# Hash-basierte Signaturverfahren

Bachelorarbeit

*Bastian Brodd*

Matrikelnummer: 3127960

24. Oktober 2019

| | |
|---|---|
| Erstprüfer: | Prof. Dr. Heribert Vollmer |
| Zweitprüfer: | Dr. Arne Meier |
| Betreuer: | Dr. Maurice Chandoo |

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die wörtlich oder inhaltlich aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

| | |
|---|---|
| Ort, Datum | Unterschrift |

**Zusammenfassung:**
In dieser Arbeit wird die Familie der Hash-basierten Signaturverfahren vorgestellt. Diese gehören nicht zu den aktuell effizientesten Verfahren, bieten dafür aber ein alternatives mathematisches Problem als Basis ihrer Sicherheit. Gerade bei konstanten Fortschritten des Brechens unserer besten Verfahren, sowie der Entwicklung von Quanten-Computern, welche die Lösung unserer effizientesten Basisprobleme trivialisieren könnten, entspricht diese alternative Basis einem vielversprechenden Forschungsansatz. Wir beginnen mit möglichen Motivationen für Post-Quanten Kryptographie, gefolgt von der Vorstellung grundlegender Einmalsignaturverfahren. Im weiteren Verlauf der Arbeit erweitern wir diese zu einem nutzbaren Mehrfachsignaturverfahren und stellen über die Jahre entwickelte Verbesserungen vor.

# Inhaltsverzeichnis

# 1 Einleitung

Es ist unbestreitbar, dass die Vernetzung der Welt einer der zentralsten Punkte unseres heutigen Lebens geworden ist. Sie ermöglicht sofortige Kommunikation zweier Personen, egal wo sich diese auf unserem Planeten befinden. Des Weiteren basiert ein riesiger Teil unserer heutigen Wirtschaft auf der Rechenleistung von Computern, auf welche in steigenden Maßen über das Internet zugegriffen wird. Ganz abgesehen von dem Überfluss an Geräten des Internet der Dinge, welche sich heutzutage überall befinden.
Jedes Jahr scheint die Vernetzung weiter in unsere Leben einzuwandern und eine Welt ohne sie scheint mittlerweile nicht mehr vorstellbar. Während die meisten Personen ohne einen Gedanken, wie dieses Wunder unserer Zeit funktioniert, ihren Tagesabläufen nachgehen, wissen wir, dass sich hinter nahezu jeder simplen Aktion weitaus mehr verbirgt, als es scheint. Es ist von größter Wichtigkeit, dass sicher nachgewiesen werden kann, von wem eine Nachricht ursprünglich stammt und dass diese nicht während der Übermittlung verändert wurde. Ohne diese Garantien könnten wir das Netz für nichts benutzen, was konkrete Konsequenzen hat, da wir keiner Nachricht vertrauen könnten. Es wäre für einen Angreifer sehr einfach uns zu hintergehen. Speziell benötigen wir die Authentizität, Integrität sowie Nichtabstreitbarkeit einer Nachricht. Hierzu existieren seit geraumer Zeit digitale Signaturverfahren, welche sich über viele Revisionen in ihre heutige komplexe Form entwickelt haben. Die effizientesten als sicher angenommenen Versionen wären RSA [14], DSA [6] und ECDSA [9]. Die Sicherheit dieser drei Verfahren beruht zu einem großen Teil auf dem Problem der Faktorisierung oder dem Problem der Berechnung diskreter Logarithmen. Über die Jahre gab es jedoch geraume Fortschritte in der Lösung dieser Probleme, wodurch die Schlüssellängen immer wieder vergrößert werden mussten[3, S.63]. Zusätzlich wurden bereits große Fortschritte in der Entwicklung moderner Quanten-Computer erzielt. Steigende Kapazität dieser führt dazu, dass die Nutzung bereits bekannter Algorithmen von Shor [15] und Grover [7] nicht mehr weit entfernt scheint[3, S.63]. Mit diesen können RSA, DSA und ECDSA in linearer Zeit angegriffen werden [3, S.63], wodurch diese für unsere Zwecke unbrauchbar werden.
Das Hauptziel digitaler Signaturen ist die Schaffung eines sicheren Systems, welches eine Nachricht so markiert, dass wir ihre Integrität sowie den Absender feststellen und verifizieren können. Hierzu muss diese Markierung ein Geheimnis enthalten, welches nur der Absender wissen kann. Sollte die Nachricht während der Übermittlung durch einen Fehler oder einen Angreifer verändert worden sein, so muss diese Verifizierung fehlschlagen. Solange dies gegeben ist, können wir unsere momentanen Verfahren weiter für effiziente Signaturen benutzen. Es gibt aber mehrere Kriterien aus welchen folgt, dass zu empfehlen ist, schon jetzt nach Alternativen zu suchen, um einer möglichen Krise zuvorzukommen [1, S.11].

**Effizienz** [1, S.11]:
Die Effizienz beschreibt, wie viel Rechenleistung und Speicherplatz wir zur Erstellung einer Signatur sowie zur Verifikation dieser benötigen. Auch die Länge unserer an die Nachricht angehängten Signatur ist Teil dieses Begriffes, da dies Zusatzaufwand für den Nutzer darstellt. Ineffiziente Kryptographie ist eine Option für Endnutzer mit leistungsstarken, zu großen Teilen unbeschäftigten, Computern oder für eine verstärkte Sicherung von extra sensiblen Daten. Für industrielle Nutzung, wie bei schwer beschäftigten Webservern, ist dies jedoch keine Option. Die Fortschritte im Brechen unserer besten Algorithmen könnte dazu führen, dass diese den Zusatzaufwand nicht mehr schultern können und der Betreiber zwischen zusätzlicher Hardware oder einer Abschaltung des Sicherheitsfeatures wählen muss. Dies ist ein großes Problem, auf welches wir uns jedoch vorbereiten können, indem wir jetzt schon beginnen Forschungsaufwand in unsere alternativen Verfahren zu investieren. Aus vergangener Erfahrung konnte dies schon oft deutliche Verbesserungen erzielen.

**Verlässlichkeit** [1, S.12]:
In kritischen Bereichen wird ein Verfahren nur benutzt, wenn sich über eine längere Zeit die Ansicht etabliert hat, dass kein Angriffsvektor übersehen wurde. Aus diesem Grund investieren wir extensive Mengen an Forschungsaufwand in die Kryptoanalyse, um etablierte sowie neu vorgeschlagene Systeme auf eben jene Angriffsvektoren und Verwundbarkeiten zu untersuchen. Manchmal finden wir katastrophale Fehler, die den kompletten Ansatz unbrauchbar machen, manchmal kleinere Fehler, welche mit ein wenig Effizienzverlust ausgleichbar sind. Zusätzlich gibt es die Fälle, bei denen trotz intensiver Forschung, keine Angriffe auf einen Algorithmus gefunden werden können. Bei diesen stellt sich langsam die Hoffnung ein, dass der Algorithmus die erträumte komplette Sicherheit erreicht hat oder jedenfalls, dass potenzielle Angreifer auch keinen Erfolg hatten. Für auf neuen Problemen basierenden Systemen gibt es jedoch keine bestehende Forschung, wie diese verwundbar sein könnten. Dies gilt auch für die andere Richtung. Ohne das Wissen über Methoden, die für die Brechung des Problems genutzt werden können, ist dieses schwer zu schützen. All dies benötigt Zeit, in welcher die gesamte kryptographische Gemeinschaft sich zusammen nach vorne bewegen muss.

**Nutzbarkeit** [1, S.12]:
Zu einem Verfahren gehört heutzutage weitaus mehr als der Algorithmus selbst. Die gesamte Infrastruktur, mit welcher ein Nutzer sich über den Algorithmus informieren und sich diesen für seine Anwendung besorgen kann, ist ein essenzieller Bestandteil geworden, ohne welche die moderne Kryptographie nicht funktionieren würde. Manche modernen Algorithmen haben klein angefangen, RSA zum Beispiel bestand anfangs nur aus „cube modulo $n$ “, haben aber über ihre Lebenszeit an Komplexität gewonnen, um verschiedensten Angriffen widerstehen zu können und trotzdem noch eine akzeptable Effizienz zu bieten. Beispiele sind das Nutzen von Padding um Timing-Attacken zu verhindern oder die asymmetrische Verschlüsselung eines kleinen Schlüssels, welcher dann für ein effizienteres symmetrisches Verfahren bei langen Nachrichten genutzt wird.

Zusätzlich müssen potentielle Nutzer genügend Informationen über das Verfahren finden können und eine Anlaufstelle für Hilfe bei Problemen haben, ansonsten wird die Methode nicht genutzt. Bei weitverbreiteten Verfahren ist es einfach sich das gewünschte Verfahren zu besorgen, zum Beispiel über einen Paket-Manager. Es gibt viele Methoden sich über diese Verfahren zu informieren, sei es über eine simple Suche im Internet, welche einfache Erklärungen liefert oder tiefgehende Kurse an Universitäten. All diese über Jahre entstandenen Entwicklungen sind einfach erhältlich und verständlich, dies muss auch für unsere neuen alternativen Verfahren gelten, damit sie real genutzt werden.

Sollten Quanten-Computer erfolgreich vergrößert werden, sodass Shor's und Grover's Algorithmen nutzbar werden, ist es bereits zu spät diese Punkte anzustreben. Um nutzbare Alternativen zu besitzen, müssen wir jetzt schon Forschungsaufwand investieren. Solche Alternativen, auf welche wir unsere Forschung konzentrieren können, existieren auch schon. Ein Beispiel ist die Familie der Hash-basierten Signaturverfahren, welche wir im Verlauf dieser Arbeit betrachten. Eine Übersicht über drei weitere Alternativen für Signatur- oder Verschlüsselungsverfahren können Sie bei Interesse in [1] finden. Meines Erachtens bieten die drei oben genannten Kriterien eine gute Übersicht über Verfahren und zeigen welche zusätzlichen Forschungsaufwand verdient haben. Wenn ein Verfahren erstaunlich effizient ist, sollten wir prüfen, ob es wirklich sicher ist. Wenn ein Verfahren trotz intensiver Prüfung keine Schwachstellen zeigt, sollten wir Aufwand investieren, um es zu optimieren. Wenn ein Verfahren weit verbreitet ist, sollten wir dieses weiterentwickeln.

**Hash-basierte Signaturverfahren** [1, S.6-7, 35-93]:
Wir beschäftigen uns in dieser Arbeit mit dieser Familie von Verfahren, da sie mehrere für uns interessante Eigenschaften bietet, welche sie zu einem der primären Kandidaten der Post-Quanten Kryptographie macht. Insbesondere sind über die Jahrzehnte seit ihrer Einführung wenige mögliche Angriffe gefunden wurden. Zudem gab es keinen Angriff, welcher nicht durch eine leichte Vergrößerung der Sicherheitsparameter umgehbar war. Wir haben speziell ein Public-Key Verfahren gewählt, da die symmetrischen Schlüssel der Secret-Key Verfahren nur sehr begrenzt durch Quantencomputer betroffen sind. Dies sollte durch eine kleine Verlängerung der Schlüssellänge umgehbar sein. Fälle, in denen ein Secret-Key Verfahren jedoch nicht nutzbar ist, wie bei einer riesigen Basis an Nutzern, die miteinander kommunizieren wollen, benötigen jedoch eine dringende Alternative. Sie sind sogar bereits so weit entwickelt, dass sie in diesem Moment aktiv benutzt werden könnten. Dies geschieht nur nicht, da sich ihre Effizienz noch nicht auf dem Level befindet, welches aktuellen Verfahren vorweisen. Doch genau dieser Punkt könnte bald nicht mehr gelten, aufgrund der oben geschilderten stetigen Lösung der besten Probleme sowie der möglichen Trivialisierung dieser mittels Quantencomputern. Während RSA, DSA und ECDSA gebrochen werden, ist dieses Verfahren größtenteils immun gegen Shor's sehr schnellen Algorithmus [15], da es auf der Schwierigkeit des Invertierens von schwer zu invertierenden Funktionen basiert. Grover's Algorithmus [7] ist der beste bekannte Algorithmus, um solche zu invertieren. Er ist jedoch weitaus langsamer, wodurch wir mit diesem Verfahren eine weitaus bessere Basis für die Post-Quanten

Zeit besitzen. Ein weiterer angenehmer Vorteil ist, dass die grundlegende Hashfunktion, auf deren Kollisionsresistenz und Unumkehrbarkeit die Sicherheit dieses Verfahrens basiert, je nach gegebener Hardware frei gewählt werden kann. Sie muss nur eine ausreichende Stärke dieser beiden Konzepte vorweisen. Daraus folgt auch, dass bei Brechung einer bestimmten Hashfunktion das Verfahren auf eine andere umgelegt werden kann. Dies funktioniert solange nicht die Gesamtheit aller Hashfunktionen gelöst wird. Die grundlegenden Einmalsignaturverfahren bieten uns einen geringen Rechenaufwand zur Erstellung und Verifikation von Signaturen, benötigen aber ohne Modifikationen auch recht große Schlüssellängen. Das größte bestehende Problem dieser Verfahren ist jedoch, dass die simplen Signaturen nur einmal verwendet werden können. Dies folgt aus dem nötigen Herausgeben der Signatur, welche aus einem Teil des privaten Schlüssels besteht. Für eine praktische Nutzung wollen wir jedoch viele Signaturen auf einen einzelnen öffentlichen Schlüssel beziehen können, was eine Konversion in eine Mehrfachsignatur voraussetzt. Dies bewirkt, besonders auf Seiten des Erstellers, einen deutlichen Zusatzaufwand. Dieser wurde mittlerweile aber schon, im Vergleich zum originalen Verfahren, deutlich verbessert. Einige Autoren behaupten außerdem, dass ihre neue Version des Verfahrens schon jetzt, bevor Shor's Algorithmus nutzbar ist, ein seriöser Konkurrent zu aktuellen Verfahren ist [2, S.43].

# 2 Grundlagen

In diesem Kapitel stellen wir einige Definitionen vor, welche im weiteren Verlauf der Arbeit benutzt werden oder von wichtiger Bedeutung für das Thema dieser Arbeit sind.

**Definition 1** (Hashfunktion [11] Def. 1.54 S.33)**.** Gegeben sind die Mengen $M \subseteq \{0,1\}^*$ und $N \subseteq \{0,1\}^n$. Eine effizient berechenbare, totale Abbildung $h : M \to N$ wird als *Hashfunktion* bezeichnet. Sie komprimiert hiermit eine größere Menge binärer Zeichenfolgen auf eine kleinere.

Hashfunktionen sind, wie der Name vermuten lässt, ein grundlegender Bestandteil der Hash-basierten Signaturverfahren. Sie werden in diesen genutzt, um einen komprimierten Fingerabdruck des Eingabewertes zu erhalten. Da Hashfunktionen eine größere auf eine kleinere Menge abbilden, ist es unmöglich Kollisionen zu vermeiden. Hierzu müssten mindestens so viele Ausgaben wie Eingaben existieren, dies liefert aber nicht die von uns gewollte Komprimierung einer Nachricht. Aufgrund dessen benötigen wir kollisionsresistente Hashfunktion für die Sicherheit unserer Verfahren.

