while (h jest równe wysokości węzła na wierzchu stosu $STACK$) **do**
        $h \leftarrow h + 1$
        $TOP \leftarrow STACK.pop()$
        $LEAF \leftarrow$ H($(TOP \mid\mid LEAF)$ xor $LMASK_h$)
Return $STACK.pop()$

Na wejściu podawany jest numer liścia do wyliczenia oraz ziarno R_α. Na podstawie R_α, otrzymana jest funkcja pseudolosowa F_{R_α}, która pozwala na wyliczenie l części klucza prywatnego `W-OTS`. Następnie, zgodnie ze schematem Winternitza, obliczana jest część klucza publicznego `W-OTS`. W pętli `for`, kolejne takie części są dodawane do stosu. Oprócz tego, w każdej iteracji jest sprawdzane, czy nowo dodawany do stosu węzeł, nie jest rodzeństwem (nie ma takiej samej wysokości), jak wierzchni węzeł stosu. Wtedy, dopóki ten warunek jest spełniony, zdejmuje się wierzchni węzeł ze stosu, i wylicza, w miejsce nowego liścia, rodzica tych dwóch węzłów. Na koniec pętli, tak otrzymany liść dodawany jest do stosu. W ten sposób drzewo Merkla jest konsekwentnie obliczane od lewej do prawej.

Po dodaniu l liści, w 11 linijce jest sprawdzany warunek, czy rozmiar stosu jest różny od 1. Jeśli tak, to znaczy, że l jest potęgą 2 i obliczony został korzeń `L-drzewa`, tzn. liść głównego drzewa. W przeciwnym wypadku, ostatnio dodany na stos węzeł nie ma prawego brata, co blokuje dalsze operacje. Wtedy należy zwiększyć wysokość takiego węzła, do momentu kiedy sam nie stanie się prawym bratem. Dopiero wtedy można wyliczyć korzeń, z pozostałych w stosie węzłów, który jest wynikowym liściem o indeksie α, podanym na wejściu algorytmu.

analiza złożoności

Na złożoność tego algorytmu składają się dwie rzeczy. Pierwszą z nich jest generowanie l kolejnych części klucza publicznego `W-OTS`. Aby to osiągnąć należy wywołać l razy funkcję pseudolosową F_{R_α} oraz l razy konstrukcję $F^{w-1}_{K_{priv_i}}(X)$, która wywołuje funkcje pseudolosowe z rodziny F $w-1$ razy. Otrzymana złożoność czasowa wynosi $O(l \cdot F + l \cdot (w-1) \cdot F)$ $= O(F \cdot l \cdot w)$, gdzie F to koszt użycia funkcji pseudolosowej F, a w to parametr Winternitza. Pamięć, z racji wykorzystania generatorów, ogranicza się do przetrzymywania ziarna otrzymanego na wejściu.

Drugą składową złożoności jest wyliczenie korzenia, czyli tak naprawdę l-krotne wykonanie algorytmu $Treehash$ (z tą różnicą, iż przekazywane są gotowe liście, a nie indeksy do wyliczenia). Wymaga to $O(l-1)$ wywołań funkcji haszującej (linijki 9 i 19 w pseudokodzie 3.1). Jeżeli chodzi o pamięć, stos $STACK$ przetrzymuje najwyżej $H = \lceil log(l) \rceil$ węzłów w jednym momencie.

Otrzymana złożoność czasowa algorytmu $leafcalc$ wynosi zatem $O(H \cdot (l-1) + F \cdot l \cdot w)$, a pamięciowa $O(\lceil log(l) \rceil \cdot n)$ bitów, (w opisywanej implementacji koszt przetrzymywania węzła to około n bitów).

3.2 Algorytm Treehash

implementacja

Algorytm $Treehash$, opisany w pracy [1], służy do obliczania poszczególnych węzłów w drzewie Merkla (zarówno korzenia jak i tych, znajdujących się niżej). Jego zaletą jest efektywne zarządzanie pamięcią. Korzysta ze stosu, który przetrzymuje tylko te węzły,

które są niezbędne do dalszych obliczeń.

Pseudokod 3.2: Algorytm Treehash

Input: Indeks liścia α, stos $STACK$, ziarno $SEEDACTIVE$
Output: Zaktualizowany stos $STACK$

$R_\alpha \leftarrow$ otrzymany za pomocą generatora pseudolosowego $FSGen(Treehash_h.SEEDACTIVE) = (S_{\alpha+1}||R_\alpha)$
$Treehash_h.SEEDACTIVE \leftarrow S_{\alpha+1}$ aktualizacja $Treehash_h.SEEDACTIVE)$
$LEAF \leftarrow$ `LEAFCALC`(α, R_α)
$h \leftarrow$ wysokość $LEAF$ (liście zawsze mają $h = 0$)
while (h jest równe wysokości węzła na wierzchu stosu $STACK$) **do**
    $h \leftarrow h + 1$
    $TOP \leftarrow STACK.pop()$
    $LEAF \leftarrow \mathrm{H}((TOP \,||\, LEAF)$ xor $MASK_h)$
$STACK.push(LEAF)$
Return $STACK$

Analogiczne rozwiązanie było już przedstawione w algorytmie $leafcalc$ 3.1, które jest tak naprawdę szczególnym przypadkiem $Treehash$ (różnią się generacją liści, oraz "dopełnianiem" brakujących liści w $leafcalc$. Instancje $Treehash$ jednak, w odróżnieniu od $leafcalc$, są wykorzystywane jako obiekty w algorytmie obliczania węzłów uwierzytelniających 3.3. Obiekty te są indeksowane wysokością ($Treehash_h$) oraz mają następujące pola i metody:

* $Treehash_h.node \leftarrow$ jeśli to pole jest puste, trafia do niego pierwszy liść wrzucony do algorytmu. Pozostałe węzły trafiają na stos $STACK$. Pole $node$ zawsze przetrzymuje węzeł, o aktualne najwyższej wysokości. Gdy instancja $Treehash_h$ zakończy działanie, w $Treehash_h.node$ będzie się znajdował węzeł o wysokości h.

* $Treehash_h.initialize(\alpha, SEEDNEXT_h) \leftarrow$ metoda inicjalizuje instancję $Treehash_h$ indeksem liścia α oraz $Treehash_h.SEEDACTIVE$, pozwalającego na wyliczenie pierwszego liścia.

* $Treehash_h.update() \leftarrow$ wywołany jest algorytm $Treehash$, opisany w pseudokodzie 3.2, dla następnego indeksu $\alpha + 1$ i ziarna $SEEDACTIVE$ (po wykonaniu algorytmu $SEEDACTIVE$ jest zaktualizowany : $Treehash_h.SEEDACTIVE = S_{\alpha+1}$.

* $Treehash_h.height \leftarrow$ te pole przetrzymuje najmniejszą wysokość, spośród wszystkich węzłów trzymanych na stosie $STACK$ lub w $Treehash_h.node$. Jeśli dana instancja nie ma jeszcze zapisanego węzła w $Treehash_h.node$ (tym bardziej na stosie $STACK$), to $Treehash_h.node = h$. Jeśli $Treehash_h$ nie jest zainicjowana, lub jest już zakończona, to $Treehash_h.node = \infty$.

* $Treehash_h.SEEDACTIVE \leftarrow$ przetrzymuje ziarno, pozwalające na wyliczenie liścia, potrzebnego do wykonania iteracji algorytmu.

Każdy $Treehash_h$ ma na celu wyliczenie określonego prawego węzła na wysokości h. Wszystkie instancje korzystają z jednego, wspólnego stosu $STACK$. Dowód poprawności działania (poszczególne instancje nie przeszkadzają sobie, mimo wspólnego stosu) przedstawiony jest w pracy [1].

analiza złożoności

Złożoność $Treehash$ będzie analogiczna do złożoności $leafcalc$. Potrzeba 2^h-krotnie wywołać algorytm (wykonać $Treehash_h.update()$), aby obliczyć pożądany węzeł. Wymaga to $O(2^h - 1)$ wywołań funkcji haszującej. Oprócz tego dochodzi czas generowania liści, czyli 2^h-krotne wywołanie $leafcalc$, oraz 2^h-krotne użycie $FSGen$ (czyli wedle tej implementacji, dwukrotne wywołanie funkcji pseudolosowej z rodziny F). Otrzymana złożoność czasowa wynosi : $O((2^h-1)\cdot S+2^h\cdot(S\cdot(l-1)+F\cdot l\cdot w+2\cdot F)) = O(2^h\cdot(F\cdot(l\cdot w+2)+l\cdot S)-S)$, gdzie S to koszt użycia funkcji haszującej, a F koszt użycia funkcji pseudolosowej.

Złożoność pamięciowa natomiast wynosi $O(\lceil log(l)\rceil \cdot n)$ bitów.