**Definition 2** (Kollisionsresistenz [11] S.324)**.** Eine Hashfunktion wird als *stark kollisionresistent* bezeichnet, wenn es für einen Angreifer effektiv unmöglich ist, ein $x$ und ein $y$ zu finden, für welche $h(x) = h(y)$ gilt. Eine Hashfunktion wird als *schwach kollisionresistent* bezeichnet, wenn es für einen Angreifer effektiv unmöglich ist, zu einem gegebenen $x$ ein $y$ zu finden, für welches $h(x) = h(y)$ gilt.

Effektiv unmöglich bedeutet, dass der benötigte Rechenaufwand mit heutzutage verfügbarer Hardware zu lange dauert, um praktisch nutzbar zu sein oder die Chance für einen Erfolg so gering ist, dass diese vernachlässigt werden kann. Für die Sicherheit von Signaturen wird mindestens die schwache Kollisionsresistenz benötigt, damit zu einer gegebenen Signatur keine alternative Eingabenachricht gefunden werden kann. Hierzu benötigen wir eine Hashfunktion mit einer ausreichend großen Ausgabemenge, welche beliebige Eingaben möglichst gleichmäßig auf die komplette Ausgabemenge verteilt. Es sollte jedoch nicht möglich sein, durch Kenntnis der Funktion eine Methode zu finden, welche deutlich effektiver als der Brute-Force Ansatz ist, wie bei Modulo.

**Definition 3** (Unumkehrbarkeit [11] Def. 9.9 S.327)**.** Eine Funktion $f$ wird als *unumkehrbar* oder auch als Falltür- oder Einwegfunktion bezeichnet, wenn die Funktion für alle nutzbaren $x$ einfach zu berechnen ist, aber es effektiv unmöglich ist, zu einem gegebenen $y$ ein beliebiges passendes $x$ zu finden, für welches $f(x) = y$ gilt.

Dies ist durch viele verschiedene Methoden erreichbar, zum Beispiel durch Entfernung eines Teils der originalen Informationen aus der Eingabemenge.

**Definition 4** (Kryptographische Hashfunktion [11] S.323)**.** Eine Hashfunktion, welche Kollisionsresistenz und Unumkehrbarkeit vorweist, wird als kryptographische Hashfunktion bezeichnet.

Ohne die Existenz kryptographischer Hashfunktionen wäre es nicht möglich, sichere digitale Signaturen zu erstellen, da es einfach wäre zu einer Signatur eine zweite passende Nachricht zu finden[1, S.35]. Schauen wir uns nun an einem kurzen Beispiel an, ob diese beiden Merkmale an einer Beispielfunktion zutreffen.

**Beispiel 5.** Eine simple aber häufig verwendete Hashfunktion wäre die Divisionsrestmethode $h(k) = k \mod m$. Welche Modulo nutzt, um die Menge der natürlichen Zahlen auf eine Teilmenge von 0 bis $m - 1$ einzugrenzen. Mit $k = 20$ und $m = 8$ wäre dies

$$h(20) = 20 \mod 8 = 4$$

Betrachten wir diese nun vor dem Hintergrund der vorher definierten Merkmale einer für uns sinnvollen Hashfunktion. Hier fällt uns zuerst auf, dass die Unumkehrbarkeit gegeben ist. Es ist uns nicht möglich aus der 4 zu schließen, dass die Eingabe eine 20 war. Durch das Nutzen von Modulo könnte es ein beliebiges Vielfaches von acht mit vier addiert sein. Dies zeigt jedoch auch sofort, dass diese Hashfunktion nicht kollisionsresistent ist, da jede Addition von acht genau dieselbe Ausgabe erzielt.

Als letzten Punkt definieren wir hier noch das Sicherheitslevel, welches einen Vergleichswert für kryptographische Algorithmen darstellt.

**Definition 6** (Sicherheitslevel)**.** Das Sicherheitslevel bestimmt wie viel Rechenaufwand ein Angreifer investieren muss, um einen spezifischen Algorithmus zu brechen und somit dessen Schutz zu umgehen. Die hier genutzte Sicherheitsdarstellung von $b$-Bit besagt, dass ein Angreifer $2^b$ Operationen mithilfe der besten bekannten Angriffsmethode benötigt, um diesen zu brechen.

Eine Übersicht über verschiedene Sicherheitslevel und deren Berechnungszeit ist in Abbildung 2.1 zu sehen. Dieses Sicherheitslevel kann genutzt werden, um die Effizienz

| $b$ | $2^b$ | 5GHz Prozessor | 50GHz Prozessor |
|---|---|---|---|
| 16 | 65536 | <1 Sekunde | <1 Sekunde |
| 32 | $4.29 * 10^9$ | ≈1 Sekunde | <1 Sekunde |
| 50 | $1.12 * 10^{15}$ | ≈2.6 Tage | ≈6.25 Stunden |
| 64 | $1.85 * 10^{19}$ | ≈116 Jahre | ≈12 Jahre |
| 100 | $7.92 * 10^{28}$ | $\approx 5 * 10^{11}$ Jahre | $\approx 5 * 10^{10}$ Jahre |
| 128 | $3.4 * 10^{38}$ | $\approx 2.16 * 10^{21}$ Jahre | $\approx 2.16 * 10^{20}$ Jahre |

Abbildung 2.1: Berechnungsdauer verschiedener Sicherheitslevel

verschiedener Algorithmen zu vergleichen. Dies geschieht, indem der Rechen- oder Speicheraufwand, welcher benötigt wird, um ein bestimmtes Sicherheitslevel zu erreichen, verglichen wird. Wir könnten zum Beispiel vorschlagen, dass bei heutiger Rechenleistung ein 128-Bit Sicherheitslevel ausreichend ist. Dies wären etwa $3.4 * 10^{38}$ Rechenoperationen, welche bei 5GHz Operationen pro Sekunde etwa $2.16 * 10^{21}$ Jahre dauern würde. Im Gegensatz dazu bietet ein 64-Bit Sicherheitslevel nur etwa 116 Jahre an Rechenoperationen auf einem einzelnen 5GHz Rechenkern. Wenn wir neue oder verbesserte Angriffsoptionen und die Leistung heutiger Supercomputer einbeziehen, können wir dies auf längere Zeit gesehen, nicht mehr als verlässlich bezeichnen. Nun können wir für jeden Algorithmus eine auf $n$ basierende Formel berechnen, welche angibt wie viel Rechen- und Speicheraufwand wir für ein Sicherheitslevel von n benötigen. Hiermit haben wir eine einigermaßen gute Übersicht, mit welcher wir Algorithmen, die auf komplett verschiedenen Basen beruhen, vergleichen können. Dies ist nicht zu verwechseln mit dem Sicherheitsparameter eines Algorithmus, dieser ist ein intern genutzter Wert, welcher die Länge genutzter Schlüssel und anderer Werte wie Ein-/Ausgabelängen von Funktionen definiert.

# 3 Einmalsignaturverfahren

## 3.1 Digitale Signaturen

Signaturen sind ein wichtiger Bestandteil unserer Gesellschaft, ohne welche wir weder dem Inhalt noch Absender einer Nachricht vertrauen könnten. Unsere gesamte digitale Gesellschaft könnte nicht existieren, oder wäre stark abgeändert, da es trivial wäre sich als jemand anders auszugeben.

**Definition 7** (Digitales Signaturverfahren [11] S.426)**.** Ein digitales Signaturverfahren besteht aus folgenden Bestandteilen:

- Eine aus der zugehörigen Nachricht generierte Signatur, welche die Nachricht der erstellenden Entität zuordnet.
- Einen Signaturalgorithmus S, welcher aus einer gegebenen Nachricht die zugehörige Signatur generiert.
- einen Verfikationsalgorithmus V, welcher die Authentizität der Signatur prüft.

Wenn V erfolgreich verifiziert, sollte dies sicherstellen, dass die Nachricht unverändert ist. Zusätzlich muss es garantieren, dass der Absender die Person ist, die das Geheimnis kennt, welches für die passende Signatur benötigt wird.

Es ist nicht effizient eine digitale Signatur zu benutzen, welche dieselbe Länge wie die Nachricht besitzt, Speicherplatz ist nicht unendlich vorhanden und eine Verdoppelung

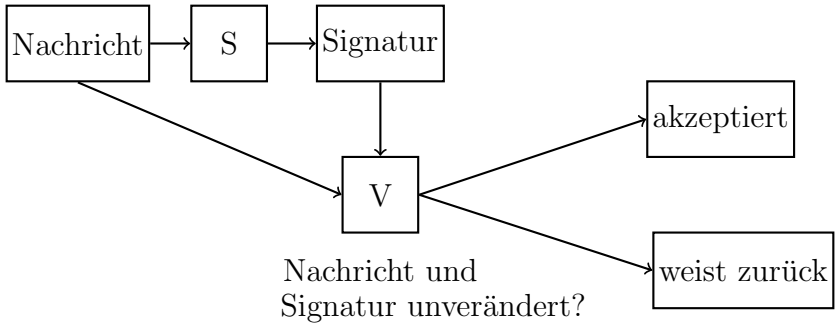


Abbildung 3.1: Ablauf eines digitalen Signaturverfahrens

der zu übertragenen Daten ist nicht wünschenswert. Aus diesem Grund werden Hashfunktionen benutzt, um beliebig lange Nachrichten auf eine Signatur mit vorbestimmter Länge zu reduzieren. Hierdurch entsteht jedoch das unvermeidbare Problem möglicher Kollisionen. Sollten diese effizient findbar sein, würden Angreifer die Möglichkeit haben einer Signatur eine komplett andere Nachricht zuzuweisen. Dies würde dazu führen, dass wir unser System nicht länger als sicher betrachten könnten. Deswegen werden kryptographische Hashfunktionen mit ausreichender Kollisionsresistenz zum Berechnen der Signatur einer Nachricht benutzt, damit ein Angreifer einen unzumutbaren Rechenaufwand zum Finden einer möglichen Alternative benötigt. Die Existenz kollisionsresistenter Hashfunktionen kann als Voraussetzung für die Existenz digitaler Signaturen gesehen werden. Ohne kollisionsresistente Hashfunktionen wäre es möglich, zu jeder beliebigen Signatur mit geringem Rechenaufwand eine alternative passende Nachricht zu generieren. Sollte dies möglich sein, wären weder Authentizität noch Integrität gegeben [1, S.35]. Zusätzlich könnte der originale Absender einer Nachricht erfolgreich verneinen diese gesendet zu haben, da nicht nachweisbar ist, dass die Nachricht nicht gefälscht wurde.

## 3.2 Lamport-Diffie Einmalsignatur

Die Lamport-Diffie Einmalsignatur wurde von L. Lamport in 1979 [10] als Weiterentwicklung einer Methode von M. Rabin [13] veröffentlicht. Sie ist aufgrund ihrer geringen Sicherheitsbedingungen bis heute immer noch größtenteils unversehrt, solange nicht mehrere Nachrichten unter demselben Schlüssel veröffentlicht werden. Die Sicherheit dieser, sowie der später vorgestellten Winternitz-Einmalsignatur, beruht rein auf der Kollisionsresistenz der genutzten Hashfunktion und der Unumkehrbarkeit der benutzten Einwegfunktion. Aus diesem Grund bietet sich dieses Verfahren als ein vielversprechender Kandidat für ein sicheres Post-Quanten System an [1, S.36].

**Definition 8** (Lamport-Diffie Signaturverfahren [10])**.**
Das Lamport-Diffie Verfahren besitzt folgende Komponenten:

- Die zu signierende Nachricht $M$ beliebiger Länge.
- Den zu wählenden Sicherheitsparameter $n$, welcher die Länge anderer Komponenten beeinflusst.
- Die Menge $K \in \{0, ..., 2^n - 1\} \in \mathbb{N}$ der möglichen Schlüssel.
- Eine kollisionsresistente kryptographische Hashfunktion $f\colon M \to K$, welche den Hashwert der Nachricht berechnet.
- Den aus der Nachricht berechneten Hashwert $h$ in Binärdarstellung.
- Eine Einwegfunktion $o\colon K \to K$, welche integraler Teil der Generierung und Verifikation der Signatur ist.
- Einen Signaturschlüssel $X$ aus $2n$ zufälligen Elementen von $K$, dargestellt als $n$ Indexpaare mit [0]/[1] als Zusatzindex.

- Ein Verifikationsschlüssel $Y$, auch aus $2n$ Elementen von $K$, dieser wird jedoch aus $X$ durch einmalige Anwendung von $o$ auf jeden Wert erstellt.
- Eine Signatur $\sigma$, welche aus $n$ Elementen des Signaturschlüssels $X$ besteht, diese werden mittels der binären Darstellung von $h$ gewählt. Entspricht $h_0 = 0$ nehmen wir $X_0[0]$ und bei $h_0 = 1$ nehmen wir $X_0[1]$, $h_1$ würde zwischen $X_1[0]$ und $X_1[1]$ entscheiden und so fort.

Nun werden wir zur Veranschaulichung die Funktion an einem Beispiel durchgehen. Dieses orientiert sich an [1, S.36-38], präsentiert aber die zweite Hälfte anhand von Feldern statt binären Matrizen, da ich diese Darstellungsweise als verständlicher erachte.

**Beispiel 9.** Damit die Mengen nicht zu unübersichtlich werden, wählen wir einen kleinen Sicherheitsparameter mit $n = 4$, zusätzlich wählen wir eine kollisionsresistente kryptographische Hashfunktion $f\colon M \to \{0, ..., 2^n - 1\}$ und die Einwegfunktion $o\colon x \to x + 1 \mod 16$.

**Signaturgenerierung:** Zuerst berechnen wir den Signaturschlüssel

$$X := (X_{n-1}[1], X_{n-1}[0], ..., ..., X_0[1], X_0[0])$$

Das Wählen zufälliger Zahlen ergibt

$$X = (2, 7, 6, 15, 7, 13, 1, 0)$$

Aus diesem können wir nun mittels $e : x \to x+1 \mod 16$ unseren Verifikationsschlüssel $Y$ berechnen.

$$Y := (o(X_{n-1}[1]), o(X_{n-1}[0]), ..., ..., o(X_0[1]), o(X_0[0]))$$

Mit unseren Werten entspricht dies

$$Y = (3, 8, 7, 0, 8, 14, 2, 1)$$

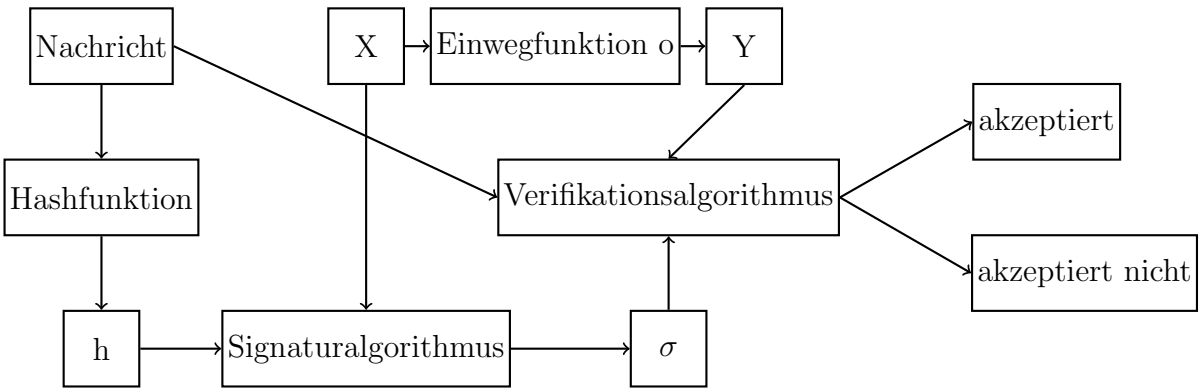


Abbildung 3.2: Ablauf des Lamport-Diffie Verfahrens

Als Nächstes berechnen wir den Hashwert unserer Nachricht in $\{0,1\}^n$ mithilfe von $f$, für dieses Beispiel wählen wir

$$h = 11 = (h_3, h_2, h_1, h_0) = (1, 0, 1, 1)$$

Mittels diesem können wir nun die Signatur berechnen

$$\sigma := (X_{n-1}[h_{n-1}], X_{n-2}[h_{n-2}], ..., X_0[h_0])$$

Bei unserem Beispiel ergäbe dies

$$\sigma = (X_3[1], X_2[0], X_1[1], X_0[1]) = (2, 15, 7, 1)$$

als Signatur und Ende der Signaturgenerierung.

**Signaturverifikation:** Wir benötigen die Signatur $\sigma$, den Verifikationsschlüssel $Y$, die Einwegfunktion $o\colon x \to x + 1 \mod 16$ und die Nachricht M.
Wir berechnen zuerst den Hashwert der Nachricht mittels f(M)=h=11=(1,0,1,1), daraufhin müssen wir prüfen ob die folgende Gleichung gilt

$$o(\sigma) = (Y_{n-1}[h_{n-1}], Y_{n-2}[h_{n-2}], ..., Y_0[h_0])$$

Dies wäre bei diesem Beispiel

$$(o(\sigma_3), o(\sigma_2), o(\sigma_1), o(\sigma_0)) = (Y_3[h_3], Y_2[h_2], Y_1[h_1], Y_0[h_0])$$

$$(o(2), o(15), o(7), o(1)) = (Y_3[1], Y_2[0], Y_1[1], Y_0[1])$$

$$(3, 0, 8, 2) = (3, 0, 8, 2)$$

Hiermit endet der Verifikationsalgorithmus mit einem Erfolg. Sollten beide Seiten der Gleichung nicht übereinstimmen, wird die Signatur zurückgewiesen und es besteht ein Problem mit der Signatur, dem Verifikationsschlüssel oder der Nachricht.

Nun zeigen wir noch anschaulich, warum die Signatur- und Verifikationsschlüssel dieser Methode möglichst nur einmal benutzt werden sollten.

**Beispiel 10** ([1]S.38)**.** Nehmen wir an, wir wollen 2 Nachrichten mit demselben Signaturschlüssel $X$ signieren, unser Sicherheitsparameter hier entspricht $n = 6$. Die Hashwerte unserer beiden Nachrichten sind $h_1 = (1, 1, 1, 1, 0, 0)$ und $h_2 = (0, 1, 0, 1, 1, 1)$. Dann veröffentlichen wir die folgenden 2 Signaturen

$$(X_5[1], X_4[1], X_3[1], X_2[1], X_1[0], X_0[0]), (X_5[1], X_4[0], X_3[1], X_2[1], X_1[1], X_0[1])$$

Jeder in einer der beiden Signaturen enthaltene Index des Signaturschlüssels wird veröffentlicht und ist somit dem Angreifer bekannt. Hiermit würde ein Angreifer Zugriff auf 10 der 12 Einträge unseres originalen Signaturschlüssels bekommen.

$$(X_5[1], X_5[0], X_4[1], X_3[1], X_3[0], X_2[1], X_1[1], X_1[0], X_0[1], X_0[0])$$

Die einzigen Teile des Signaturschlüssels, welche dem Angreifer nicht bekannt sind, wären $X_4[0]$ und $X_2[0]$. Mit der Kenntnis von $o$ und $Y$, welche mit jeder Nachricht verfügbar sein müssen, kann der Angreifer alle Signaturen die nicht $h_4 = 0$ oder $h_2 = 0$ enthalten fälschen. Eine mögliche Fälschung wäre $h = (1, 1, 1, 1, 1, 1)$.

*Anmerkung* 10.1 ([1]S.38). Solange die Hashfunktion zum Berechnen des Hashwertes kryptographisch sicher ist, sollte ein Angreifer nur valide Signaturen erzeugen können aber keine zu einer bestimmten Signatur passenden Nachrichten finden.

Zuletzt wollen wir uns noch kurz den nötigen Aufwand und Speicherverbrauch der Lamportsignatur anschauen. Für die Signaturgeneration benötigen wir das Berechnen eines $n$-Bit Hashwertes, das Füllen eines $2n$ langen Feldes mit $n$-Bit Zahlen, $2n$ Anwendungen der Einwegfunktion und das Heraussuchen von $n$ Zahlen aus einem Feld. Für die Signaturverifikation benötigen wir das Berechnen eines $n$-Bit Hashwertes, das Anwenden von $n$ Einwegfunktionen und das Lösen einer Gleichung. Die Länge der beiden Schlüssel beträgt $2n * n$. Die Länge der Signatur besteht aus dem $n * n$ langen halben Signaturschlüssel, der genutzten Hashfunktion und der genutzten Einwegfunktion.

## 3.3 Chaining

Das Lamport Verfahren besitzt zwei große Nachteile, die großen Schlüssellängen sowie die Unfähigkeit mehr als eine Nachricht mit einem Signaturschlüssel zu unterzeichnen, ohne dass dieser angreifbar wird. Der erste Nachteil führt zu langen Signaturen. Den zweiten Nachteil wollen wir umgehen, damit wir mehrere Nachrichten einem öffentlichen Schlüssel zuordnen können. Dies würde uns ermöglichen, jedem Teilnehmer einen einzigen oder wenigstens eine kleine Menge an öffentlichen Schlüsseln zuzuordnen. Ansonsten müssten wir eine Datenbank mit einem öffentlichem Schlüssel pro Person pro Nachricht erstellen. Es gibt zwei bekannte Methoden, um unser Einmalsignaturverfahren in ein Mehrfachsignaturverfahren umzuwandeln. Dazu gehören das hier beschriebene Chaining von Nachrichten sowie das im nächsten Kapitel beschriebene Merkle-Baum Verfahren, beide besitzen ihre eigenen Vor- und Nachteile.

Das Chaining von Nachrichten beinhaltet, dass der Signierer einen weiteren öffentlichen Schlüssel an das Ende jeder Nachricht setzt. Mit diesem wird dann die jeweils folgende Nachricht verifiziert. Aus dieser simplen Methode folgt jedoch, dass zum prüfen der Signatur der $n$-ten Nachricht alle vorigen Signaturen benötigt werden, womit die Länge der an die Nachricht angehängten Signatur mit jeder weiteren Iteration des Verfahrens linear ansteigt. Auch der Aufwand, den der Verifizierer leisten muss, steigt mit jeder weiteren Nachricht, da er jede einzelne Nachricht der Kette prüfen muss. Des Weiteren veröffentlicht die Länge der Signatur mit diesem Verfahren zusätzliche Informationen für den Angreifer, nämlich wie viele Signaturen zuvor erstellt worden. Die normalerweise genutzte Methode ist das Merkle-Baum Verfahren, welches in Kapitel 4 vorgestellt wird. Dieses setzt zwar voraus, dass wir vorher festlegen, wie viele Signaturen erstellt werden, bietet dafür aber ein logarithmisches Wachstum der Signaturlänge, welches der Effizienz unserer Grundverfahren bei großen Sicherheitsparametern deutlich weniger schadet [1, S.6-7] [5, S.96].

## 3.4 Winternitz Einmalsignatur

Die Technik des Winternitz Verfahrens ist eine Modifikation des Lamport Verfahrens, sie wurde R. Merkle 1979 von Winternitz vorgestellt und 1989 von R. Merkle veröffentlicht[12]. Sie versucht, aufgrund der guten Recheneffizienz des Lamport Verfahrens, die lange Schlüssellänge zu balancieren, indem sie zusätzlichen Rechenaufwand einsetzt, um die Länge der Signatur deutlich zu verkürzen. Die Grundidee ist mehrere Bits des Hashwertes der Nachricht für einen Wert der Signatur zu nutzen, indem diese zu einer natürlichen Zahl zusammengefügt werden und die Einwegfunktion entsprechend oft auf den Signaturwert angewendet wird. Dies verringert die Signatur grob um den Faktor $2w$ (siehe unten)[12, S.10].

**Definition 11** (Winternitz-Signaturverfahren )**.** [5, S.98] [1, S.38]
Das Winternitz Verfahren besitzt folgende Komponenten:

- Die zu signierende Nachricht $M$ beliebiger Länge.
- Den gewählten Sicherheitsparameter $n$, welcher die Länge anderer Komponenten beeinflusst.
- Den zu wählenden Winternitzparameter $w \geq 2$, welcher angibt wie viele Bits des Hashwertes pro Signaturwert signiert werden.
- Den aus $n$ und $w$ berechneten Hilfswert $t$ mit

$$t := t_1 + t_2 \text{ wobei } t_1 := \left\lceil \frac{n}{w} \right\rceil \text{ und } t_2 := \left\lceil \frac{\lfloor log_2 t_1 \rfloor + 1 + w}{w} \right\rceil$$

- Die Menge $K \in \{0, ..., 2^n - 1\} \in \mathbb{N}$ der möglichen Schlüssel.
- Eine kollisionsresistente kryptographische Hashfunktion $f\colon M \to K$, welche den Hashwert der Nachricht berechnet.
- Den aus der Nachricht berechneten Hashwert $h$ in Binärdarstellung.
- Eine Einwegfunktion $o\colon K \to K$, welche Teil integraler der Generierung und Verifikation der Signatur ist.
- Einen Signaturschlüssel $X$ aus $t$ zufälligen Elementen von $K$.
- Ein Verifikationsschlüssel $Y$ auch aus $t$ Elementen von $K$, dieser wird aus $X$ durch $2^w - 1$ Anwendungen von $o$ auf jeden Wert erstellt.
- Ein Hilfsfeld $b$ mit $t$ Elementen aus $\{2^{w-1}, ..., 0\}$, welches mithilfe von $h$ berechnet wird.
- Eine Signatur $\sigma$ bestehend aus den Werten des Signaturschlüssels $X$, auf welche die Einwegfunktion $o$ sooft angewendet wurde, wie der zugehörige $b$ Index vorgibt.

*Anmerkung* 11.1. Unauffällige Veränderung: Die Länge von $X$ und $Y$ wird nun durch t bestimmt, welches auf $n$ und $w$ basiert, statt durch $2n$ wie bei Lamport.

Diese Methode ist im Grundprinzip sehr ähnlich zu Lamports Algorithmus, birgt aber zusätzliche Komplexität durch das nötige Berechnen des Feldes b, wo bei Lamport nur der Hashwert h genutzt wurde. Auch diese Methode wird folgend an einem Beispiel durchgeführt.

**Beispiel 12** ([1] S.37)**.** Für dieses Beispiel wählen wir den Sicherheitsparameter $n = 4$ und den Winternitzparameter $w = 3$. Wir nutzen wieder die kollisionsresistente kryptographische Hashfunktion $f\colon M \to K$ und die Einwegfunktion $o\colon x \to x + 1 \mod 16$.

**Signaturgenerierung:** Wir beginnen damit, den Hilfswert $t$, $t_1$ und $t_2$ zu berechnen.

$$t_1 := \left\lceil \frac{n}{w} \right\rceil \qquad t_2 := \left\lceil \frac{\lfloor log_2 t_1 \rfloor + 1 + w}{w} \right\rceil \qquad t := t_1 + t_2$$

$$t_1 = \left\lceil \frac{4}{3} \right\rceil = 2 \qquad t_2 = \left\lceil \frac{\lfloor log_2(2) \rfloor + 1 + 3}{3} \right\rceil = 2 \qquad t = 2 + 2 = 4$$

Als Nächstes erstellen wir den Signaturschlüssel $X$ mit zufälligen Werten aus $K$.

$$X := (X_{t-1}, ..., X_0) \in K^t$$

$$X = (5, 14, 3, 9)$$

Mittels diesem können wir nun den Verifikationsschlüssel $Y$ durch $2^w - 1$ Anwendungen der Einwegfunktion o für jeden Wert errechnen.

$$Y := (o^{2^w-1}(X_{t_1}), ..., o^{2^w-1}(X_0))$$

Mit Anwendung der Einwegfunktion erhalten wir

$$Y = (o^7(5), o^7(14), o^7(3), o^7(9)) = (12, 5, 10, 0)$$

Nun müssen wir unser Hilfsfeld $b$ berechnen, um $\sigma$ bilden zu können, hierzu benötigen wir den Hashwert $h$ der Nachricht welchen wir in diesem Beispiel als $h = (1, 0, 1, 1)$ wählen. Wir stellen eine minimale Menge an Nullen vor diesen Wert, um eine durch $w$ teilbare Länge zu erhalten, das Ergebnis bezeichnen wir als $d$.

$$d = (0, 0, 1, 0, 1, 1)$$

Nun teilen wir diesen in $w$ lange Binärzahlen auf und überführen diese in die natürlichen Zahlen.

$$d = (001, 011)_2 = (1, 3)_{10}$$

Aus $d$ können wir nun den ersten Teil unseres Hilfsfeldes $b$ bestimmen.

$$d = (d_{t-t_1}, ..., d_0) = (b_{t-1}, ..., b_{t-t_1}) \Rightarrow b = (1, 3, ?, ?)$$

Um den zweiten Teil von $b$ zu erhalten, berechnen wir die Checksumme von $d$.

$$c := \sum_{i=t-t_1}^{t-1} (2^w - b_i) \Rightarrow c = (8 - 1) + (8 - 3) = 12$$

Diese Checksumme überführen wir in ihre Binärform und wiederholen, was wir vorher mit $h$ gemacht haben.

$$c = 12 = (1, 1, 0, 0) \Rightarrow (0, 0, 1, 1, 0, 0) \Rightarrow (001, 100)_2 = (1, 4)_{10}$$

Diese Werte fügen wir wie mit $d$ in die restlichen Plätze des Hilfsfeldes $b$ ein

$$c = (b_{t_{2-1}}, ..., t_0) \Rightarrow b = (?, ?, 1, 4)$$

Kombiniert mit den bekannten Werten folgt $b = (1, 3, 1, 4)$, die Signatur ergibt sich nun aus folgender Formel

$$\sigma := (o^{b_{t-1}}(x_{t-1}), ..., o^{b_0}(x_0))$$
$$\sigma = (o^1(5), o^3(14), o^1(3), o^4(9)) = (6, 1, 4, 13)$$

**Signaturverifikation:** Wir benötigen die Nachricht M, die Signatur $\sigma = (6, 1, 4, 13)$, den Verifikationsschlüssel $Y = (12, 5, 10, 0)$, die kryptographische Hashfunktion $f$ und die Einwegfunktion $o$: $x \to x + 1 \mod 16$. Wir beginnen damit, $b$ genauso wie oben dargestellt zu berechnen und erhalten $b = (1, 3, 1, 4)$, dies nutzen wir um folgendes zu prüfen.

$$(o^{2^w-1-b_{t-1}}(\sigma_{t-1}), ..., o^{2^w-1-b_0}(\sigma_0) = (Y_{t-1}, ..., Y_0)$$
$$(o^6(6), o^4(1), o^6(4), o^3(13) = (12, 5, 10, 0)$$
$$(12, 5, 10, 0) = (12, 5, 10, 0)$$

Wie wir sehen, gilt die Gleichung, wodurch der Verifikationsalgorithmus erfolgreich abschließt. Sollte die Nachricht, die Signatur oder der Verifikationsschlüssel fehlerhaft sein, gilt die Gleichung nicht und der Algorithmus bricht ab.