3.3 Algorytm obliczania węzłów uwierzytelniających

implementacja

W tej sekcji przedstawiony jest algorytm obliczania węzłów uwierzytelniających, opisany w publikacji [1]. Znajdywanie ścieżki uwierzytelniającej dla konkretnego liścia jest kluczowym problemem podczas składania podpisów w schematach opartych na MSS. Ideą algorytmu jest rozłożenie kosztu związanego z obliczaniem liści ($leafcalc$), gdyż jest to operacja dużo kosztowniejsza, niż wyliczanie węzłów o wysokościach $h > 0$.

Algorytm, w celu obliczenia ścieżek uwierzytelniających dla kolejnych liści drzewa Merkla, korzysta ze swoich wcześniejszych stanów, które przechowują niżej wymienione struktury danych. Parametr k oznacza, z ilu górnych poziomów drzewa, zostaną zapamiętane prawe węzły (wymiana czasu wykonania na zajmowaną pamięć). Parametr k powinien być tak dobrany, aby $H - k$ było parzyste i $k \geqslant 2$.

* $AUTH_h,\ h = 0,\ldots,H-1 \leftarrow$ tablica przetrzymująca aktualną ścieżkę uwierzytelniającą.
* $RETAIN_h,\ h = H-k,\ \ldots,H-2 \leftarrow$ tablica stosów, na których trzymane są najtrudniejsze do obliczenia prawe węzły (czyli wszystkie prawe węzły na wyższych wysokościach $h = H-k,\ldots,H-2$).
* $STACK \leftarrow$ stos węzłów, na którym operują instancje $TREEHASH_h$.
* $TREEHASH_h,\ h = 0,\ldots,H-k-1 \leftarrow$ instancje algorytmu $Treehash$, opisane w sekcji 3.2. Dzielą wspólny stos $STACK$.
* $KEEP_h,\ h = 0,\ldots,H-2 \leftarrow$ tablica przetrzymująca węzły, które są przydatne w wyliczaniu lewych węzłów uwierzytelniających.
* $SEEDNEXT_h,\ h = 0,\ldots,H-k-1 \leftarrow$ tablica przetrzymująca ziarna generatora pseudolosowego. Inicjowane są nimi instancje $TREEHASH_h$. Ziarna są aktualizowane po każdej iteracji algorytmu.

Podczas inicjalizacji obiektu opisywanego algorytmu, w tablicy $AUTH$ zapisywana jest ścieżka dla $\alpha = 0$, w instancjach $TREEHASH$ trzymane są następne prawe węzły uwierzytelniające ($TREEHASH_h.node = W_h[3]$, dla $h = 0,\ldots,H-k-1$), w $RETAIN$ zapisywane są prawe węzły dla wysokości $h = H-k,\ldots,H-2$ ($RETAIN_h.push(W_h[2j+$

3], dla $h = H - k, \ldots, H - 2$ i $j = 2^{H-h-1} - 2, \ldots, 0$), a $SEEDNEXT$ są inicjowane ziarnami dla liści $3 * 2^h$ ($SEEDNEXT_h = SEED_{3*2^h}$, dla $h = 0, \ldots, H - k - 1$).

Pseudokod 3.3: Algorytm obliczania węzłów uwierzytelniających

Input: indeks węzła $\alpha \in \{0, \ldots, 2^H - 2\}$, `H`, `K`, stan algorytmu
Output: ścieżka uwierzytelniająca dla liścia o indeksie $\alpha + 1$

$R_\alpha \leftarrow$ otrzymany przy aktualizacji klucza prywatego schematu (aktualny klucz prywatny S_{priv} znajduje się w stanie algorytmu):
$FSGen(S_{priv}) = (S_{\alpha+1}||R_\alpha) = (F_{S_\alpha}(0)||F_{S_\alpha}(1))$

$S_{priv} \leftarrow S_{\alpha+1}$ aktualizacja ziarna w kluczu prywatnym (dzięki ciągłemu modyfikowaniu klucza prywantego, schemat jest `Forward secure`)

Aktualizacja $SEEDNEXT_h$ dla $h \in 0, \ldots, H - k - 1$, aby instancje $TREHASH_h$ były zainicjowane poprawnym indeksem :
for ($h = 0$ **to** $H - k - 1$) **do**
  $SEEDNEXT_h \leftarrow S_x$ wzięte z $FSGen(SEEDNEXT_h) = (S_x||R_{x-1})$

Niech $\tau = 0$, jeśli α jest lewym węzłem, w przeciwnym razie niech τ będzie wysokością pierwszego rodzica α, który jest lewym węzłem :
$\tau \leftarrow max\{h : 2^h|(\alpha + 1)\}$

Jeśli rodzic liścia α na wysokości $\tau + 1$ jest lewym węzłem, zapisz węzeł uwierzytelniający, aktualnie trzymany w $AUTH_\tau$, w $KEEP_\tau$:
if ($\lfloor \alpha/2^{\tau+1} \rfloor \ mod\ 2 \ == \ 0$ AND $\tau < H - 1$) **then**
  $KEEP_\tau \leftarrow AUTH_\tau$

Jeśli węzeł α jest lewym, to dodaj go do ścieżki uwierzytelniającej $\alpha + 1$:
if ($\tau \ == \ 0$) **then**
  $AUTH_0 \leftarrow$ `LEAFCALC`(α,R_α)

W przeciwnym razie, jeżeli liść α jest prawym, to ścieżka uwierzytelniająca dla liścia $\alpha + 1$ zmienia się na wysokościach $0, \ldots, \tau$:
if ($\tau \ > \ 0$) **then**

  Ścieżka uwierzytelniająca dla liścia $\alpha + 1$ wymaga nowy lewy węzeł na wysokości τ. Jest on obliczany za pomocą węzła znajdującego się aktualnie w $AUTH_{\tau-1}$ i węzła trzymanego w $KEEP_{\tau-1}$ (po tej operacji węzeł trzymany w $KEEP_{\tau-1}$ może być usunięty) :
  $AUTH_\tau \leftarrow$H($(AUTH_{\tau-1}||\ KEEP_{\tau-1}\ )$ xor $MASK_\tau$), zwolnij $KEEP_{\tau-1}$

  Ścieżka uwierzytelniająca dla liścia $\alpha + 1$ wymaga nowych prawych węzłów na wysokościach $h = 0, \ldots, \tau - 1$. Dla $h \leqslant H - k - 1$ owe węzły są trzymane w $TREEHASH_h$ i dla $h \doteq H - k$ w $RETAIN_h$:
  for ($h = 0$ **to** $\tau - 1$) **do**
    if ($h \leqslant H - k - 1$) **then**
      $AUTH_h \leftarrow TREEHASH_h.node$
    else
      $AUTH_h \leftarrow RETAIN_h.pop()$

  Na wysokościach $min\{\tau - 1, H - k - 1\}$ instancje $TREEHASH$ muszą zostać powtórnie zainicjowane. Każdy obiekt $TREEHASH_h$ zostaje zainicjowany z indeksem $\alpha + 1 + 3 \cdot 2^h$, o ile jest on mniejszy niż 2^H :
  for ($h = 0$ **to** $min\{\tau - 1, H - k - 1\}$) **do**
    if ($\alpha + 1 + 3 \cdot 2^h < 2^H$) **then**
      $TREEHASH_h.initialize(\alpha + 1 + 3 \cdot 2^h < 2^H), SEEDNEXT_h)$

8 Następnie wykonywanych jest do $\frac{H-k}{2}$ aktualizacji instancji $TREEHASH$, aby na czas obliczyć węzły potrzebne do przyszłych ścieżek uwierzytelniających :
for (1 **to** $\frac{H-k}{2}$) **do**
Rozważamy tylko obiekty $TREEHASH$, które są zainicjowane i niezakończone. Niech s będzie indeksem instancji $TREEHASH$, której najmniejszy trzymany węzeł ma najmniejszą wysokość z wszystkich pozostałych, branych pod uwagę, instancji. W razie, gdyby było kilka takich obiektów, brana jest instancja z najmniejszym indeksem:
$s \leftarrow min\{h : TREEHASH_h.height() = min_{j=0,\ldots,H-k-1}\{TREEHASH_j.height()\}\}$
$TREEHASH_s.update()$