Nun zeigen wir, analog zu Lamport, warum der Signaturschlüssel dieses Verfahrens nicht für mehrere Nachrichten benutzt werden darf.

**Beispiel 13** ([1] S.40 [12] S.227)**.** Das Konzept verläuft äquivalent wie in Beispiel 10, bei jeder Herausgabe einer Nachricht veröffentlichen wir die Signatur derer. Diese besteht aus Teilen des Signaturschlüssels, auf welche die Einwegfunktion eine bestimmte Anzahl angewandt wurde. Nehmen wir an die beiden veröffentlichten Nachrichten nutzen $n = 4$ und $w = 3$ und die berechenbaren $b$ Werte ergeben (1,3,2,6) und (7,4,2,0). Jeder in einer der beiden Signaturen enthaltene modifizierte Index des Signaturschlüssels wird veröffentlicht und ist somit dem Angreifer bekannt. Nun hat ein Angreifer Zugriff auf jeden dieser Bausteine sowie alle Werte die mittels der Einwegfunktion aus diesen bis zu $o^7(X_i)$ berechnet werden können, dies wären

$o^1(X_3)$ bis $o^7(X_3)$ $\quad$ $o^3(X_2)$ bis $o^7(X_2)$ $\quad$ $o^2(X_1)$ bis $o^7(X_1)$ $\quad$ $o^0(X_0)$ bis $o^7(X_0))$

Der Angreifer könnte nun zum Beispiel eine Nachricht, aus welcher $b = (6, 7, 3, 0)$ berechnet wird, signieren. Eine Nachricht mit einem Wert von $b$, welcher geringer als in den gefundenen Signaturen ist, wie z. B. $o^2(X_2)$, wäre nicht möglich.

*Anmerkung* 13.1 ([1] S.40)*.* Wie bei Lamport in Anmerkung 10.1 gilt, solange die Hashfunktion zum Berechnen des Hashwertes kryptographisch sicher ist, kann ein Angreifer nur valide Signaturen erzeugen, aber keine zu einer bestimmten Signatur passende Nachricht finden.

Zuletzt wollen wir uns noch kurz anschauen, wie das Winternitz Verfahren den benötigten Aufwand und Speicherverbrauch des Lamport Verfahrens abgeändert hat. Winternitz ist durch das Nutzen von $w$ erstaunlich flexibel darin, wie viel Speicherplatz wir gegen Rechenleistung tauschen wollen, solange wir bei einigermaßen kleinen $w$ bleiben. Für die Signaturgeneration benötigen wir das Berechnen mehrerer Zwischenwerte, das Füllen eines $t$ langen Feldes mit $n$-Bit Zahlen sowie $t * 2^{w-1}$ Anwendungen der Einwegfunktion. Für die Signaturverifikation benötigen wir nahezu denselben Aufwand, bis auf das initiale Generieren des Signaturschlüssels, wobei die $t * 2^{w-1}$ Anwendungen der Einwegfunktion der Worst Case sind, wenn die Signatur exakt dem Signaturschlüssel entspricht. Aus $t * 2^{w-1}$ folgt, dass jede Vergrößerung unseres $w$ um eins die nötige Menge an Anwendungen der Einwegfunktion verdoppelt. Die auf $t$ basierende Länge der Schlüssel halbiert sich jedoch nicht, sondern senkt sich immer langsamer. Dies kommt zustande, da $t$ sich bei hohen Werten von $n$ an $t_1 \approx \frac{n}{w}$ annähert, da $t_2$ durch $log_2 t_1$vergleichsweise sehr klein wird. Des Weiteren folgt daraus der auf $n$ und $w$ basierende Speicherverbrauch des Verfahrens. Solange $n$ ausreichend groß gewählt wird, nähert sich die Anzahl unserer Schlüsselwerte an $t \approx t_1 \approx \frac{n}{w}$ an. Lamports Speicheraufwand der Schlüssel entsprach $2 * n * n$-Bit, die Schlüssel von Winternitz hingegen benötigen $t * n$-Bit, dies entspricht bei hohen $n$ in etwa $\frac{n}{w} * n$-Bit und bietet somit die versprochene Einsparung von $2w$. Der zusätzliche Rechenaufwand befindet sich primär in den zusätzlich nötigen Anwendungen der Einwegfunktion. Die Anzahl dieser basiert auf dem Genutzten $w$, bei einem $w$ von 4 bräuchten wir z. B. acht Anwendungen pro Wert unserer Schlüssel, diese sind aber auch in etwa $2w$ weniger, weswegen der Unterschied nicht gravierend ist.

Winternitz ist nicht sonderlich schwerer zu implementieren und bietet einen deutlich geringeren Speicherverbrauch für eine vertretbare Vergrößerung des Rechenaufwandes. Aufgrund dessen ist Winternitz der Algorithmus, welcher bei der Implementierung Hash-basierter Signaturverfahren normalerweise als Grundverfahren verwendet wird.

# 4 Merkle-Bäume

## 4.1 Merkle-Baum Signaturverfahren

Wie in Abschnitt 3.3, dargestellt besteht der größte Nachteil der Lamport und Winternitz Verfahren daraus, dass Schlüssel für nur je eine Nachricht genutzt werden sollten. Die dort vorgestellte Technik des Chaining löst das Problem und ermöglicht uns, eine unbegrenzte Menge an aufeinanderfolgenden Signaturen zu unterzeichnen. Sie birgt jedoch den Nachteil, dass jede weitere Signatur eine Steigerung des Speicherverbrauchs und der benötigten Rechenzeit vorweist. Ab einer bestimmten Länge der Kette wird diese Technik somit untragbar. In diesem Kapitel werden wir die alternative Methode des Merkle-Baum Verfahrens betrachten, welches von R. Merkle 1979 [12] veröffentlicht wurde. Dieses bietet im Vergleich einen weitaus kontrollierbareren, logarithmisch mit der Signaturmenge skalierenden, Mehraufwand durch das Nutzen eines Binärbaumes statt einer Kette. Wir müssen hier zwar eine feste Anzahl an zu erstellenden Signaturen wählen, dafür bleibt deren benötigte Rechenzeit und Größe einigermaßen konstant [5]. Bei unseren heutigen Algorithmen ist es theoretisch nicht nötig, solch eine Zahl festzulegen, praktisch jedoch ist die Anzahl an zu erstellenden Signaturen aber durch Regulationen oder der Limitierung einzelner Geräte begrenzt [1]. Das grundlegende Verfahren war, als es veröffentlicht wurde, zu ineffizient im Vergleich zu RSA und hat deswegen keine weitverbreitete Nutzung gefunden. Es bietet jedoch ein erstaunlich modulares Prinzip, wie beispielsweise die Möglichkeit der Nutzung eines beliebigen Einmalsignaturverfahren. Dies hat dazu geführt, dass viele Verbesserungen seit der Veröffentlichung vorgestellt wurden, von denen wir uns einige im weiteren Verlauf dieses Kapitels anschauen werden. Kombiniert mit dem Problem der stetigen Lösung sowie den leistungsstarken Quantenalgorithmen für RSA und Co, stellt dieses Verfahren eine vielversprechende Methode für zukünftige digitale Signaturen da.

**Definition 14** (Merkle Signaturverfahren [1])**.**
Das Merkle-Verfahren nutzt einen vollständigen Binärbaum, dessen Tiefe am Anfang festgelegt werden muss. Jedes Blatt des Baumes entspricht einer möglichen Signatur. Zusätzlich wird ein Sicherheitsparameter, ein Einmalsignaturverfahren, eine kryptographische Hashfunktion und eine Einwegfunktion benötigt. Jedem der Blätter wird ein Signaturschlüssel und der, mittels dem gewählten Einmalsignaturverfahren berechnete, Verifikationsschlüssel zugewiesen. Aus dem Verifikationsschlüssel wird, mittels der Hashfunktion, der Wert des Blattes berechnet. Die Werte der weiteren Knoten werden berechnet, indem die Hashfunktion auf die Verkettung der Werte ihrer Kinder angewendet wird. Die Wurzel bildet den öffentlichen Schlüssel des Baumes, während der private Schlüssel durch die Folge der Signaturschlüssel aller Blätter gebildet wird. Um

eine Signatur zu erstellen, wird zuerst das gewählte Einmalsignaturverfahren mit einem noch nicht benutzten Signaturschlüssel normal durchgeführt. Daraufhin müssen der Pfad vom Blatt zur Wurzel berechnet und die angrenzenden Knotenwerte gespeichert werden. Die Verifikation der Signatur verläuft nach den Vorgaben des Einmalsignaturverfahrens, außer dass nach erfolgreicher Verifikation nun die mitgelieferten Knotenwerte genutzt werden, um den Pfad vom Verifikationsschlüssel zum öffentlichen Schlüssel des Baumes zu rekonstruieren. Verläuft dies erfolgreich, wird die Signatur akzeptiert.

Wie zuvor werden wir dieses Verfahren anschaulicher darstellen, jedoch nicht durchrechnen, da dies nahezu ausschließlich aus der Berechnung von Hashwerten besteht.

Zu beginn legen wir ein $H \in \mathbb{N} \geq 2$ von 3 fest, welches uns $2^H = 8$ Signaturen ermöglicht. Zusätzlich wählen wir eine kryptographische Hashfunktion $f : \{0,1\}^* \rightarrow \{0,1\}^n$, eine für die Einmalsignatur benötigte Einwegfunktion e und einen Sicherheitsparameter n. Nun wählen wir zufällig gefüllte, für das gewählte Einmalsignaturverfahren passende, Signaturschlüssel $X_i$ für jedes Blatt und berechnen mittels dieser die zugehörigen Verfikationsschlüssel $Y_i$. Dies ergibt einen Binären Baum, wie in Darstellung 4.1.

Die Werte der Blätter bestehen aus der Hashfunktion f angewendet auf die Verifikationsschlüssel jedes Blattes, sie werden berechnet mit:

$$k_{0,j} := f(Y_j) \quad \text{z. B.} \quad k_{0,1} = f(Y_1)$$

Den Wert der inneren Knoten berechnen wir aus den Kindern der Knoten, welche mit folgender Formel zu finden sind, wobei $||$ die Verkettung der Knotenwerte darstellt:

$$k_{j,i} := f(k_{j-1,2*i}||k_{j-1,2*i+1}) \quad \text{z. B.} \quad k_{2,1} = f(k_{1,2}||k_{1,3})$$

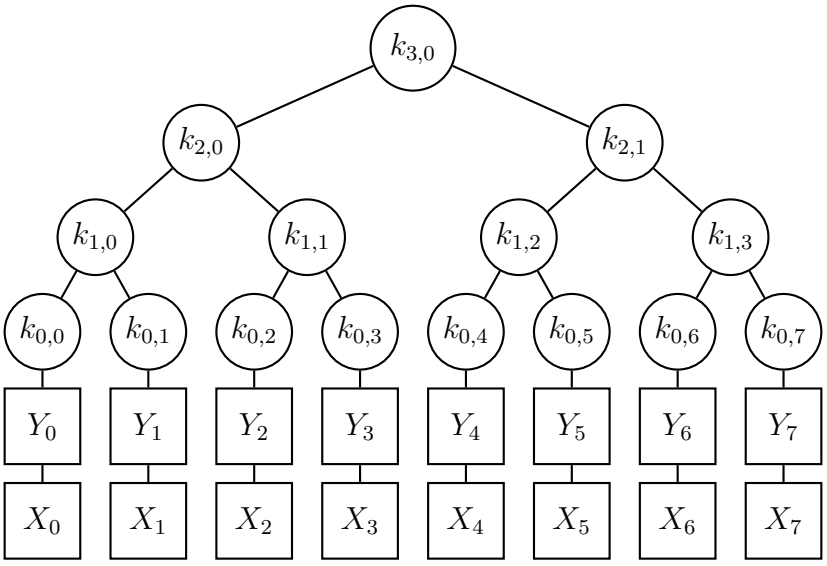


Abbildung 4.1: Merkle Baum mit Höhe H=3 [1, S.41]

Unser öffentlicher Schlüssel des Baumes wäre die Wurzel beziehungsweise $k_{H,0}$, welche rekursiv via oben vorgestellter Formel berechnet werden kann.
Der private Schlüssel wäre $(X_0||...||X_7)$, was eine einfache Konkatenation der Signaturschlüssel jedes Blattes ist.

Um nun eine Nachricht zu signieren, nehmen wir den Signatur- und Verifikationsschlüssel des unbenutzten Blattes mit dem niedrigsten Index und berechnen die Signatur normal nach unserem gewählten Einmalsignaturverfahren. Zusätzlich müssen wir aber noch den Verifikationspfad unseres Blattes zur Wurzel berechnen und die für den Pfad benötigten Knoten sowie den Index unseres Blattes mit unserer Signatur zusammen übermitteln.

$$\sigma := (s, \sigma_{ots}, Y_s, (k_0, ..., k_{H-1})$$

Hierbei ist s der Index des Blattes, $\sigma_{ots}$ ist die Signatur des Einmalverfahrens, $Y_s$ ist der zugehörige Verifikationsschlüssl und $(k_H, ..., k_1)$ ist der Authentikationspfad. Um eine erhaltene Signatur zu verifizieren, prüfen wir zuerst mittels des Einmalsignaturverfahrens ob die Signatur stimmt. Sollte dies zutreffen, nutzen wir die mitgelieferten Knotenwerte, um den Weg vom Verifikationsschlüssel zum öffentlichen Schlüssel des Baumes zu berechnen. Dies ist dargestellt in Abbildung 4.2, die grau gefüllten Knoten bilden den Verifikationspfad und die gestrichelten Knoten entsprechen den benötigten angrenzenden Knoten.

Für einen Merkle-Baum dieser Größe ist die Menge aller Knoten sehr übersichtlich und es wird wenig Platz benötigt um all diese rekursiv zu berechnen und abzuspeichern. Bei größeren Varianten der Struktur wird die Menge an $2^H - 1$ Knoten jedoch nicht mehr der trivialen Größe einiger Bytes entsprechen, wodurch wir hier den Treehash-Algorithmus zur Generation des Baumes vorstellen.

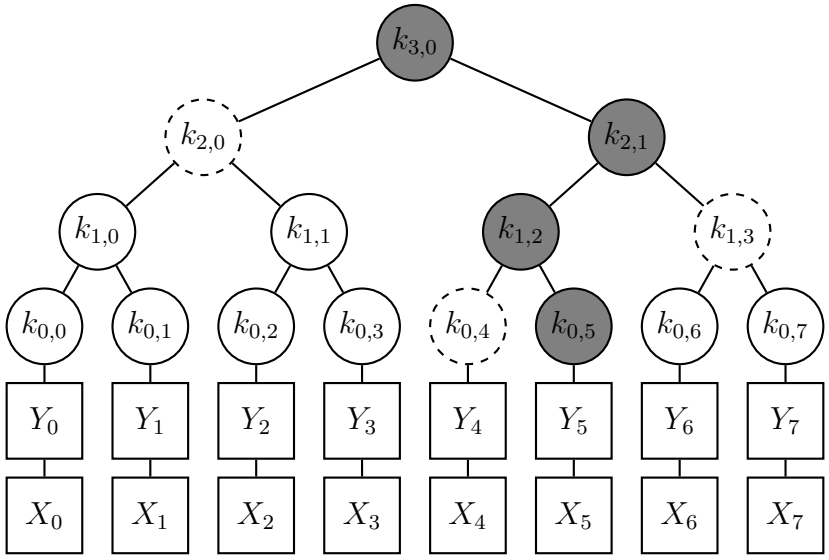


Abbildung 4.2: Verifikationspfad mit angrenzenden Knoten, Grau: Verifikationspfad, Gestrichelt: Authentifikations Knoten [1, S.41]

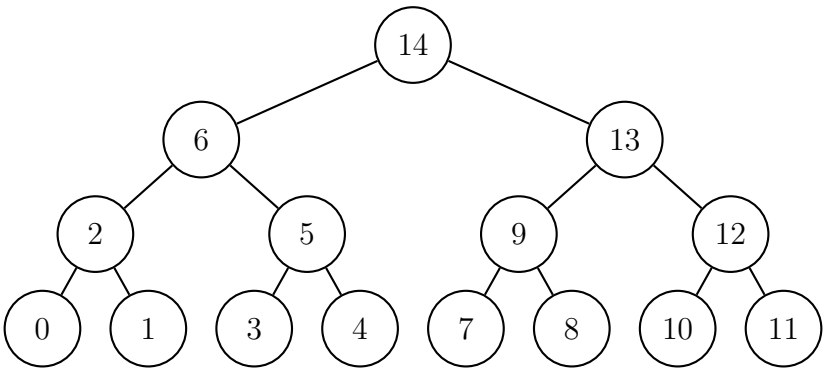


Abbildung 4.3: Ablaufsreihenfolge des Treehash Algorithmus [1, S.42]

**Algorithmus 15** (Treehash-Algorithmus [1])**.**
Eingabe: Höhe $H \geq 2$, Menge der Signaturschlüssel $Y$, Hashfunktion $f$
Ausgabe: Wurzel des Baumes
Ablauf:

1. Für $j = (0, ..., 2^H - 1)$ tue:
   a) Berechne das $j$te Blatt durch Anwendung der Hashfunktion auf $Y_j$ als $\text{Knoten}_1$
   b) Während $\text{Knoten}_1$ dieselbe Höhe wie der oberste Knoten des Stacks hat:
      i. Hole den obersten Knoten vom Stack als $\text{Knoten}_2$
      ii. Berechne den Elternknoten mittels $f(\text{Knoten}_1||\text{Knoten}_2)$
   c) Speichere den momentanen Knoten oben auf dem Stack ab
2. Gebe den letzten auf dem Stack vorhandenen Knoten zurück

Dieser Algorithmus verwendet einen Stack zum Abspeichern und ordnen der berechneten Knoten und verrechnet die Blätter des Baumes iterativ von links nach rechts. Um den Speicherverbrauch klein zu halten, hasht er jedes Mal, wenn es möglich ist, zwei Geschwisterknoten zum Wert ihres Elternknotens. Beim Beenden gibt er uns den gesuchten Wert der Wurzel und damit den öffentlichen Schlüssel des Baumes zurück. Abbildung 4.3 zeigt hierbei die durch den iterativen Verlauf hervortretende Berechnungsabfolge des Treehash Algorithmus.

## 4.2 Speicheroptimierung mithilfe eines Zufallsgenerators

Die erste Modifikation des Merkle-Verfahrens, welche wir hier vorstellen, ist die Nutzung eines deterministischen Zufallsgenerators für die Generierung der Signaturschlüssel. Diese wurde in [4] im Zusammenhang mit einer Überarbeitung des Merkle-Verfahrens vorgestellt und in [1] vom selben Autor erklärt. Das Merkle-Verfahren speichert bei einem Baum der Tiefe $H$ eine Menge von $2^H$ Signaturschlüsseln ab. Dies wären bei einem $H$ von 10 ganze 1024 Signaturschlüssel, die benötigt werden, um die Verifikationsschlüssel zum Berechnen des Baumes zu erstellen. Anstatt all diese Schlüssel fest abzuspeichern,

könnten wir ein wenig Rechenaufwand aufwenden, um dies zu umgehen. Dies ist dieselbe Grundidee, welche das Winternitz Verfahren dazu inspiriert hat, Lamport zu verbessern. Dies wäre möglich, indem wir einen deterministischen Zufallsgenerator nutzen, um aus einem $n$-Bit Seed eine passende Zufallszahl sowie einen Folgeseed für den nächsten Schlüssel zu generieren. Hierbei ist $n$ wie immer unser gewünschter Sicherheitsparameter.

$$z(\text{Seed}_i) \rightarrow (\text{Zufall}, \text{Seed}_{i+1})$$

Durch das Nutzen eines solchen Zufallsgenerators müssten wir nur den initialen Seed für unseren ersten Schlüssel speichern und dann alle Schlüssel zweimal generieren. Einmal bei der Erstellung unseres öffentlichen Schlüssels bzw. der Wurzel des Baumes sowie bei der Erstellung der Signaturen. Dieser Seed würde dann auch den privaten Schlüssel ersetzen, welcher vorher aus den Signaturschlüsseln aller Blätter bestand. Eine Runde des Zufallsgenerators wird jedoch nicht ausreichen, da die Signaturschlüssel mehrere Werte der Länge n benötigen, was zu einem recht langen Seed führen würde. Dies können wir aber lösen, indem wir die Ausgaben unserer ersten Runde des Zufallsgenerator in

$$z(\text{Seed}_{X_i}) \rightarrow (\text{Seed}_{x_0}, \text{Seed}_{X_{i+1}})$$

abändern und dann für jeden Signaturschlüssel einen genesteten Durchlauf unseres Generators durchführen. Für eine Übersicht siehe Abbildung 4.4.

$$z(\text{Seed}_{x_i}) \rightarrow (\text{Zufall}, \text{Seed}_{x_{i+1}})$$

Bei der normalen Durchführung des Merkle-Verfahrens müssen wir jeden Signaturschlüssel mit zufälligen Zahlen füllen, wofür vermutlich auch ein deterministischer Zufallsgenerator genutzt wird. Der Vorteil dieser Methode ist dadurch, dass wir einen zusätzlichen Durchlauf unseres Generators dagegen eintauschen, die Menge an $2^H$ recht

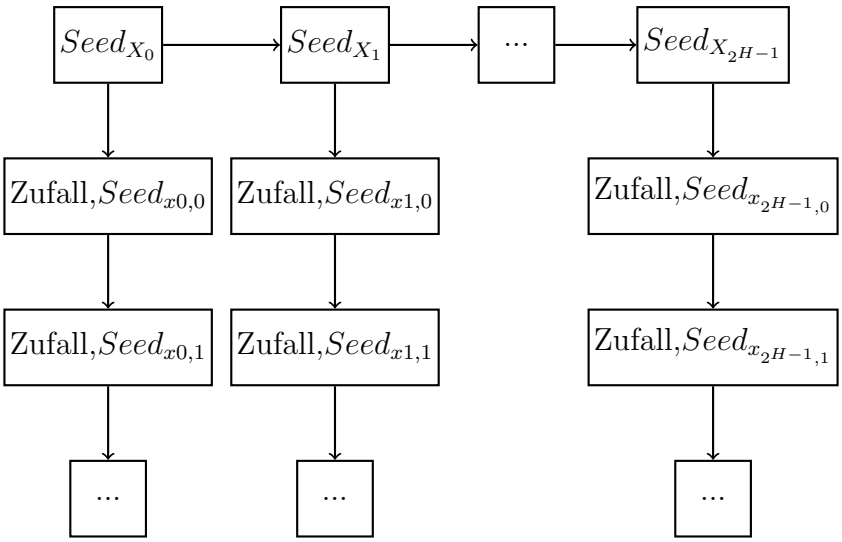


Abbildung 4.4: Veranschaulichung der Verzweigung des Zufallsgenerators [1, S.45]

langen Schlüsseln durch einen einzelnen $n$-Bit Seed im Speicher zu ersetzen. Dazu ist anzumerken, dass wir einen kryptographisch sicheren deterministischen Zufallsgenerator wählen müssen, da ein Signierer ansonsten zuvorkommende Signaturschlüssel berechnen könnte. Der Autor der Methode merkt außerdem an, dass das Nutzen eines solchen Zufallsgenerators dazu führt, dass das Merkle-Verfahren vorwärts sicher wird, solange der genutzte Zufallsgenerator vorwärts sicher ist. Dies bedeutet, dass bei der Widerrufung einer Signatur alle vorigen Signaturen gültig bleiben. Da selbst mit Kentnis des Generators nur auf den widerrufenen Seed folgende Seeds berechnet werden können.

## 4.3 Tree Chaining

In diesem Abschnitt stellen wir das Tree-Chaining-Verfahren vor, welches in [4] veröffentlicht wurde, es ist der primäre Bestandteil des dort vorgestellten Chained-Merkle-Signature-Verfahrens (CMSS). Die Motivation dieses Verfahrens ist, dass Merkle mit seinen $2^H$ möglichen Signaturen, ab einem $H$ von 20, sehr lange Schlüssel und eine sehr lange Generationszeit aufweist. Das dort vorgestellte Verfahren soll dagegen $2^{40}$ Signaturen unterzeichnen können, bevor es zu ineffizient wird. Dazu benutzt es mehrere Merkle-Bäume auf einmal. Es besteht aus einem Hauptbaum, welcher über seine Blätter mit mehreren Subbäumen verbunden wird. Diese Subbäume können auch weitere Subbäume an ihre Blätter anheften, die Tiefe der Baumstruktur wird mit $T \geq 2$ bestimmt. Wir nehmen in diesem Abschnitt an, dass die Tiefe $H$ jeder Ebene gleich ist aber theoretisch könnten wir dies variieren. Die Blätter des Hauptbaumes und der mittleren Subbäume leiten zu den eine Stufe tiefer liegenden Subbäumen, während die Blätter der untersten Stufe die für die Nutzer bestimmten Signaturschlüssel enthalten. Die Wurzel des Hauptbaumes ist der öffentliche Schlüssel des Verfahrens, während die Wurzeln der Subbäume als Signatur zur Verifikation ihrer Verbindung zum höher gelegenen Baum dienen. Ein Beispiel, wie dies aussehen könnte, wäre Abbildung 4.5, welche uns einen Hauptbaum und den zu $k_{0,0}$ und $k_{0,3}$ zugehörigen Subbaum zeigt. $\text{sig}_i$ und $Y_i$ sind hierbei die Signatur und Verifikationsschlüssel welche die Wurzel des Subbaumes mit dem Blatt des darüber liegenden Baumes verbinden.

Die Signatur dieses Verfahrens ist ein wenig aufwendiger zu erstellen, ihre Formel ist:

$$\sigma := (s, \text{Sig}_T, Y_T, \text{Auth}_T, ..., \text{Sig1}, Y_1, \text{Auth}_1)$$

Hierbei ist s der Index das Blattes welches wir nutzen, $\text{Sig}_T$ die Einmalsignatur des zu unterzeichnenden Dokumentes, $Y_T$ der zugehörige Verifikationsschlüssel und $\text{Auth}_T$ der Verifikationspfad zur Wurzel des untersten Subbaumes auf wessen Blatt unsere Signatur saß. Die folgenden Tupel sind dann die Einmalsignatur des momentanen Subbaumes, dessen Verifikationsschlüssel und der Verifikationspfad des Baumes auf Höhe $T - 1$ bis wir bei $T = 1$ ankommen, welches dem Hauptbaum entspricht. Zur Verifikation eines $\sigma$ verifizieren wir zuerst mittels unserem Einmalsignaturverfahren, ob $\text{sig}_T$ mittels dem zugehörigen $Y_T$ verifizierbar ist. Daraufhin berechnen wir den Weg zur Wurzel des Subbaumes mittels des gelieferten Authentikationspfades. Dies wiederholen wir rekursiv für

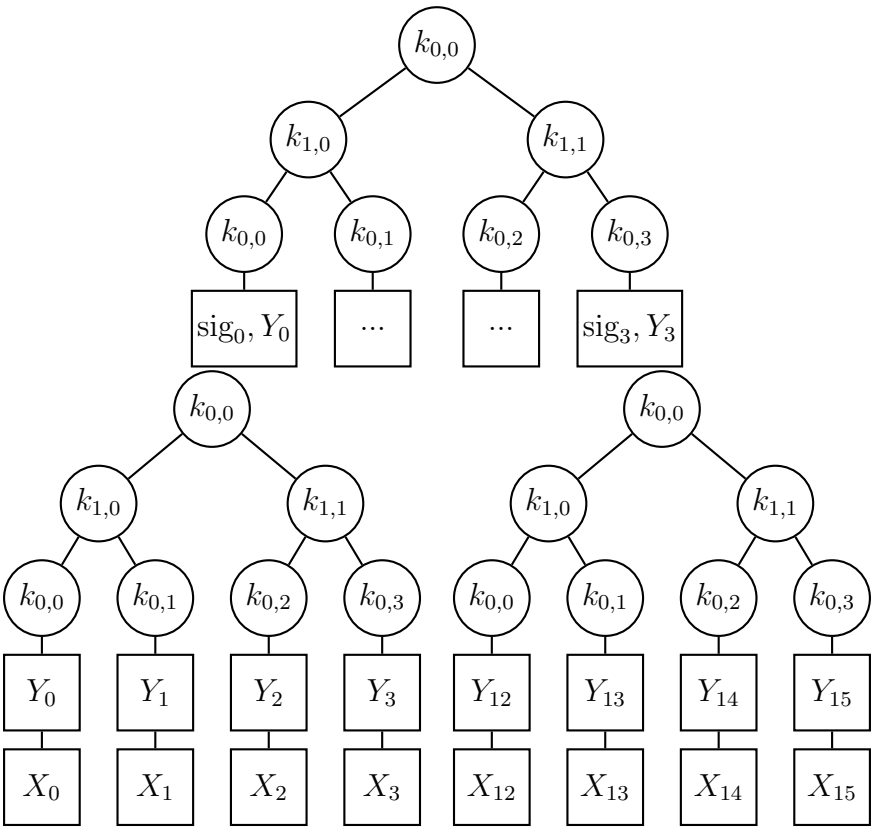


Abbildung 4.5: Ein CMSS Hauptbaum und 2 Subbäumen mit $H = 2$ und $T = 2$

jede Ebene der Struktur bis wir die Wurzel von Ebene 1 konstruiert haben, wir vergleichen dann den Wert der konstruierten Wurzel mit dem öffentlichen Schlüssel unserer Struktur und sollte diese Gleichung aufgehen, ist die Verifikation erfolgreich [1].