9 Return $AUTH_0, \ldots, AUTH_{H-1}$ (ścieżka uwierzytelniająca dla liścia o indeksie $\alpha + 1$);

Uogólniając (dokładniejszy opis algorytmu znajduje się w pracy [1]), powyższe rozwiązanie problemu znajdywania ścieżki uwierzytelniającej, opiera się na przetrzymywaniu niektórych przydatnych węzłów w pamięci oraz balansowaniu kosztu obliczeń pozostałych potrzebnych węzłów. Wysokość τ oznacza poziom, na którym ścieżka uwierzytelniająca dla $\alpha + 1$ potrzebuje nowego (względem ścieżki α) lewego węzła uwierzytelniającego oraz nowych prawych węzłów na poziomach $0, \ldots, \tau - 1$. Lewy węzeł jest pozyskiwany na podstawie wartości zapamiętanych w stanie algorytmu, a prawe są wyliczane w instancjach $TREEHASH$, bądź odczytywane z $RETAIN$, jeśli dany węzeł akurat został zapisany. W ten sposób, dzięki wykorzystywaniu wcześniej obliczonych węzłów i rozkładaniu obliczeń na kilka iteracji, koszty algorytmu są zbalansowane oraz zminimalizowane.

analiza złożoności

Algorytm w każdej iteracji musi zaktualizować $H - k$ wartości $SEEDNEXT$, co wymaga $H-k$ wywołań funkcji pseudolosowej. Ponadto wykonanych jest $\frac{H-k}{2}$ aktualizacji obiektów $TREEHASH$, co ma złożoność $O(\frac{H-k}{2} \cdot (F \cdot (l \cdot w + 2) + l \cdot S))$, gdzie F to koszt wywołania funkcji pseudolosowej, a S to koszt wywołania funkcji haszującej. Sumaryczna złożoność czasowa to $O(\frac{H-k}{2} \cdot (F \cdot (l \cdot w + 4) + l \cdot S))$ dla każdej iteracji.

Jeżeli chodzi o pamięć, algorytm musi w swoim stanie mieć zapisane wszystkie struktury danych wymienione na wstępie tej sekcji. Zgodnie z obliczeniami przeprowadzonymi w pracy [1], algorytm potrzebuje przetrzymywać w najgorszym przypadku $3 \cdot H + \lfloor \frac{H}{2} \rfloor - 3 \cdot k - 2 + 2^k$ węzłów, do czego dochodzi koszt trzymania $2 \cdot (H - k)$ n-bitowych ziaren do generatora pseudolosowego. Jako, że w opisywanej implementacji węzły zajmują n-bitów, złożoność pamięciowa to $O((5 \cdot H + \lfloor \frac{H}{2} \rfloor - 5 \cdot k - 2 + 2^k) \cdot n)$ bitów.

3.4 Tworzenie struktury oraz kluczy

implementacja

Tworzenie struktury hierarchii drzew, służącej do generowaniu podpisów, według schematu opisywanego w tej pracy, w zasadzie jest równoważne z wygenerowaniem pary kluczy publicznego i prywatnego. Na początku podane są parametry m,n oraz wysokoś dolnego oraz górnego drzewa (wraz z parametrami w i k dla każdego). Wybrane losowo oraz dodane do klucza publicznego zostają główna maska, maska dla l-drzewa oraz n-bitowy parametr X. Kolejnym krokiem jest losowe wybranie początkowych stanów (ziaren) generatorów

$FSgen$ dla górnego, dolnego i następnego dolnego drzewa oraz zapisanie otrzymanych wartości do klucza prywatnego. Następnie należy wygenerować korzenie górnego i dolnego drzewa (nie następnego), poprzez iterowanie po kolejnych stanach generatorów i wywoływanie algorytmu $Treehash$, tym samym budując drzewa od lewej do prawej. Podczas wykonywania tej operacji, należy również zapamiętywać węzły potrzebne do zainicjowania obiektów algorytmu znajdywania ścieżki uwierzytelniającej (opis węzłów potrzebnych do inicjacji w 3.3). Otrzymany korzeń górnego drzewa należy zapisać w kluczu publicznym, a korzeń dolnego podpisać za pomocą pierwszego liścia górnego drzewa i tak otrzymany podpis zapisać w kluczu prywatnym. Przed złożeniem podpisu, należy jednak zainicjować obiekty algorytmu znajdywania ścieżek uwierzytelniających dla górnego i dolnego drzewa, a następnie zapisać aktualne stany tych algorytmów w kluczu prywatnym. Oprócz tego, należy dodać do klucza prywatnego instancję algorytmu $TREEHASH_{next}$, która jest odpowiedzialna za budowanie następnego drzewa dolnego (budowa dokonuje się podczas wykorzystywania kolejnych liści aktualnego drzewa dolnego). W ten sposób wygenerowane zostają klucze schematu i algorytm jest gotowy do składania podpisów.

analiza złożoności

Podczas generowania kluczy, koszt czasowy determinowany jest przez wyliczanie korzeni drzew górnego oraz dolnego. Zgodnie z analizą złożoności przeprowadzonej w sekcji 3.2, zajmuje to $O((2^{H_u} + 2^{H_l}) \cdot (F \cdot (l \cdot w + 2) + l \cdot S) - 2 \cdot S)$, gdzie S to koszt użycia funkcji haszującej, a F koszt użycia funkcji pseudolosowej. Dochodzi do tego pomijalny czas podpisywania dolnego drzewa przez górne $O(F \cdot w_u \cdot l_u \cdot (1 + \frac{H_u - k_u}{2}) + 2 \cdot (H_u - k_u) + l_u \cdot S)$ (operacja ta jest dokładniej opisana w sekcji 3.5).

Koszt pamięciowy przedstawia się następująco. Klucz publiczny przetrzymuje na n bitach korzeń górnego drzewa, na $max(H_u,H_l) \cdot n$ bitach główną maskę, na $\lceil log(l) \rceil \cdot n$ bitach maskę l-drzewa oraz na n bitach parametr X, co sumarycznie daje $(2 + max(H_u,H_l) + \lceil log(l) \rceil) \cdot n$ bitów zajętej pamięci. Klucz prywatny natomiast trzyma trzy n-bitowe ziarna do generatorów $FSgen$, dwa stany algorytmu obliczania węzłów uwierzytelniających - $O((5 \cdot H + \lfloor \frac{H}{2} \rfloor - 5 \cdot k - 2 + 2^k) \cdot n)$ bitów ($H = H_u$ dla górnego drzewa i $H = H_l$ dla dolnego drzewa), jeden stan algorytmu $Treehash$ -$O(\lceil log(l_l) \rceil \cdot n)$ bitów i podpis cyfrowy dolnego drzewa. Podpis cyfrowy dolnego drzewa posiada ścieżkę uwierzytelniającą górne drzewo - $O(H_u \cdot n)$ bitów, sygnaturę wiadomości - $l_u \cdot n$ bitów oraz indeks drzewa. Klucz prywatny waży zatem $O(((5 \cdot H_l + \lfloor \frac{H_l}{2} \rfloor - 5 \cdot k_l - 2 + 2^{k_l}) + (5 \cdot H_u + \lfloor \frac{H_u}{2} \rfloor - 5 \cdot k_u - 2 + 2^{k_u}) + H_u + l_u + \lceil log(l_l) \rceil) \cdot n)$ bitów.

3.5 Składanie podpisów

implementacja

Algorytm składania podpisów na wejściu dostaje wiadomość M, klucz prywatny schematu oraz indeks i użytego liścia z dolnego drzewa. W kluczu prywatnym znajduje się ziarno do generatora $FSGen$, który należy wywołać, tym samym aktualizując go i pozyskując ziarno R_i. Za pomocą tej wartości, zgodnie ze schematem Winternitza opisanym w 2, z wiadomości M zostaje obliczona wartość δ, która jest częścią podpisu cyfrowego. Następną częścią jest ścieżka uwierzytelniająca dolnego drzewa, wyliczana za pomocą instancji algorytmu opisanego w sekcji 3.3, którego aktualny stan (mający już gotową ścieżkę

dla i-tego liścia) trzymany jest w kluczu prywatnym. Należy wywołać ten algorytm, aby miał aktualny stan dla następnego liścia $i+1$. Oprócz tego w podpisie cyfrowym znajduje się podpis dolnego drzewa (również trzymany w kluczu prywatnym), składający się analogicznie z indeksu użytego górnego liścia, górnej ścieżki uwierzytelniającej oraz wartości δ otrzymanej z korzenia dolnego drzewa.

Ponieważ opisywany w tej pracy schemat, w odróżnieniu od $XMSS+$, umożliwia dołączenie kolejnej hierarchicznej struktury w przypadku zużycia wcześniejszych, do podpisu dołączona jest jeszcze lista trzymająca podpisy kolejnych dodanych struktur, złożonych przez poprzedzające struktury. W ten sposób utworzona jest hierarchia struktur (i ich podpisów), przez którą w razie weryfikacji trzeba przejść, zaczynając od najnowszej struktury, a kończąc na korzeniu pierwszej, będącego częścią klucza publicznego całego schematu. Samo zastępowanie starszej struktury przez nowszą, polega na podpisaniu ostatnim liściem górnego drzewa, korzenia nowo wygenerowanej, dwupoziomowej struktury. Procedura może się powtarzać dowolną ilość razy, co jednak wydłuża czas weryfikacji z powodu konieczności sprawdzenia wszystkich podpisów.

Ważną czynnością wykonywaną podczas generowania podpisów, jest budowanie przyszłego dolnego drzewa, które ma zastąpić aktualne, w razie jego zużycia. Ponieważ schemat zakłada, iż każde następne poddrzewo ma wysokość o 1 większą niż poprzednie (a więc dwa razy więcej liści), na każdy zużyty liść (każdy złożony podpis), wywołane są dwie iteracje instancji $TREEHASH_{next}$ znajdującej się w kluczu prywatnym. W ten sposób na jeden podpis generowane są dwa następne liście, co pozwala ukończyć drzewo na czas i zbudować je w sposób zbalansowany. Kiedy ostatni liść dolnego drzewa jest użyty do podpisu, następuje podmiana dolnych drzew, poprzez podpisanie nowego dolnego korzenia (który znajduje się w instancji $TREEHASH_{next}$). Inicjowana jest również nowa instancja algorytmu obliczania ścieżek uwierzytelniających dolnego drzewa oraz przygotowana zostaje nowa instancja $TREEHASH_{next}$, która będzie budować kolejne poddrzewo.

analiza złożoności

Czas złożenia podpisu zależy głównie od czasu wytworzenia wartości δ zgodnie ze schematem Winternitza, oraz od wykonania iteracji algorytmu obliczania ścieżki uwierzytelniającej. Koszt obliczenia δ wynosi $O(F \cdot w \cdot l)$, gdzie F to koszt wywołania funkcji pseudolosowej, a w to parametr Winternitza. Koszt wykonania iteracji obliczania ścieżki uwierzytelniającej wynosi $O(\frac{H-k}{2} \cdot (F \cdot (l \cdot w + 4) + l \cdot S))$, gdzie S jest kosztem wywołania funkcji haszującej. Zatem sumaryczna złożoność to $O(F \cdot w \cdot l \cdot (1 + \frac{H-k}{2}) + 2 \cdot (H-k) + l \cdot S)$.

Generator musi posiadać klucz prywatny (opis zajmowanej pamięci w sekcji 3.4) oraz ewentualną listę podpisów kolejnych dwupoziomowych struktur. Podpis cyfrowy zawiera ścieżki uwierzytelniające dla drzewa górnego i dolnego, które zajmują $((H_u + H_l) \cdot n)$ bitów. Posiada też wartości δ dla wiadomości oraz korzenia dolnego drzewa, których całościowy rozmiar wynosi $n \cdot (l_u + l_l)$ bitów (l_u to l górnego drzewa, l_l to l dolnego drzewa) oraz ewentualną listę podpisów kolejnych dwupoziomowych struktur, których rozmiar jest analogiczny. Podpis cyfrowy wiadomości wymaga zatem $n \cdot (l_u + l_l + H_u + H_l)$ bitów (lub odpowiednio więcej, jeśli dwupoziomowa struktura była zastępowana przez kolejne, nowsze struktury).

3.6 Weryfikacja podpisów

implementacja

W celu zweryfikowania wiadomości M, należy na wejściu podać podpis cyfrowy oraz klucz publiczny schematu. W podpisie cyfrowym znajduje się wartość δ, z której należy wyliczyć, sposobem opisanym w rozdziale 2, klucz publiczny `W-OTS`. Mając klucz, za pomocą algorytmu $leafcalc$, zostaje otrzymany liść dolnego drzewa użyty do podpisu. Znając indeks liścia oraz ścieżkę uwierzytelniającą dolnego drzewa (dane te są w podpisie), możliwe jest obliczenie korzenia drzewa (wzór znajduje się w rozdziale 2). Otrzymany tak korzeń należy potraktować analogicznie do wiadomości M, korzystając z podpisu cyfrowego dolnego drzewa, który jest zagnieżdżony wewnątrz podpisu cyfrowego wiadomości. Powtarzając powyższe czynności, otrzymany jest korzeń górnego drzewa, który, o ile użyta została tylko jedna struktura, powinien być równy korzeniowi podanemu w kluczu publicznym schematu. Jeżeli wartości są różne, podpis nie jest prawidłowy i wiadomość nie jest wiarygodna.

W przypadku, gdy do złożenia podpisu zostało użytych kilka dwupoziomowych struktur, w podpisie cyfrowym znajduje się lista podpisów cyfrowych kolejnych struktur. Wtedy, po otrzymaniu górnego korzenia, należy ponownie powtórzyć powyższe czynności weryfikacyjne, tym razem korzystając z kolejnych podpisów cyfrowych znajdujących się liście. Dopiero po przejściu przez wszystkie podpisy, otrzymany jest korzeń pierwotnej struktury, który może być porównany z wartością trzymaną w kluczu publicznym.

analiza złożoności

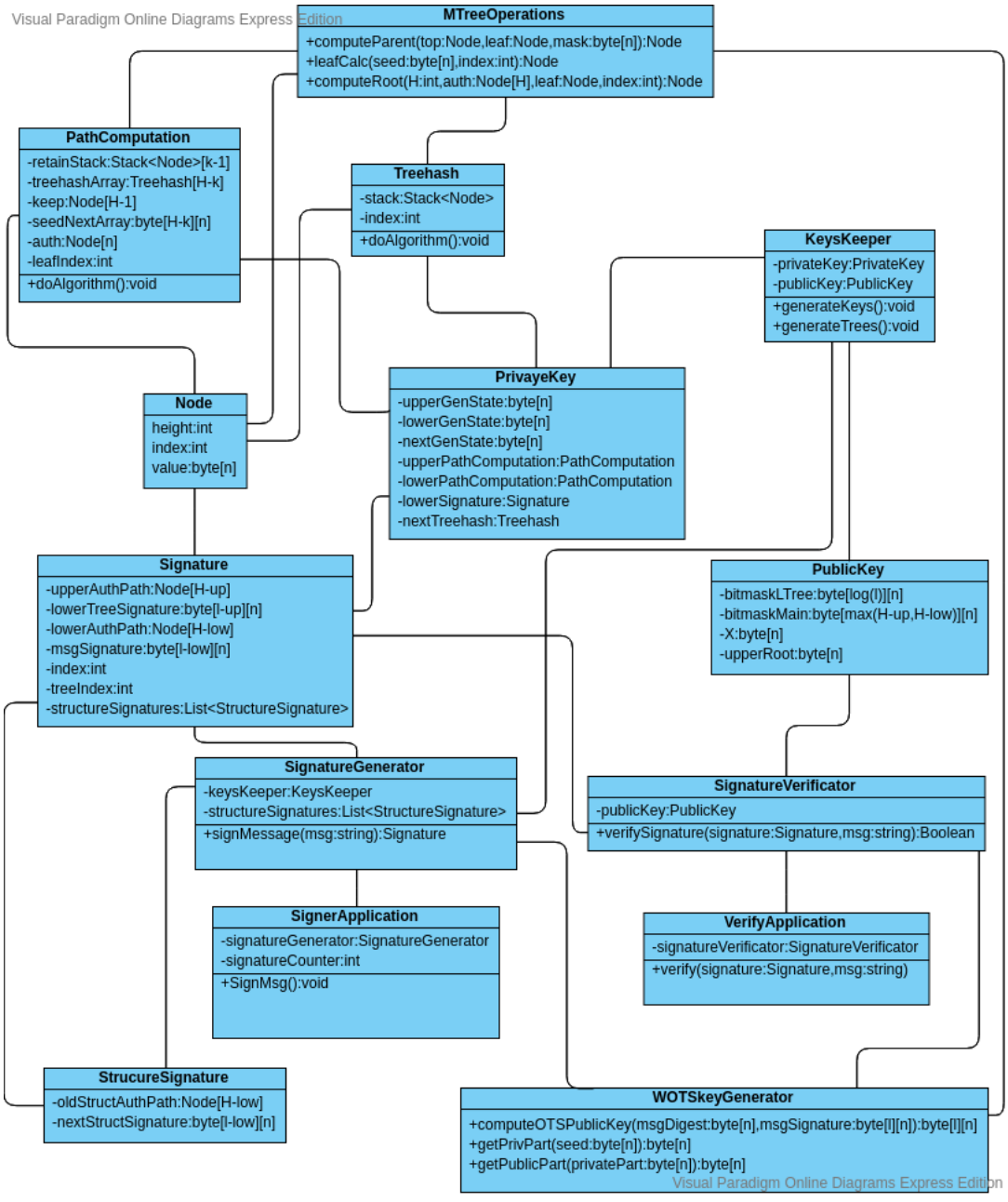
Czas weryfikacji podpisu wymaga $(H_u + H_l)$ wywołań funkcji haszującej w trakcie obliczania górnego korzenia, za pomocą ścieżek uwierzytelniających, oraz dwukrotnego (dla górnego i dolnego drzewa) $O(w \cdot l)$-krotnego wywołania funkcji pseudolosowej, w celu obliczenia klucza publicznego `W-OTS`. Ponadto dochodzi dwukrotny koszt obliczania liści z kluczy publicznych `W-OTS`, wynoszący $2 \cdot O(2^{\lceil log(l) \rceil} - 1)$ wywołań funkcji haszującej. Otrzymana złożoność wynosi zatem $O((w_l \cdot l_l + w_u \cdot l_u) \cdot F + (2^{\lceil log(l_l) \rceil} + 2^{\lceil log(l_u) \rceil} - 2) \cdot S)$, gdzie indeks l oznacza parametr dla drzewa dolnego, indeks u parametr dla drzewa górnego, F koszt wywołania funkcji pseudolosowej, a S koszt wywołania funkcji haszującej.