Diese Methode kann, statt den $2^H$ Signaturen des normalen Merkle Signaturverfahrens, die Summe der H werte aller Ebenen also $2^{H_1+\cdots+H_T}$ Signaturen signieren. Wenn alle Bäume dasselbe H benutzen, entspricht dies $2^{H*T}$ Signaturen. Die Menge an Signaturen ist equivalent dazu $2^{H_2+\cdots+H_T}$ einzelne Merkle Bäume mit demselben H Wert zu konstruieren. Wir nehmen den zusätzlichen Rechenaufwand der Bäume höherer Ebenen in Kauf, um all diese Signaturen auf einen einzelnen öffentlichen Schlüssel zusammenzuführen. Man könnte auch einen einzelnen Merkle-Baum mit $H_{merkle} = H_{Chain} * T_{Chain}$ generieren, dieser würde die zusätzliche Verifikation zwischen den Ebenen sparen aber dafür eine sehr lange Zeit zum Generieren brauchen. Der primäre Vorteil dieser Methode ist, dass wir die gesamte Struktur nicht vor dem Vergeben einer einzelnen Signatur komplett generieren müssen. Stattdessen können wir die Subbäume iterativ nach Nachfrage erstellen. Hierzu generieren wir zuerst nur den Hauptbaum sowie den ersten Subbaum jeder Ebene. Dies ist möglich, da die Bäume über die zugewiesene Einmalsignatur und den zugehörigen Verifikationsschlüssel verbunden werden, anstatt der Hashberechnung des normalen Merkle-Verfahrens. Hierbei werden die Verifikationsschlüssel aller Bäume beim Erstellen des Baumes generiert und die passende Einmalsignatur eines Kindes

wird berechnet, sobald dieses benötigt wird. Hierdurch benötigen wir nur $2^{H_1} + ... + 2^{H_T}$ Schlüsselpaare und $2^{H_1+1} + ... + 2^{H_t+1} - T$ Evaluationen einer Hashfunktion, um die ersten Signaturen nutzen zu können. Ein equivalenter Merkle-Baum würde die vollständigen $2^{T*H}$ Schlüsselpaare und $2^{T*H+1} - 1$ Hashevaluationen benötigen, bevor die erste Signatur genutzt werden kann [4]. Wenn nun eine Signatur angefertigt wird, berechnet der Signierer am Schluss noch einen Teil des nächsten Baumes auf der untersten Ebene. Hierzu generiert er den Teil des nächsten Teilbaumes der Ebene, welcher aus dem Blatt mit demselbem Index, welchen er für seine Signatur genutzt hat, gebildet werden kann. Sollte der Index dem letzten Blatt des Baumes entsprechen und somit den nächsten Teilbaum vollenden, nutzt er nun den passenden Signaturschlüssel des eine Ebene höher liegenden Baumes und berechnet die dem vollendeten Baum zugehörige Einmalsignatur. Nach dem Berechnen der Einmalsignatur wiederholt sich der Vorgang, indem er nun das nächste Blatt des eine Ebene höheren Baumes berechnet. Dies endet erst, wenn ein Blatt berechnet wird, welches nicht das letzte eines Baumes ist oder wenn der bereits vollständig generierte Hauptbaum erreicht wird.

## 4.4 Verteilte Signaturgeneration

Die im vorigen Abschnitt beschriebene Methode des Tree-Chainings erhöht zwar die effiziente Kapazität eines Baumes des Merkle-Verfahrens, aber sie birgt auch einige unerwünschte Nachteile. Der erste Nachteil ist die Vergrößerung der schon sehr großen Grundsignatur der Hash-basierten Verfahren durch die Hinzufügung einer zusätzlichen Einfmalsignatur für jede zusätzliche Baumwurzel, die durchlaufen wird. Der zweite ungewünschte Effekt ist, dass einige Signaturen deutlich mehr Aufwand kosten als andere, besonders das letzte Blatt jedes Baumes. Das Verfahren verteilt die komplette Berechnung der höherliegenden Bäume auf die Signierer des letzten Blattes, während der Rest jeweils nur ein Blatt berechnet und dieses in den Knotenkonstruktionsalgorithmus einfügt. Besonders schlimm trifft es einen Signierer, wenn er das letzte Blatt aus allen $T - 1$ Ebenen erwischt und somit ein Blatt in jeder Ebene sowie $T - 1$ Wurzelsignaturen generieren muss. Um dieses Problem anzugehen, wurde im Folgejahr nach der Veröffentlichung des Tree-Chainings die Verteilte Signaturgeneration entwickelt und in [2] vorgestellt. Das Ziel dieser Erweiterung war die Gleichverteilung der Arbeit aller Signierer und die Kürzung der Signaturlängen. Die Autoren waren auch mit den $2^{40}$ Signaturen von CMSS nicht zufrieden und haben die effizient nutzbare Menge mit diesem Verfahren auf $2^{80}$ bzw. etwa 1.2 Quadrillionen erhöht, was für nahezu jede Anwendung bei weitem ausreichend sein sollte.

**Signaturverkürzung**[2]: Zuerst schauen wir uns an, wie die Länge der Signaturen verkürzt wird. Hierzu beginnen wir mit der Beobachtung, dass sich die Authentikationspfade in den höheren Bäumen nicht sehr oft verändern. Ein kompletter Unterbaum durchläuft immer denselben Pfad aller seiner Väterbäume und nutzt dieselben Wurzel-Einmalsignaturen für deren Übergänge. Da wir das Winternitz Verfahren zur Berechnung unserer Einmalsignaturen nutzen, können wir mittels dem Winternitzparameter $w$ modifizieren, wie viel Rechenaufwand wir aufwenden, um die Länge unserer Signa-

turen zu kürzen. Normalerweise haben wir ein $w$ gewählt, welches ein gutes Verhältnis von Rechenaufwand zur Verkürzung einer Signatur vorwies. Wenn wir jedoch unser $w$ in höheren Bäumen vergrößern, bewirkt dies nicht nur die Verkürzung einer Signatur, sondern aller Signaturen, die aus diesen abstammen.

**Beispiel 16.** Angenommen 64 Signaturen durchlaufen die Wurzel-Einmalsignatur, deren $w$ wir ändern, die Länge dieser Einmalsignatur wählen wir hier als 10 Schlüssel. Wenn wir $w$ jetzt zum Beispiel von 4 auf 5 erhöhen, tauschen wir den doppelten Rechenaufwand gegen eine Verkürzung der Signatur um ein fünftel ein, sie entspricht dann also nur noch 8 Schlüsseln. Nun vergleichen wir die kombinierte Länge dieser Einmalsignatur in den 64 Signaturen, die sie durchlaufen. Vorher hatten wir $64 * 10 = 640$ Schlüssel, nun mit $w = 5$ haben wir nur noch $64 * 8 = 512$ Schlüssel, wodurch wir kombiniert 128 Schlüssel eingespart haben statt der 2 Schlüssel, welche unsere vorige $w$ Berechnung berücksichtigt hätte.

*Anmerkung* 16.1. Die Einmalsignatur eines Blattes des höheren Baumes entspricht hierbei der Signatur der Wurzel des spezifischen Kindes, zu welchem das Blatt verbindet. Dementsprechend verändern wir das $w$ nicht nur für den höheren Baum sondern auch für diese Verbindung.

Die Berechnung des optimalen $w$ sollte also berücksichtigen, wie viele Signaturen von der Einsparung profitieren.

**Aufwandsverteilung**[2]: Der zweite Teil dieser Methode beschäftigt sich nun damit, wie der Aufwand zur verteilten Berechnung der Struktur und der nun teureren Wurzel-Einmalsignaturen sinnvoll verteilt wird, sodass wir einen möglichst einheitlichen Signaturaufwand haben. Das Ziel ist die Berechnung der Blätter, Wurzel-Einmalsignaturen und Verifikationspfade möglichst gleichmäßig auf alle Blätter zu verteilen. Hierzu müssen wir zuerst bei der Initialisierung des Baumes nicht nur den ersten Baum jeder Ebene konstruieren und deren Wurzeln signieren, sondern auch den zweiten Baum jeder Ebene konstruieren. Die Berechnung dieser drei Eigenschaften, der Blätter, Wurzel-Einmalsignaturen und Verifikationspfade, wird nun auf drei separate Wege durchgeführt, welche nach der normalen Signaturberechnung eines Signierers ausgeführt werden.

**Wurzelsignatur**[2]: Das erste Blatt eines Baumes berechnet die Menge an Hash- und Einwegfunktionen, die nötig sind, um die Wurzel des nächsten Baumes derselben Ebene mit dem zugehörigen $w$ zu signieren und teilt diesen Aufwand durch die Anzahl der Blätter, die daran arbeiten werden. Jedes folgende Blatt berechnet nun so viele Hashfunktionsanwendungen oder Einmalsignaturanwendungen, wie das Ergebnis ihnen vorgibt und das letzte teilnehmende Blatt vollendet die Berechnung. Hierbei ist zu beachten, dass die Blätter des aktiven Baumes nicht nur die Wurzel des nächsten Baumes auf der untersten Ebene signieren, sondern gleichzeitig an der Signatur der Wurzel des nächsten Baumes eine Ebene höher arbeiten. Jedoch teilen sie sich diese mit allen anderen Kindern ihres Vaterbaumes. Genauso wird an dem Baum der höheren Ebene mit noch mehr Mitarbeitern gearbeitet. Dieser Vorgang setzt sich so fort. Letztlich ist noch zu beachten, dass das $w$, welches zur Berechnung einer Wurzelsignatur genutzt wird, dem $w$ der darüberliegenden Ebene entspricht, da jeder Verifikationspfad des Baumes

durch dieses Blatt führt und somit denselben Vorteil durch das erhöhte $w$ erfährt, wie der darüberliegende Baum. Das $w$ einer Ebene wird also zur Signierung der Blätter genutzt, für die Wurzel wird das $w$ der darüberliegenden Ebene gewählt.

**Pfadberechnung**[2]: Die Blätter eines Baumes berechnen den nötigen Authentikationspfad für die Kinder des nächsten Blattes ihres Vaterbaumes. Hierzu berechnet das erste Blatt wieder, wie viel Aufwand jedes andere Blatt leisten muss, woraufhin diese, wenn sie signiert werden, den ihnen zugewiesen Aufwand ausführen. Equivalent zur Wurzelsignatur teilen sie sich mit den anderen Kindern ihres Vaterbaumes die Berechnung des nächsten Blattes dessen Vaters und so weiter.

**Baumkonstruktion**[2]: Damit die Wurzelsignatur des nächsten Baumes durchgeführt werden kann, muss dieser vorher konstruiert werden. Hierfür müssen die Blätter, wenn sie signiert werden, den übernächsten Baum konstruieren. Für die Berechnung von Blättern, die Teil von Bäumen der untersten Ebene sind, gibt es keine verteilte Berechnung, da für jedes berechnete Blatt genau ein zu berechnendes Blatt mit demselben Index zwei Bäume weiter existiert. Die beste Verteilung bei der untersten Ebene entspricht also einfach einer eins zu eins Verteilung, welche nach Abschluss an den Baumkonstruktionsalgorithmus übergeben wird. Für die übernächsten Bäume auf den höher liegenden Ebenen berechnet jedes Blatt das Blatt mit demselben Index, welches sie in ihrem Vaterbaum durchlaufen. Das erste Blatt berechnet wie immer den nötigen Aufwand und die restlichen führen diesen durch. Alle Blätter, die aus $\text{Blatt}_j$ ihres Vaterbaumes folgen, berechnen also $\text{Blatt}_j$ des übernächsten Baumes der Ebene sowie die nun mittels Treehash neu bildbaren Knoten.

Letztlich haben wir also drei verschiedene zusätzliche Aufgaben, die ein Signierer nach dem Erstellen seiner eigenen Signatur teilweise ausführen muss. Dazu zählen die Berechnung des nächsten Verifikationspfades der Väterbäume, das Signieren der Wurzel des nächsten Baumes auf jeder Ebene sowie die Berechnung der Blätter des übernächsten Baumes auf jeder Ebene. Wie dies in einer kompletten Baumstruktur in etwa aussieht, ist in Darstellung 4.6 zu sehen. Hier markiert der graue Knoten den aktiven untersten Baum, bei den leichtgrauen werden Verfikationspfade berechnet, bei den gestrichelten

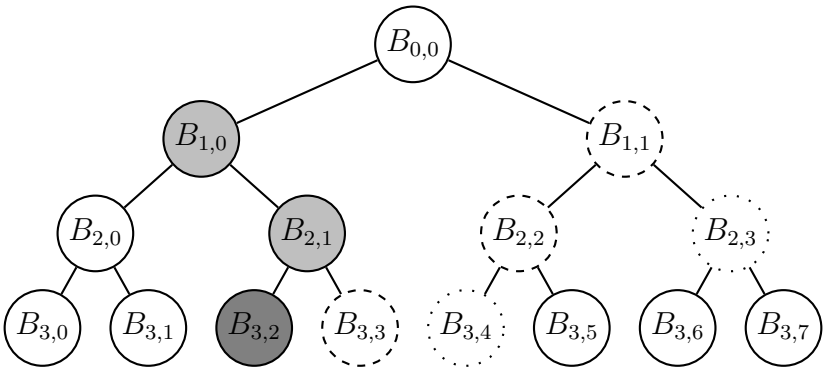


Abbildung 4.6: Verteilung der zusätzlichen Aufgaben, grau=aktiver Baum, leichtgrau=Pfadberechnung, gestrichelt=Wurzelsignaturberechnung, gepunktet=Baumkonstruktion

werden Wurzeln signiert und bei den gepunkteten werden Blätter und somit die Bäume konstruiert.

## 4.5 Algorithmen zur Verifikationspfadberechnung

Ein großer Teil des Rechen- und Speicheraufwandes der Signaturgeneration besteht aus der Berechnung des Verifikationspfades. Dieser ist einzigartig und fest für jede Signatur, wodurch folgt, dass dem Verifizierer egal ist, welchen Algorithmus wir hierfür nutzen, die übermittelten Knoten müssen nur korrekt sein. Es gibt jedoch trotzdem viele Möglichkeiten diese Berechnung effizienter zu gestalten, da sich zwischen einzelnen Blättern oft nur kleine Teile des Pfades verändern. Dies führt dazu, dass wir hier, wie so oft in diesem Feld, ein Optimierungsproblem bezüglich der Abwägung zwischen Speicher- und Rechenaufwand vorfinden. Die rechnerisch effizienteste Methode ist, den kompletten Baum in den Speicher zu laden und mit wenig Zusatzaufwand, mithilfe der Knotenindexe, den Pfad für jedes Blatt zu bilden. Der geringste Speicherplatz wird benötigt, wenn wir jeden Knoten jedes Pfades einzeln von den Blättern komplett neu berechnen. Natürlich sind diese beiden jeweils nur ideal, wenn die ausführende Hardware deutliche Begrenzungen in einem der beiden Bereiche vorweist.

### 4.5.1 Merkles Verifikationspfad Algorithmus

Wir werden uns nun den von Merkle [12] vorgestellten Verifikationspfad-Algorithmus anhand einer übersichtlicheren Darstellung in [16] anschauen. Ein allgemeiner Verifikationspfad-Algorithmus besteht aus drei Phasen:

1. Schlüsselgeneration: Berechnung der Wurzel des Baumes und Vorbereitung der ersten Runde des Algorithmus.
2. Ausgabe: Rundenbasierte Anwendung des Algorithmus, welcher jeweils den Authentikationspfad der momentanen Signatur zurückgibt und den Authentikationspfad der nächsten vorbereitet.
3. Verifikation: Prüfung, ob der zurückgegebene Pfad korrekt ist.

Die Idee der Verifikationspfad-Algorithmen ist, dass wir in einer Runde den Verifikationspfad der momentanen Signatur ausgeben und daraufhin, in Vorbereitung der nächsten Runde, die für die nächste Signatur nötigen Modifikationen vornehmen. Hierbei ist die Herausforderung, dass der Algorithmus eine ausreichende Menge an Operationen pro Runde bereitstellt, um die Berechnung aller nötigen Knoten des nächsten Pfades zu garantieren.