Wymagana pamięć weryfikatora ogranicza się do pamięci, wystarczającej do przetrzymywania klucza publicznego, czyli $(2 + max(H_u, H_l) + \lceil log(l) \rceil) \cdot n$ bitów.

Implementacja systemu

4.1 Opis implementacji

Budowę systemu przedstawia dokładniej poniższy diagram klas 4.1:



Rysunek 4.1: Diagram klas schematu podpisów

Implementacja schematu `XMSS+` oraz schematu o rosnących dolnych drzewach została wykonana w Javie. Jest to język wysokopoziomowy, więc w przypadku, w którym wydajność algorytmu jest krytyczna (np. karty chipowe), warto pomyśleć o wykorzystaniu języków niskopoziomowych. Cały system złożony jest z aplikacji składającej podpisy oraz z aplikacji weryfikującej.

Aplikacja podpisująca posiada klucz prywatny, który ze względów bezpieczeństwa, nie wychodzi w żaden sposób poza aplikację (więc w razie jej restartu - jest tracony). Udostępnia ona klucz publiczny i informacje o parametrach algorytmu oraz generuje podpisy cyfrowe dla danych plików tekstowych.

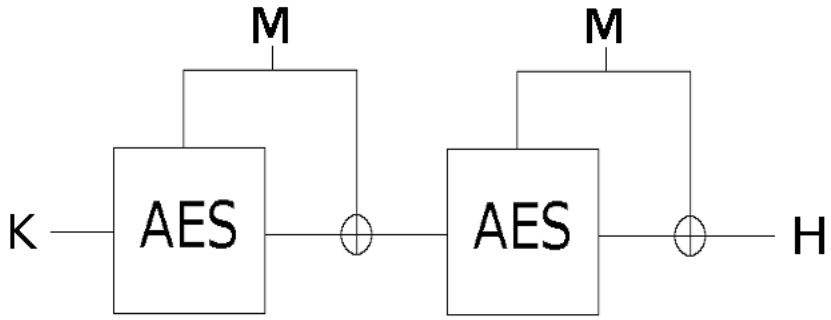
Aplikacja weryfikująca sprawdza, czy podpis cyfrowy, dany wraz z odpowiadającym mu plikiem tekstowym, jest wiarygodny. W tym celu wykorzystuje udostępniony klucz publiczny i parametry algorytmu.

Jak widać na diagramie klas 4.1, poszczególne obiekty odpowiadają, opisanym we wcześniejszych rozdziałach, algorytmom i strukturom. W celu uproszczenia powyższego UML, pominięto przetrzymywanie i przekazywanie parametrów m,n,l,k,w, masek oraz definicje funkcji pomocniczych. Wszystkie klasy i zawarte w nich metody zostały pokryte testami jednostkowymi. Ponadto utworzona została klasa wykonująca testy wydajnościowe, które są opisane w rozdziale 5.

W opisywanym schemacie kluczowym aspektem jest implementacja funkcji pseudolosowej oraz funkcji haszującej, na których opiera się złożoność algorytmów. Obie funkcje zostały zaimplementowane zgodnie z wyborem i zaleceniami autorów pracy [3].

Rodzina funkcji pseudolosowych jest wyrażona za pomocą szyfru blokowego AES (ang. Advanced Encryption Standard), ze względu na jego bezpieczeństwo, dużą szybkość działania oraz niskie zużycie pamięci. Wybór funkcji pseudolosowej, należącej do tej rodziny, jest jednoznaczny z wyborem n-bitowego klucza do szyfru blokowego AES (rodzina zawiera więc 2^n funkcji pseudolosowych).

Funkcja haszująca została otrzymana, przy użyciu konstrukcji `Merkle-Darmgarda`, która iteruje po kolejnych blokach wiadomości, stosując na nich jednokierunkową funkcję skrótu. Owa funkcja skrótu została zbudowana za pomocą metody `Matyasa-Meyera-Oseasa`, która przy pomocy szyfru blokowego i operacji XOR zwraca n-bitowy skrót z $2{\cdot}n$-bitowego ciągu.



Rysunek 4.2: konstrukcja funkcji haszującej

W powyższym rysunku 4.2, przedstawiona jest, wykorzystywana w implementacji podpisów cyfrowych, funkcja haszująca $H_k : \{0,1\}^{2 \cdot n} \rightarrow \{0,1\}^n$. Pierwszy n-bitowy blok wejściowego $2 \cdot n$-bitowego ciągu jest szyfrowany za pomocą AES z kluczem k. Otrzymany szyfrogram oraz pierwszy blok M poddawany jest operacji XOR i tak uzyskany n-bitowy ciąg staje się kluczem dla kolejnego szyfru blokowego. W sposób analogiczny, drugi blok jest szyfrowany za pomocą AES i poddawany operacji XOR. Tak utworzony zostaje hasz dla wejściowego $2 \cdot n$-bitowego ciągu.

Konstrukcja `Matyasa-Meyera-Oseasa` jest nieznacznie wolniejsza od konkretnych implementacji funkcji skrótu, jednak dzięki jej wykorzystaniu, wystarczającym jest zaimplementowanie bloku szyfrowego (AES), który zapewnia zarówno funkcje pseudolosowe jak i haszujące. Powoduje to oszczędność pamięciową, która jest niezwykle ważna dla niewielkich systemów wbudowanych, z myślą o których zaprojektowany jest opisywany schemat podpisów cyfrowych.

4.2 Komunikacja ASN.1

W tym podrozdziale dokładniej opisany zostanie sposób komunikacji między aplikacjami. Użyta w tym celu jest notacja `ASN.1` (ang. Abstract Syntax Notation One).

ASN.1 jest standardem służącym do opisu struktur przeznaczonych do reprezentacji, kodowania, transmisji i dekodowania danych. Notacja została wybrana ze względu na jej powszechność, niezależność od urządzeń/środowiska/języka programowania, precyzyjność oraz odpowiedni balans między zawartością informacji, a jej czytelnością.

Jako, że ASN.1 określa jedynie składnię przetrzymywania informacji, konieczny jest wybór kodowania, które pozwoli przekształcić kompatybilną z ASN.1 strukturę w ciąg bitów. W pracy zostało użyte kodowanie `BER` (ang. Basic Encoding Rules). Kodowanie to opiera się na trzech podciągach, opisujących kolejno `tag`, `długość` i `wartość` (tag identyfikuje typ określony w notacji ASN).

Aplikacja podpisująca, po uruchomieniu i wygenerowaniu kluczy, tworzy dwa pliki - *publicKey.txt* i *params.txt*, zawierające zakodowane za pomocą `BER` wartości. Wartości te są nośnikiem, reprezentowanych w strukturach `ASN.1`, klucza publicznego oraz zbioru parametrów. Analogicznie, każda generacja podpisu cyfrowego kończy się utworzeniem pliku *signature.txt*, w którym znajduje się zakodowany podpis cyfrowy, który wcześniej został odpowiednio dostosowany do notacji `ASN.1`.

Aplikacja weryfikująca, odczytuje klucz publiczny, parametry z plików oraz pożądane do weryfikacji podpisy, znajdujące się pod podaną ścieżką. Dzięki uniwersalności `ASN.1`, proceder ten jest bardzo elastyczny (np. aplikacje mogłyby zostać napisane w innych językach programowania lub działać na zupełnie różnych urządzeniach fizycznych).