Merkles Algorithmus beginnt die Schlüsselgeneration mit der Konstruktion des kompletten Baumes durch den in Kapitel 4.2 vorgestellten Treehash Algorithmus. Währenddessen werden die Verifikationsknoten des ersten Pfades, für alle Ebenen $h < H$, als $\text{Auth}_h$ abgespeichert. Zusätzlich wird der Knoten, welcher auf der Ebene $h$ als Nächstes benötigt wird, auf dem der Ebene zugehörigen $\text{Stack}_h$ hinterlegt. Hierbei entspricht $\text{Auth}_h$ während der Schlüsselgeneration immer dem ersten rechten Knoten der Ebene

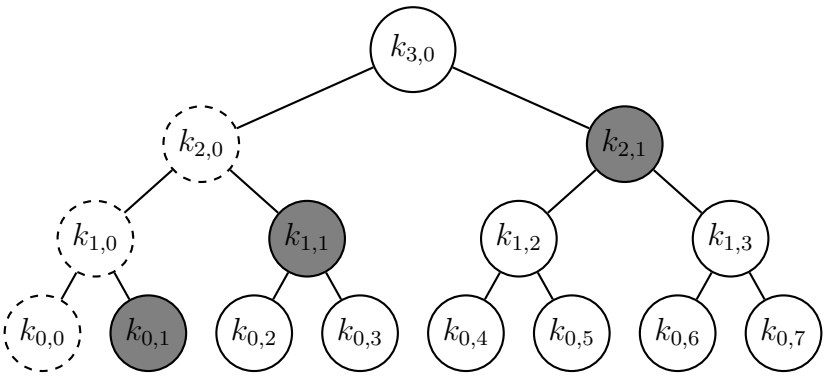


Abbildung 4.7: Grau: Initiales $\text{Auth}_h$, Gestrichelt: Initialer Knoten auf $\text{Stack}_h$

und $\text{Stack}_h$ enthält den ersten linken Knoten der Ebene, wie in Darstellung 4.7 ersichtlich.

Für die Aktualisierung dieser Werte während der Ausgaberunden nutzt Merkles Algorithmus eine Instanz des Treehash Algorithmus für jede Ebene $h < H$, welche jeweils den zugehörigen $\text{Stack}_h$ nutzt. Der Verifikationspfad bewegt sich auf der Ebene $h$ alle $2^h$ Runden, hieraus folgt, dass die der Ebene zugehörige Treehash Instanz den Wert des nächsten Knoten innerhalb von $2^h$ Runden bereitstellen muss. Das Berechnen eines Knotens benötigt $2^{h+1} - 1$ Berechnungen eines Blattes oder Anwendungen der Hashfunktion. Hierdurch folgt, dass wir jede Runde 2 dieser Operationen für jede Treehash Instanz veranschlagen müssen. Sobald der nächste Knoten einer Ebene bei der $2^h$-en Signatur benötigt wird, setzen wir den in $\text{Stack}_h$ enthaltenen Knoten als $\text{Auth}_h$ und initialisieren die $\text{Treehash}_h$ Instanz für den nächsten auf der Ebene benötigten Knoten.

Die Verifikationsphase verläuft bei diesem und den folgenden Algorithmen exakt wie die Verifikation einer Signatur durch Berechnung der Wurzel und folgenden Vergleich zum öffentlichen Schlüssel.

**Algorithmus 17** (Klassischer Merkle Verifikationspfad Algorithmus [16])**.**

1. Setze $s = 0$.
2. Ausgabe:
   - Gebe für jedes $h \in [0, H-1]$ $\text{Auth}_h$ zurück.
3. Erneuere $\text{Auth}_h$ Knoten:
   Für alle $h$ bei denen $s+1$ durch $2^h$ teilbar ist:
   - Setze den einzigen Knoten in $\text{Stack}_h$ als $\text{Auth}_h$.
   - Wähle das Startblatt der neuen $\text{Treehash}_h$ Instanz mit Index $(s+1+2^h) \oplus 2^h$
   - Initialisiere eine neue $\text{Treehash}_h$ Instanz für den nächsten benötigten Knoten der Ebene. $\text{Stack}_h$ initialisiert mit dem gewählten Startblatt.
4. Update Treehash Instanzen:
   Für alle $h \in [0, H-1]$:

- Führe zwei Operationen für Treehash$_h$ durch.

5. Setze $s = s + 1$.
6. Solange $s \leq 2^H$ gehe zu Schritt 2.

Merkles Verifikationspfad Algorithmus bildet einen Mittelweg zwischen den zuvor dargestellten Extremen für Rechen- und Speichereffizienz. Er behält den weiter nutzbaren Teil des vorigen Pfades bei und berechnet alle nächsten Knoten von Grund auf rechtzeitig, bevor diese benötigt werden. Schon bei der Veröffentlichung merkte der Autor jedoch einige Ansatzpunkte für Verbesserungen an [12, S.235], welche in den folgenden Algorithmen implementiert werden.

### 4.5.2 Szydlos Verbesserung der Treehash Stackgrößen

Merkles Verifikationspfad Algorithmus enthält ein Worst Case Szenario, bei diesem erreicht jede der $H$ Treehash Instanzen gleichzeitig die maximale Höhe ihres Stacks. Jede Treehash Instanz kann hierbei bis zu $h+1$ Knoten enthalten, einen für jede Ebene sowie das zusätzliche letzte Blatt, bevor es mit seinem Nachbarn verhasht wird. Dies wird dargestellt in Abbildung 4.8. Dies führt dazu, dass unser Algorithmus keinen Platzverbrauch unter $1+2+...+H$ garantieren kann, was einem quadratischen Speicheverbrauch von $\Omega(\frac{H^2}{2})$ entspräche[16]. Dies kann jedoch deutlich verbessert werden, indem die Treehash Operationen so verteilt werden, dass Instanzen mit einem großen Stack priorisiert werden, wie in dem in [16] vorgestellten Algorithmus von Szydlo. Das Ziel dieses Algorithmus ist, die Stacks der Treehash Instanzen möglichst leer zu halten, indem die vorher gleichverteilten Operationen dieser nun auf einzelne Instanzen fokussiert werden. Dazu erweitert der Algorithmus die Funktionen der Stacks um die Funktion Stack$_{h.\text{low}}$.

**Definition 18** (Funktion Stack$_{h.\text{low}}$[16])**.**

1. Stack$_{h.\text{low}}$ gibt die Höhe des tiefsten Knotens im Baum, welcher in Stack$_h$ gespeichert ist, zurück.
2. Ist der Stack leer, entspricht Stack$_{h.\text{low}} = h$.

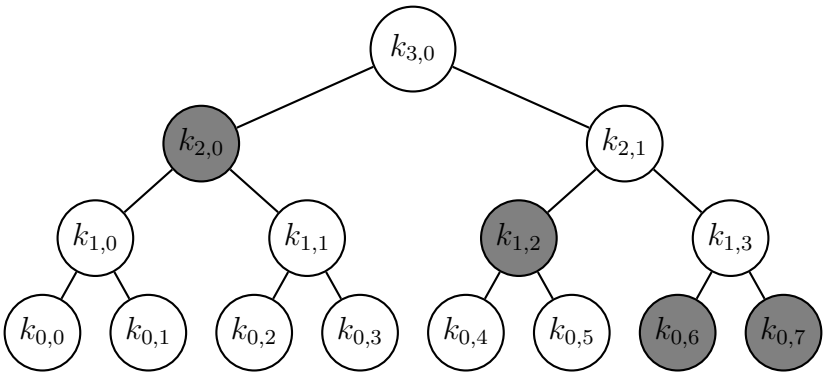


Abbildung 4.8: Worst Case Speichernutzung für Berechnung von $k_{3,0}$

3. Ist der Treehash Algorithmus abgeschlossen, entspricht $\text{Stack}_{h.\text{low}} = \infty$ .

Mithilfe dieser Funktion wird nun der neue Verifikationspfad Algorithmus vorgestellt, wobei der einzige Unterschied zu Merkles Algorithmus die Verteilung der Operationen bei Schritt 4 ist. Hier werden, statt einer gleichmäßigen Verteilung der Operationen, nun alle $2H$ Operationen auf die Instanz mit dem geringsten Wert für $\text{Stack}_{h.\text{low}}$ angewendet.

**Algorithmus 19** (Logarithmischer Merkle Verifikationspfad Algorithmus [16])**.**
Dieser Algorithmus ist bis auf Schritt 4 identisch zu Algorithmus 17.
4. Update Treehash Instanzen:
Wiederhole 2H-1 mal:

- Setze $l_{min}$ gleich dem kleinsten $\text{Stack}_{h.\text{low}}$.
- Setze $fokus$ gleich dem kleinsten $h$ für welches $\text{Stack}_{h.\text{low}} = l_{min}$ gilt.
- Führe eine Operation für $Stack_{fokus}$ durch.

Diese Veränderung bevorzugt Treehash Instanzen mit niedrigem $h$, außer eine andere Instanz hat einen Schweif mit einem tiefer gelegenen Knoten, welcher dann zuerst abgefertigt wird. Daraus folgt, dass eine Instanz der Ebene $h$ nur beendet und somit die nächste Iteration gestartet wird, wenn alle anderen aktiven Instanzen bis mindestens Höhe $h$ verhasht wurden.

Aus dieser komplexeren Verteilung der Operationen ist nun jedoch nicht mehr offensichtlich, dass jede Treehash Instanz in der Zeit bis ihr Knoten benötigt wird, genügend Operationen zugewiesen bekommt, um ihre Aufgabe vollständig zu erledigen. Die Autoren in [16] beweisen dies, indem sie aufzeigen, dass zum Zeitpunkt wo ein spezifischer Knoten benötigt wird, dieser und alle anderen berechenbaren Knoten mit der vorhandenen Menge an Operationen vollständig berechnet werden können. Menge an Operationen vollständig berechnet werden können.

Zuletzt wollen wir uns anschauen, wie stark unser voriger Worst Case mittels dieser Methode verbessert wurde. Hierzu summieren die Autoren die Menge der $\text{Auth}_h$ Knoten, der vollständig berechneten anstehenden Knoten und die maximale Größe aller Stacks.

1. Es existiert immer ein $\text{Auth}_h$ pro Ebene $\Rightarrow H$.
2. Die anstehenden Knoten existieren auf jeder Ebene außer $H \Rightarrow C < H$.
3. Für jede Ebene, wo der anstehende Knoten nicht fertig berechnet ist, existiert ein Stack mit maximal $h$ Einträgen. Der Algorithmus garantiert jedoch, dass eine $\text{Treehash}_h$ Instanz niemals Knoten generiert, solange eine höhere andere Instanz einen Knoten besitzt, welcher tiefer als h liegt. Hierdurch können zwei Knoten in verschiedenen Instanzen nicht auf derselben Ebene existieren. Zudem kann jede Instanz auf maximal einer Ebene zwei Knoten besitzen, wodurch wir zu maximal einem Knoten pro Ebene plus maximal einen Knoten pro aktive Instanz kommen $\Rightarrow H - 2 + (H - C)$.

$$H + C + (H - 2 + (H - C)) = 3H - 2 \approx 3H$$

Hiermit ist gezeigt, dass die Anzahl der maximal gespeicherten Knoten logarithmisch zur Anzahl der Blätter des Baumes wächst.

### 4.5.3 Fraktale Verifikationspfadberechnung

Ein weiteres Problem von Merkles Algorithmus ist, dass alle Ebenen unabhängig voneinander arbeiten und keine Zwischenergebnisse miteinander teilen. Dies führt dazu, dass jede Ebene alle Berechnungen der vorigen Ebene erneut durchführt. In diesem Abschnitt betrachten wir den in [8] vorgestellten Fraktalen Verifikationspfad Algorithmus, welcher solch eine Teilung der Zwischenergebnisse nutzt.

Die Idee des Algorithmus ist nicht eine Treehash Instanz für jede Ebene, sondern eine modifizierte Version des Treehash Algorithmus zu nutzen, um nur durch einen Bruchteil der Ebenen iterieren zu müssen. Hierzu wird ein Parameter $h < H$ gewählt, welcher den Baum in $L = H/h$ Level aufteilt, jedes dieser Level wird einen aktiven Subbaum enthalten. Nun wird für jedes Level eine Instanz des modifizierten Treehash Verfahrens initialisiert. Diese modifizierte Version von Treehash berechnet nicht die Wurzel, sondern hält eine Runde vorher an, sobald die beiden Kinder der Wurzel berechnet wurden. Zusätzlich verwirft der Algorithmus nicht alle verbrauchten Knotenwerte, sondern speichert alle Knoten, die maximal $h$ Ebenen von der Wurzel entfernt sind, zusammen als Subbaum ab.

Die Ursprungsebene der Subbäume wird so gewählt, dass ihre Wurzel einem Blatt des höher liegenden Subbaumes entspricht, aus diesem Grund wird die Wurzel weder berechnet noch mit ihren Kindern zusammen abgespeichert. Diese Verkettung an Subbäumen bildet einen vollständigen Weg von der ursprünglichen Wurzel des Baumes zum aktiven Blatt, für welches der momentane Verifikationspfad berechnet werden soll. Der Konstruktion folgt, dass der komplette Verifikationsalgorithmus für das Blatt im Speicher geladen ist und abgelesen werden kann. Darstellung 4.9 zeigt, wie ein solcher Stack an Subbäumen aussehen könnte und wie die Wurzel eines Subbaumes einem Blatt des darüber liegenden entspricht.

Ein Subbaum ist solange nützlich, bis der Verifikationspfad des nächsten Blattes nicht mehr durch diesen führt. Wie in anderen Algorithmen muss also hier der nächste Sub-

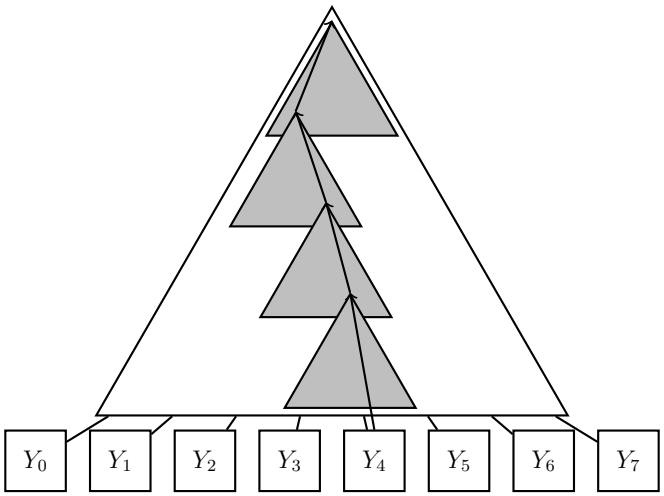


Abbildung 4.9: Mögliche Anordnung der Subbäume innerhalb eines Merkle Baumes

baum des Levels vorberechnet werden, die Wurzel des nächsten Subbaumes entspricht hierbei dem nächsten Blatt des ein Level höher liegenden Subbaumes.

**Definition 20** (Fraktaler Verifikationspfad Algorithmus [8])**.**

1. Schlüsselgeneration: Berechnung der Wurzel des Baumes, Speichern des ersten Subbaumes von jedem Level sowie Übergabe der Wurzel des zweiten Subbaumes jedes Levels an eine modifizierte Treehash Instanz.
2. Ausgabe: Rundenbasierte Ausgabe des Verifikationspfades für das momentane Blatt, Berechnung einiger Knoten des nächsten Subbaumes jeder Ebene sowie Verwerfung nicht mehr benötigter Knoten der momentanen Subbäume.
3. Verifikation: Prüfung, ob der zurückgegebene Pfad korrekt ist.