Do utworzenia struktur `ASN.1` oraz kodowania/dekodowania `BER` użyta została biblioteka `bouncycastle` w Javie. Na następnej stronie przedstawiony jest sposób reprezentacji podpisu cyfrowego oraz klucza publicznego w `ASN.1`.

```
Signature-Schema DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
  Signature ::= SEQUENCE
  {
     upperAuthPath            SEQUENCE OF Node,
     lowerTreeSignature       SEQUENCE OF OCTET STRING,
     lowerAuthPath            SEQUENCE OF Node
     msgSignature             SEQUENCE OF OCTET STRING,
     index                    INTEGER,
     treeIndex                INTEGER,
     structureSignatures      SEQUENCE OF StructureSignature
  }

  Node ::= SEQUENCE
  {
     height          INTEGER,
     value           OCTET STRING,
     index           INTEGER
  }

  StructureSignature ::= SEQUENCE
  {
     oldStructAuthPath           SEQUENCE OF Node,
     nextStructSignature         SEQUENCE OF OCTET STRING
  }

  PublicKey ::= SEQUENCE
  {
     bitmaskLTree       SEQUENCE OF OCTET STRING,
     bitmaskMain        SEQUENCE OF OCTET STRING,
     x                  OCTET STRING,
     upperRoot          OCTET STRING
  }

  Parameters ::= SEQUENCE
  {
     m                  INTEGER,
     n                  INTEGER,
     upperH             INTEGER,
     lowerH             INTEGER,
     wU                 INTEGER,
     wL                 INTEGER,
     treeGrowth         INTEGER,
     hashFunctionKey    OCTET STRING
  }
END
```

W powyższej notacji :

- `SEQUENCE` - oznacza sekwencję wartości trzymanych w pojedynczej strukturze,
- `SEQUENCE OF` - odpowiada tablicy typu danych, który jest podany bezpośrednio po tej deklaracji,
- `OCTET STRING` - oznacza ciąg bajtów,
- `INTEGER` - jest typem liczb całkowitych,
- `Node,Signature,PublicKey,StructurSignature,Parameters` - oznaczają dodatkowo utworzone struktury danych.

Przykładowo, dla pewnego wywołania aplikacji i dla wcześniej już przedstawionej struktury `ASN.1` *Parameters* :

```
Parameters ::= SEQUENCE
{
      m                   INTEGER,
      n                   INTEGER,
      upperH              INTEGER,
      lowerH              INTEGER,
      wU                  INTEGER,
      wL                  INTEGER,
      treeGrowth          INTEGER,
      hashFunctionKey     OCTET STRING
}
```

w pliku *params.txt* znajduje się poniższy ciąg zapisany w bazie o podstawie 64 :

```
MIACASACARACAQYCAQcCAQgCAQgCAQEkgAQQEOcCEjuwuvHbek9OHHdnQwAAAAA=
```

z którego, po zdekodowaniu i otrzymaniu poniższej struktury, można odczytać wartości poszczególnych parametrów:

```
SEQUENCE (8 elem)
      INTEGER 32
      INTEGER 16
      INTEGER 6
      INTEGER 7
      INTEGER 8
      INTEGER 8
      INTEGER 1
      OCTET STRING (1 elem)
            OCTET STRING (16 byte) 10E702123BB0BAF1DB7A4F4E1C776743
```

Proceder jest analogiczny w przypadku odczytywania danych z plików *publicKey.txt* oraz *signature.txt*.

Testy wydajnościowe

5.1 Wprowadzenie do testów

W celu sprawdzenia efektywności opisywanego w tej pracy schematu, zostały przeprowadzone testy wydajnościowe dla różnych konfiguracji wysokości górnego i dolnego drzewa. W każdym przypadku testowym została przyjęta funkcja skrótu `SHA3(256-bitowa)`, ustalając tym samym parametr oznaczający ilość bajtów w szyfrowanym ciągu na $m = 32$ oraz długość pojedynczego bloku na $n = 128$ bitów.

Parametr k (definiujący ilość przetrzymywanych w pamięci prawych węzłów), pozwala na wymianę zajmowanej pamięci klucza prywatnego, na przyśpieszenie wykonania algorytmu obliczania węzłów uwierzytelniających. Należy mieć na uwadze, iż wedle pracy [3], od pewnego momentu zwiększanie k nie poprawia już czasów generowania podpisów. Parametr Winternitza w natomiast definiuje bazę systemu liczbowego, w którym wiadomość jest przeliczana na wartość δ (czym jednocześnie determinuje l, czyli długość kluczy Winternitza oraz δ). W praktyce współczynnik w powoduje zmniejszenie wielkości podpisu cyfrowego o $log(w)$, zaś zwiększenie czasu generowania kluczy, podpisywania i weryfikowania. Autorzy [3] zauważyli, że wzrost czasu algorytmów dla małych w jest pomijalny, jednak dla $w > 16$ jest już liniowy, co jest spowodowane ilością i długością łańcuchowego wykonywania funkcji pseudolosowej przy generowaniu kluczy jednorazowych. Oprócz tego w [3] twierdzą, iż wysokie w zmniejsza bezpieczeństwo kluczy Winternitza, co również jest warte uwagi podczas wybierania wartości owego współczynnika. Parametry k i w zostały ustalone, zgodnie z wnioskami, otrzymanymi po wykonaniu eksperymentów w [3], na wartości $k = 4$ dla drzew o parzystej wysokości, $k = 3$ dla drzew o nieparzystej wysokości i $w = 16$ (zarówno dla górnych jak i dolnych drzew).

Przeprowadzone zostały trzy rodzaje testów (wymienione poniżej), gdzie dla testów 2 i 3 porównywane są wyniki schematu `XMSS+` [3] z wynikami schematu opisanego w [2], którego implementacja omawiana jest w tej pracy.

1. Sprawdzenie czasów weryfikacji dla małych drzew, gdy zostają dołączone do podpisu kolejne struktury

2. Porównanie czasu generowania struktur dla różnych rozmiarów drzew

3. Czasy generowania i weryfikowania podpisu dla drzew praktycznych rozmiarów

Wszystkie testy zostały wykonane na laptopie z linuksowym, 64-bitowym systemem operacyjnym dystrybucji Ubuntu 18.04.01 LTS, procesorem czterordzeniowym Intel Core i5-6300HQ @ 2.30 GH x 4 i z 8 GB pamięci RAM. Dla każdego czasu podanego w poniższych tabelach jednostką jest milisekunda [ms].

5.2 Wyniki testów i ich analiza

W pierwszym teście celem jest sprawdzenie, ile kosztuje usprawnienie wprowadzone względem `XMSS+`, polegające na tworzeniu opisanej w publikacji [2] hierarchii drzew, tzn. kiedy zostanie zużyta cała pojemność aktualnej struktury, to jest ona zastępowana przez nową, świeżo wygenerowaną i podpisaną strukturę (dzięki podpisowi klucz publiczny schematu nie musi się zmieniać).

Do zobrazowania tego kosztu, użyta została struktura o wysokości górnego drzewa $H_u = 3$ i dolnego drzewa $H_l = 2$. Taka konstrukcja posiada pojemność na $2^9 - 2^2 = 508$ podpisów. W próbie, której wyniki prezentują się poniżej, złożono 10000 podpisów. Wymusza to, dla ostatnich podpisów wiadomości, załączenia 19 dodatkowych podpisów struktur i tym samym przejścia ścieżek uwierzytelniających dla każdej z nich podczas weryfikacji.

Tablica 5.1: Dołączanie kolejnych struktur

struktura	śr. czas podpisu	ostatni podpis	śr. czas weryfikacji
1	2.867	105.0	0.332
2	2.477	99.0	0.57
3	2.359	92.0	0.516
4	2.387	93.0	0.637
5	2.836	95.0	0.918
6	2.879	116.0	1.09
7	2.477	95.0	1.09
8	2.5	94.0	1.137
9	2.668	108.0	1.539
10	2.406	93.0	1.449
11	2.559	106.0	1.621
12	2.465	101.0	1.715
13	2.473	104.0	1.844
14	2.375	94.0	1.824
15	2.352	98.0	1.961
16	2.449	99.0	2.195
17	2.398	93.0	2.285
18	2.91	119.0	2.832
19	2.527	97.0	2.648

Jak widać w tabeli 5.1, przedstawione są trzy rekordy : `średni czas podpisu, ostatni podpis, średni czas weryfikacji`. `Ostatni podpis` oznacza tutaj czas wykonania podpisu, przy użyciu ostatniej, jednorazowej pary kluczy Winternitza. Taki podpis jest wyjątkowy, gdyż podczas jego składania generowana, podpisywana i podpinana jest kolejna struktura, z czego wynikają znacznie odbiegające od średniej czasy podpisów. Zaobserwować można, iż o ile czasy składania podpisów nie różnią się od siebie, to czas weryfikacji dla każdej kolejnej konstrukcji podnosi się średnio o około 0.12 ms, co stanowi ponad 1/3 czasu weryfikacji przy pojedyńczej strukturze. Dla podpisu zawierającego już 19 dwupoziomowych drzew, czas zwiększył się o około 800% w stosunku do podpisu z jednym dwupoziomowym drzewem.

W kolejnym teście przedmiotem badania jest czas generowania struktur, tzn. kluczy publicznego i prywatnego schematu. Porównano wyniki rozwiązania `XMSS+` oraz rozwią-

zania omawianego w tej pracy. Do porównania dobrano drzewa o podobnej pojemności (z pomijalnie małą różnicą) i w ten sposób ustanowiono cztery pary drzew przedstawione w tabeli 5.2.

Tablica 5.2: Czasy generowania struktur

pojemność	XMSS+			schemat z rosnącymi drzewami		
	H górne	H dolne	czas genr. [ms]	H górne	H dolne	czas genr. [ms]
2^{14}	7	7	354	3	7	76
2^{20}	10	10	794	4	5	17
2^{30}	15	15	20920	4	15	12434
2^{40}	20	20	654613	5	9	196

W tabeli 5.2 widać wyraźnie znacznie krótsze czasy generowania po stronie schematu z rosnącymi dolnymi drzewami. Szczególnie imponujący jest czas generowania dla pojemności 2^{40}, gdzie `XMSS+` jest gorszy aż 333-krotnie od drugiej konstrukcji. Ważną informacją jest fakt, że zbudowanie dwóch drzew o $H = 20$ zajęło 654613 ms $\approx$ 10 minut, z czego wynika, że budowanie wyższych drzew staje się trudne i mało praktyczne. Dla schematu o stałej wysokości dolnych drzew, czas ostro rośnie w zależności od wysokości (30-krotnie między wysokościami 10 - 15 i 15 - 20, można więc szacować czas generacji dla wysokości $H = 30$ na około 160 godzin). Co interesujące, w schemacie z rosnącymi drzewami, czas generacji niekoniecznie się zwiększa razem ze wzrostem pojemności (nawet obserwowane są bardzo duże dysproporcje). Dzieje się tak, ponieważ w tej konstrukcji mały wzrost wysokości drzewa górnego znacznie zwiększa ilość możliwych podpisów (bo istotnie zwiększa ilość dolnych drzew). W takim przypadku można ustalić niewielkie drzewa dolne, jeśli górne jest wystarczająco wysokie (warto zauważyć, że już wysokość $H_u = 5$ daje aż 31 rosnących drzew) i osiągnąć stosunkowo dużą pojemność do bardzo małego czasu generacji. Jest to okupione rosnącym czasem podpisywania wiadomości, który w momencie, gdy struktura zbliża się do limitu swojej pojemności, jest znacznie większy niż w `XMSS+`.

W trzecim teście ponownie poddane zostały porównaniu dotychczas omawiane schematy z prac [3] i [2], tym razem biorąc dwie struktury, którymi podpisano po 1000000 wiadomości. Przyjęto $H_u = 4$ i $H_l = 5$ dla rosnących drzew i $H_u = H_l = 10$ dla konstrukcji stałych dolnych drzew. Obie struktury mają pojemność $2^{20} - 2^{10}$ dla `XMSS+` i $2^{20} - 2^5$ dla drugiego ze schematów, co jest równe niewiele ponad 1000000 podpisom. Porównane zostały czasy podpisywania oraz weryfikowana w zależności, z którego dolnego drzewa uzyskane zostały klucze Winternitza, którymi podpisy zostały złożone. Rozpatrujemy wszystkie 14 zużyte drzewa dolne z rosnącej struktury oraz wybrane 14 drzew z 976 ($2^{10} \cdot 976 \approx 1000000$) zużytych w strukturze `XMSS+`. Podobnie jak w poprzedniej tabeli, `ostatni` oznacza ostatni liść zużytego drzewa, po którego wykorzystaniu podmieniane jest zużyte dolne drzewo na kolejne i następuje podpisanie nowego dolnego drzewa przez górne. Generacja następnego dolnego drzewa jest rozłożona na całe poprzednie drzewo, tzn. przy każdym podpisie generowane są dwa liście/jeden liść następnego drzewa (w zależności czy następne drzewo jest wyższe o 1, czy jak w `XMSS+` stałe).

Tablica 5.3: 1000000 podpisów [XMSS+] : $H_u = 10$, $H_l = 10$

nr drzewa	czasy podpisywania		czasy weryfikacji
	śr.	ostatni	śr.
całość	1.958	-	0.367
0	2.202	2.0	0.405
70	1.985	2.0	0.339
140	1.924	2.0	0.416
210	1.953	2.0	0.362
280	1.97	3.0	0.365
350	1.933	2.0	0.363
420	1.959	2.0	0.335
490	1.964	2.0	0.372
510	1.949	3.0	0.371
580	1.94	2.0	0.37
650	1.925	3.0	0.407
720	1.981	2.0	0.357
800	1.942	3.0	0.359
975	1.979	2.0	0.351

Tablica 5.4: 1000000 podpisów [drzewa rosnące] : $H_u = 4$, $H_l = 5$

nr drzewa	czasy podpisywania		czasy weryfikacji
	śr.	ostatni	śr.
całość	3.592	-	0.376
0	1.438	2.0	0.406
1	1.516	2.0	0.375
2	1.781	1.0	0.414
3	1.82	2.0	0.406
4	2.021	1.0	0.369
5	2.14	2.0	0.342
6	2.407	2.0	0.366
7	2.446	5.0	0.338
8	2.748	4.0	0.38
9	2.796	15.0	0.342
10	3.137	14.0	0.356
11	3.14	52.0	0.384
12	3.449	53.0	0.365
13	3.495	207.0	0.393

Po wynikach prezentowanych w powyższych tabelach 5.3 i 5.4, widać wyraźnie, iż średnie czasy składania podpisu są średnio ok. 1.8 razy dłuższe dla schematu o rosnących dolnych drzewach, niż dla schematu o stałych dolnych drzewach. Jest to cena za krótszy czas generacji (z tabeli 5.2 jest to kolejno 17 ms i 794 ms $\approx$ 46 razy szybciej). Średni całościowy czas podpisywania dla tabeli 5.4 może wydawać się podejrzany, ponieważ jest największą wartością spośród wyników z poszczególnych drzew. Jest tak, ponieważ w wypisanych 14 drzewach mieści się tylko 524255 podpisów. Pozostała część, która z racji rosnących kosztów generacji jest najbardziej czasochłonna i znacznie zwiększyła wartość całościowego czasu średniego, złożona jest w drzewie o indeksie 14, które nie jest uwzględnione w tabeli 5.4, ponieważ nie zostało do końca zużyte. Warto zauważyć, iż w `XMSS+` czasy podpisywania są takie same dla wszystkich drzew, a w drugim ze schematów, wyraźnie rosną wraz z indeksem dolnego drzewa (dla początkowych 3 indeksów szybciej o ok. 0.5 ms, niż w `XMSS+`, dla pozostałych wolniej maksymalnie o 1.5 ms), co jest spowodowane coraz wyższymi i trudniejszymi do generacji następnymi drzewami dolnymi. Interesującą rzeczą, jest rosnący czas podpisywania za pomocą ostatnich liści, dla końcowych drzew rosnących. Czasy zaczynają być wyższe, ponieważ po dodaniu ostatnich dwóch liści, algorytm obliczający korzeń nowego drzewa zaczyna być kosztowny, z powodu wysokości $H > 15$. Ponadto w tabelach 5.3 i 5.4 widać, iż czasy weryfikacji są praktycznie stałe dla wszystkich drzew w obu schematach.

Tabela 5.5 zawiera wyniki testu wydajnościowego drzewa o wysokościach $H_u = 6$ dla drzewa górnego i $H_l = 7$ dla dolnego. Struktura została wyróżniona w pracy, ponieważ jej parametry wydają się odpowiednimi dla efektywnego systemu składania i weryfikowania podpisów cyfrowych. Posiada ona ogromną pojemność $2^{70} - 2^7 \approx 10^{20}$, a początkowe wybory H dają podstawy, żeby sądzić iż klucze struktury oraz pierwsze podpisy będą generowane stosunkowo szybko (chociaż będą konsekwentnie rosnąć).

Tablica 5.5: 2097023 podpisów: $H_u = 6$, $H_l = 7$

nr drzewa	czasy podpisywania		czasy weryfikacji
	śr.	ostatni	śr.
całość	2.936	-	0.420
0	2.672	2.0	0.75
1	2.301	2.0	0.543
2	2.23	2.0	0.418
3	2.161	2.0	0.38
4	2.474	2.0	0.341
5	2.487	5.0	0.394
6	2.827	5.0	0.359
7	2.832	16.0	0.372
8	3.131	16.0	0.364
9	3.179	56.0	0.398
10	3.478	56.0	0.376
11	3.501	210.0	0.41
12	3.802	206.0	0.372
13	4.024	899.0	0.405

Jak widać w tabeli 5.5, w teście zużyto 14 pierwszy drzew struktury, tym samym wykonując 2097023 podpisów cyfrowych. Średni czas dla generowania podpisu to 2.93 ms, a dla weryfikowania to 0.420 ms. Czasy wydają się być zadowalające i praktyczne w kontekście wykorzystania schematu. Widoczną rysą algorytmu są szybko rosnące czasy generowania podpisu z wykorzystaniem ostatniego liścia z danego drzewa dolnego, które dla wyższych drzew są już wysokie (przy generacji poprzedzającej przejście na drzewo dolne o indeksie 14, czas ten wynosi niemal sekundę). W wynikach widać również charakterystyczny wzrost czasu generacji podpisów towarzyszący wzrostowi drzewa dolnego.

5.3 Wnioski

Po przeprowadzeniu testów wydajnościowych i przeanalizowaniu wyników, można stwierdzić, iż schemat opisywany w tej pracy stanowi interesującą alternatywę dla schematu `XMSS+`. Wybór rozwiązania powinien być dostosowany do zastosowania i jego wymagań. W środowisku, w którym czas generowania kluczy publicznych i prywatnych jest krytyczny (jak np. w przypadku kart chipowych, opisywanym w pracach [2] oraz [3], w którym parametry sprzętowe są mocno ograniczone), a stopniowe pogarszanie czasowe przy składaniu podpisów nie jest problematyczne, lepiej sprawdzi się konstrukcja z rosnącymi drzewami dolnymi. Podobnie w przypadku, gdy ciężko oszacować ilość podpisów, która zostanie złożona i dobór rozmiarów stałych drzew jest kłopotliwy. Wtedy lepiej postawić na bardziej elastyczny schemat drzew rosnących, dla którego duża pojemność nie wiąże się ze znaczącym zwiększeniem średniego czasu składania podpisów dla początkowych wiadomości, co niewątpliwie miałoby miejsce w przypadku zastosowania zbyt dużego drzewa `XMSS+`. Oprócz tego, w razie przekroczenia pojemności drzewa, w implementacji opisywanej w tej pracy oraz przez autorów [2], następuje wygenerowanie oraz dołączenie kolejnej struktury i umożliwienie dalszego podpisywania wiadomości (choć ze spowolnioną weryfikacją), co jest usprawnieniem w stosunku do `XMSS+` proponowanego w pracy [3].

XMSS+	proponowany schemat
- stały czas generowania podpisu	- rosnący czas generowania podpisu
- dłuższy czas generowania klucza prywatnego i publicznego schematu	- krótszy czas generowania kluczy, kosztem obciążenia generowania podpisów
- zwiększenie pojemności struktury wiąże się ze zwiększeniem czasu jej generowania oraz czasu składania podpisów	- schemat jest bardziej elastyczny, duża pojemność nie wymusza wydłużenia czasu generowania struktury, czy początkowych podpisów
- gdy struktura się zużyje, nie można składać więcej podpisów	- gdy struktura się zużyje, dołączana jest kolejna (co jednak spowalnia weryfikację)

Podsumowanie

Teoretyczny schemat opisany w pracy [2] okazał się skuteczny w praktycznej implementacji i dostarczył bezpieczne i efektywne narzędzie do składania i weryfikowania podpisów cyfrowych. Spełnione zostały wymagane cechy podpisów tzn. autentyczność, niezaprzeczalność i integralność, przy zachowaniu odpowiedniej szybkości i wymagań pamięciowych. Udało się również skutecznie wprowadzić uniwersalną i powszechną w informatyce komunikację między aplikacjami za pomocą notacji `ASN.1`. W kategorii schematów podpisów post-quantum, dzięki odmiennym zaletom, schemat o rosnących poddrzewach stanowi sensowną alternatywę dla `XMSS+`, czego dowiodły przeprowadzone testy wydajnościowe. Podobnie jak `XMSS+`, schemat o niezbalansowanych poddrzewach jest utajniony z wyprzedzeniem (ang. forward scecure), dzięki ciągłej aktualizacji klucza prywatnego, co wynika z wykorzystania generatora pseudolosowego (co również pozwala na trzymaniu w pamięci ziarna do generatora, zamiast wszystkich par jednorazowych kluczy `W-OTS`).

Co ważne w kontekście bezpieczeństwa schematu, opiera się ono na bezpieczeństwie funkcji pseudolosowej i haszującej. W implementacji opisanej w tej pracy, użyto szyfru blokowego `AES`, jednak w razie jego złamania/niespełnienia potrzeb, można łatwo zastąpić go innym algorytmem, który aktualnie jest uważany za bezpieczny i efektywny.

Warto podkreślić dalszą potrzebę badań w temacie podpisów cyfrowych, gdyż bezpieczeństwo komputerowe i kryptografia to sektory, które się ciągle zmieniają i wymagają stałej aktualizacji, w celu zapewnienia bezpieczeństwa użytkownikom. Dlatego należy podchodzić do różnych zagadnień na kilka sposobów, aby w razie kompromitacji jednego schematu, mieć w zanadrzu inny. Z tego właśnie powodu, mimo powszechności podpisów opartych na `RSA` czy `DSA`, omawiana jest równolegle alternatywa oparta na drzewach Merkla, która znalazła swoje zastosowanie w małych, wymagających wysokiej optymalizacji urządzeniach, takich jak karty chipowe.

Bibliografia

[1] J. A. Buchmann, E. Dahmen, M. Schneider. Merkle tree traversal revisited. PQCrypto 2008: 63-78.

[2] P. Błaśkiewicz, P. Kubiak, M. Kutyłowski. How to make operating systems for smart cards open. Bulgarian Cryptography Days 2012 Proceedings,ISBN 978-954-2946-22-9, pages 129-140.

[3] A. Hülsing, C. Busold, J. Buchmann. Forward secure signatures on smart cards. Part of the Lecture Notes in Computer Science book series (LNCS, volume 7707).

[4] A. Hülsing, E. Dahmen, J. Buchmann. Xmss - a practical forward secure signature scheme based on minimal security assumptions. Post-Quantum Cryptography, volume 7071 of Lecture Notes in Computer Science, pages 117–129. Springer Berlin / Heidelberg, 2011.

[5] R. C. Merkle. A certified digital signature. In CRYPTO '89: Proceedings on Advances in cryptology, volume 435 of Lecture Notes in Computer Science, pages 218–238. Springer-Verlag, 1989.

Zawartość płyty CD

Poniżej zostały wymienione ważniejsze foldery i pliki, które znajdują się w dołączonej płycie CD:

- Plik `W11_236441_2020_praca_inżynierska.pdf` - zawierający pracę dyplomową
- Folder `MerkleSignatureScheme` - zawiera kod źródłowy aplikacji podpisującej oraz aplikacji weryfikującej podpisy cyfrowe
  - folder `algorithm` - posiada kody źródłowe aplikacji i klas niezbędnych do działania algorytmu
    - folder `applications` - zawiera aplikację podpisującą oraz aplikację weryfikującą
      - plik `SignerApplication.java` - aplikacja generująca podpisy cyfrowe (posiada metodę `main`)
      - plik `VerifyApplicaton.java` - aplikacja weryfikująca podpisy cyfrowe (posiada metodę `main`)
    - folder `keys` - zawiera pakiet klas reprezentujących klucze oraz odpowiedzialnych za ich generację
    - folder `merkleTree` - zawiera pakiet klas implementujących drzewo Merkla i związane z nim operacje i struktury
    - folder `signing` - zawiera pakiet klas odpowiedzialnych za generowanie podpisów cyfrowych
    - folder `tools` - zawiera pakiet klas pomocniczych oraz definicje funkcji pseudolosowej, generatora, funkcji haszującej oraz obiektów ASN.1
    - folder `verification` - zawiera pakiet klas odpowiedzialnych za weryfikację podpisów cyfrowych
  - folder `tests` - tu się znajduje klasa przeprowadzająca testy wydajnościowe, klasa pomocnicza do rysowania wykresów i klasy testów jednostkowych
    - plik `Chart.java` - klasa odpowiedzialna za rysowanie wykresów
    - plik `PerformanceTest.java` - klasa przeprowadzająca testy wydajnościowe (posiada metodę `main`)
    - folder `unitTests` - przetrzymuje klasy zawierające testy jednostkowe klas z folderu `algorithm`

W celu uruchomienia aplikacji generującej oraz podpisującej, należy zbudować projekt oraz wywołać kolejno klasy `SignerApplication.java` oraz `VerifyApplicaton.java`. Można również skorzystać z załączonego pliku `MerkleSignatures.jar` i użyć poniższych komend (z poziomu folderu zawierającego plik `MerkleSignatureScheme`):

SignerApplication → `java -cp MerkleSignatureScheme/out/artifacts/MerkleSignatures_jar/MerkleSignatures.jar com.signature.scheme.algorithm.applications.SignerApplication`

VerifyApplicaton → `java -cp MerkleSignatureScheme/out/artifacts/MerkleSignatures_jar/MerkleSignatures.jar com.signature.scheme.algorithm.applications.VerifyApplication`