**Algorithmus 21** (Ausgabe Phase des Fraktalen Verifikationspfad Algorithmus [8])**.**

1. Setze $s = 0$.
2. Ausgabe:
   - Verifikationspfad für Blatt s
3. Ersetze Subbäume:
   Für alle $i \in 1, 2, ..., L$ für die s+1 durch $2^{i*h}$ teilbar ist:
   - Ersetze den momentanen Subbaum durch den Folge-Subbaum.
   - Initialisiere $\text{Treehash}_i$ zur Generation des nächsten Folge-Subbaums.
4. Update Subbäume:
   Für jedes $i \in 1, 2, ..., L$:
   - Entferne nicht mehr benötigte Knoten des aktiven Subbaumes.
   - Führe zwei Operationen für den Folge-Subbaum aus, solange dieser noch nicht komplett ist.
5. Setze $s = s + 1$.
6. Solange $s \leq 2^H$ gehe zu Schritt 2.

Wie bei vorigen Algorithmen ist es ausreichend jeder Treehash Instanz 2 Operationen pro Runde zuzuweisen, da diese modifizierte Version sogar eine Runde weniger arbeitet als in Merkles Algorithmus. Da die oberste Ebene den Subbaum der ursprünglichen Wurzel enthält, existiert kein Folge-Subbaum, wodurch wir

$$(2L - 1) < 2H/h$$

Operationen pro Runde haben im Vergleich zu Merkles $2H$ pro Runde. Dieser Algorithmus vermindert den benötigten Rechenaufwand um einen Faktor von $h$, da wir nur alle $h$ Ebenen, eine Instanz von Treehash mit derselben Menge an Operationen pro Runde benutzen, wie sie der ursprüngliche Merkle Algorithmus für jede Ebene benutzt [8].

Er verbraucht jedoch, je nach gewähltem $h$, eine deutliche Menge an zusätzlichem Speicherplatz, da bis zu $L$ Subbäume der Tiefe $h$, $L-1$ Folge-Subbäume der Tiefe $h$ sowie

die noch nicht verarbeiteten Schweife der Folge-Subbäume im Worst Case gleichzeitig existieren. Hieraus folgt ein notwendiger Speicherplatz von etwa

$$2L2^{h+1} + HL/2[8]$$

Diese Formel ist leicht schlechter als der wahre notwendige Speicherplatz aber deutlich übersichtlicher. Die $2L2^{h+1}$ entsprechen in etwa den $L$ sowie $L-1$ Subbäumen, welche jeweils $2^{h+1}$ Knoten maximal enthalten, während $H * L/2$ ein wenig über der maximalen Länge aller Schweife liegt. Für hohe $h$ wird der Speicherverbrauch sehr groß, da wir große Teile des Baumes in unseren Subbäumen aktiv haben. Den Autoren des Algorithmus nach wird für $h = logH$ der geringste Speicherplatz verbraucht, während die Rechenleistung im Vergleich zu Merkle immer noch durch wenigstens $logH$ geteilt wird.

## 4.6 Sicherheit des Merkle-Baum Verfahrens

In diesem Abschnitt werfen wir einen Blick auf die Sicherheit des unmodifizierten Merkle-Baum Verfahrens, anhand der in [1, S.81-91] dargestellten Forschung. Die Autoren betrachten hierbei das Verfahren im Bezug einer chosen message attack. Hier hat der Angreifer Zugriff auf den öffentlichen Schlüssel und eine bestimmte Anzahl an Nachrichten seiner Wahl, deren Signaturen er kennt und welche erfolgreich mit dem öffentlichen Schlüssel verifizieren. Bei einem Angriff auf das Einmalsignaturverfahren hat er Zugriff auf eine Nachricht seiner Wahl und deren Signatur, während er beim Merkle-Baum Verfahren Zugriff auf $2^H$ dieser hat. Das Ziel des Angreifers ist es, nach Kenntnis all dieser, eine einzelne andere Nachricht und Signatur zu generieren, welche erfolgreich mit dem öffentlichen Schlüssel verifiziert. Die Kombination aus dem nötigen Aufwand eines Versuches sowie der zugehörigen Erfolgschance ergibt hierauf die Menge an Rechenaufwand der benötigt wird, um das Signaturverfahren zu umgehen. Im Laufe der Arbeit reduzieren die Autoren diesen Aufwand auf die Kollisionsresistenz der im Merkle-Baum Verfahren genutzten Hashfunktion sowie auf die Unumkehrbarkeit der im genutzten Einmalsignaturverfahren verwendeten Einwegfunktion.

Desweiteren berechnen sie das Sicherheitslevel $b$, indem sie die nötige Zeit $t$ eines Angriffs durch die errechnete Erfolgschance $\epsilon$ teilen.

$$2^b = \frac{t}{\epsilon}$$

Für klassische Computer nehmen sie an, dass die Funktionen mittels der Brute-Force Methode oder der Geburtstags Attacke gelöst werden. Aus den Erfolgschancen dieser errechnen sie, dass das Sicherheitslevel $b$ des Merkle-Baum Verfahrens mit dem Sicherheitsparameter $n$ mindestens $b = \frac{n}{2} - 1$ beträgt, solange $H \leq \frac{n}{3}$ und $n \geq 87$ gewählt werden.

Sollten Quantencomputer ausreichend Q-Bits erreichen, um Grover's Algorithmus ausführen zu können, muss dessen Effizienz in die obige Berechnung mit einbezogen werden, hierdurch wird die Resistenz unserer Hash- und Einwegfunktionen verringert. Aufgrund dieses Algorithmus verändert sich die Sicherheit des Merkle-Baum Verfahrens

| $n$ | 128 | 160 | 224 | 256 | 384 | 512 |
|---|---|---|---|---|---|---|
| Klassisch | | | | | | |
| minimum b | 63 | 79 | 111 | 127 | 191 | 255 |
| maximales H | 42 | 53 | 74 | 85 | 128 | 170 |
| Quantum | | | | | | |
| minimum b | - | - | 73 | 84 | 127 | 169 |
| maximales H | - | - | 56 | 64 | 96 | 128 |

Abbildung 4.10: Sicherheitslevel und maximale Höhe des Merkle-Baum Verfahrens bei spezifischen $n$ [1]

zu $b = \frac{n}{3} - 1$ solange $H \leq \frac{n}{4}$ und $n \geq 196$. In Tabelle 4.10 zeigen wir eine Übersicht über das Sicherheitslevel und das maximale $H$ für einige spezifische $n$. Aus diesen Werten können wir schließen, dass unser Verfahren zu jetzigem Stand gegenüber Klassischen- und potenziellen Quantencomputern sicher ist, solange wir moderne Hash- und Einwegfunktionen mit mindestens 256-Bit Ausgabelänge benutzen.

## 4.7 Effizienz des Merkle-Baum Verfahrens

Das Merkle-Baum Verfahren wurde seit seiner Einführung nicht im großen Stil benutzt, da es in Sachen Effizienz nicht konkurrenzfähig zu RSA und anderen Signaturverfahren war. Nun stellt sich jedoch die Frage, ob die hier vorgestellten Modifikationen an diesem Punkt etwas geändert haben. Aus diesem Grund stellen wir hier die Vergleiche der Autoren von CMSS [4] und GMSS [2] vor. CMSS nutzt das Winternitz Einmalsignaturverfahren, Tree Chaining, Schlüsselgeneration mithilfe eines Zufallsgenerators sowie Szydlos Verifikationspfadberechnung. GMSS ist eine Weiterentwicklung von CMSS mit den Modifikationen aus der verteilten Signaturgeneration. Den Autoren dieser beiden Verfahren zufolge hat Merkles originales Verfahren eine Grenze von $2^{25}$ Schlüsselpaaren, bevor deren Länge und Rechenaufwand untragbar wird [4, S.353-354]. CMSS erweitert dies zu $2^{40}$ [4, S.353-354] und GMSS zu $2^{80}$ [2, S.32] was für jegliche Anwendungen ausreichen sollte. Um diese beiden Verfahren nun mit RSA, DSA und ECDSA zu vergleichen, beziehen wir uns auf die experimentellen Ergebnisse der Autoren in Tabelle 4.11 und 4.12.

Die Ergebnisse in Tabelle 4.11 wurden mit einem 1.73 Pentium M Prozessor, 1GB RAM und Windows XP erzielt. Die erste Spalte gibt die Anzahl der möglichen Signaturen von CMSS mit $2^H$ an sowie den modulo für RSA, DSA und ECDSA, welcher bestimmt wie groß die zu verschlüsselnde Nachricht sein kann. Uns fällt auf, dass die Schlüsselpaar Erstellung bei CMSS mit $H = 40$ sehr lange dauert. Dies ist jedoch vertragbar da es ein einziges mal durchgeführt werden muss, um $2^{40}$ Signaturen zu erstellen, dies ermöglicht in etwa eine Trillion Signaturen mit den gegebenen Werten. Des Weite-

ren hat CMSS einen deutlich kleineren öffentlichen Schlüssel, aber auch einen deutlich

| $H$ | $m_{publickey}$ | $m_{privatekey}$ | $m_{signature}$ | $t_{keygen}$ | $t_{sign}$ | $t_{verify}$ |
|---|---|---|---|---|---|---|
| CMSS mit SHA1 Hashfunktion, w=1 | | | | | | |
| 20 | 46 bytes | 1900 bytes | 7168 bytes | 2.9 s | 10.2 ms | 1.2 ms |
| 30 | 46 bytes | 2788 bytes | 7368 bytes | 1.5 min | 13.6 ms | 1.2ms |
| 40 | 46 bytes | 3668 bytes | 7568 bytes | 48.8 min | 17.5 ms | 1.2ms |
| CMSS mit SHA1 Hashfunktion, w=2 | | | | | | |
| 20 | 46 bytes | 1900 bytes | 3808 bytes | 2.6 s | 9.2 ms | 1.3 ms |
| 30 | 46 bytes | 2788 bytes | 4008 bytes | 1.4 min | 12.4 ms | 1.4ms |
| 40 | 46 bytes | 3668 bytes | 4208 bytes | 43.8 min | 14.9 ms | 1.3ms |
| CMSS mit SHA1 Hashfunktion, w=3 | | | | | | |
| 20 | 46 bytes | 1900 bytes | 2688 bytes | 3.1 s | 9.7 ms | 1.5 ms |
| 30 | 46 bytes | 2788 bytes | 2888 bytes | 1.5 min | 13.2 ms | 1.5ms |
| 40 | 46 bytes | 3668 bytes | 3088 bytes | 47.8 min | 16.9 ms | 1.6ms |
| CMSS mit SHA1 Hashfunktion, w=4 | | | | | | |
| 20 | 46 bytes | 1900 bytes | 2128 bytes | 4.1 s | 12.5 ms | 2.0 ms |
| 30 | 46 bytes | 2788 bytes | 2328 bytes | 2.0 min | 17.0 ms | 2.0 ms |
| 40 | 46 bytes | 3668 bytes | 2528 bytes | 62.3 min | 21.7 ms | 2.0 ms |
| $mod$ | $m_{publickey}$ | $m_{privatekey}$ | $m_{signature}$ | $t_{keygen}$ | $t_{sign}$ | $t_{verify}$ |
| RSA mit SHA1 Hashfunktion | | | | | | |
| 1024 | 162 bytes | 634 bytes | 128 bytes | 0.4 s | 13.8 ms | 0.8 ms |
| 2048 | 294 bytes | 1216 bytes | 256 bytes | 3.4 s | 96.8 ms | 3.0 ms |
| DSA mit SHA1 Hashfunktion | | | | | | |
| 1024 | 440 bytes | 332 bytes | 46 bytes | 18.2 s | 8.2 ms | 16.2 ms |
| ECDSA mit SHA1 Hashfunktion | | | | | | |
| 192 | 246 bytes | 231 bytes | 55 bytes | 5.1 ms | 5.1 ms | 12.9ms |
| 256 | 311 bytes | 287 bytes | 71 bytes | 9.6 ms | 9.8 ms | 24.3 ms |
| 384 | 441 bytes | 402 bytes | 102 bytes | 27.3 ms | 27.3 ms | 66.9 ms |

Abbildung 4.11: Schlüssellängen und Timings für CMSS, RSA, DSA und ECDSA auf einem Pentium M 1.73GHz Prozessor, 1GB RAM und Windows XP [4] S.359

| | $m_{offline}$ | $m_{privkey}$ | $m_{signature}$ | $t_{keygen}$ | $t_{sign}$ | $t_{verify}$ |
|---|---|---|---|---|---|---|
| $P_{40}$ | 3160 bytes | 1640 bytes | 1860 bytes | 723 min | 26.0 ms | 19.6 ms |
| $P'_{40}$ | 3200 bytes | 1680 bytes | 2340 bytes | 390 min | 10.7 ms | 10.7 ms |
| $P_{80}$ | 7320 bytes | 4320 bytes | 3620 bytes | 1063 min | 26.1 ms | 18.1 ms |
| $P'_{80}$ | 7500 bytes | 4500 bytes | 4240 bytes | 592 min | 10.1 ms | 10.1 ms |

$P_x$ = Anzahl der Ebenen, ($H$ der Bäume), ($w$ der Bäume)

$P40 = 2, (20, 20), (10, 5)$ und $P80 = 4, (20, 20, 20, 20), (8, 8, 8, 5)$

$P'40 = 2, (20, 20), (9, 3)$ und $P'80 = 4, (20, 20, 20, 20), (7, 7, 7, 3)$

Abbildung 4.12: Schlüssellängen und Timings für GMSS auf einem Pentium Dualcore 1.8GHz Prozessor mit einer 160-Bit Hashfunktion[2] S.41

größeren privaten Schlüssel und vor allem Signaturen von etwa der zehnfachen Länge. Dieser Speicheraufwand kann dazu führen, dass CMSS für stark Speicher begrenzte Geräte ungeeignet ist und bei ähnlicher Effizienz zu einem anderen Verfahren nicht gewählt wird. Der Zeitaufwand zum Signieren einer Nachricht ist sehr ähnlich zu RSA, DSA und ECDSA, während die Verifikationszeit kürzer als alles bis auf RSA mit kleinem modulo ist. Hieraus können wir schlussfolgern, dass CMSS in Rechenaufwand sehr nah an RSA, DSA und ECDSA herangetreten ist und nur einen Nachteil im Speicherverbrauch mit sich zieht. Sollten Quantencomputer Angriffe auf RSA, DSA und ECDSA stark vereinfachen, so scheint CMSS schon jetzt eine geeignete Alternative zu sein.

Die Ergebnisse zu GMSS wurden auf einem Pentium Dualcore mit 1.8 GHz mit einer 160-Bit Hashfunktion durchgeführt. Hier wurden uns jedoch von den Autoren keine Vergleichswerte für RSA, DSA, ECDSA oder sogar CMSS geliefert, weswegen wir dies nicht direkt mit den Werten aus der vorigen Tabelle vergleichen können. In GMSS haben wir deutlich mehr Wahl wie wir unseren Rechenaufwand verteilen wollen, um Speicherplatz zu sparen. $P_{40}$ und $P_{80}$ sind Durchführungen des Verfahrens mit höheren Werten für $w$, wodurch diese Speicherplatz sparen. $P'_{40}$ und $P'_{80}$ dagegen nutzen kleinere $w$, wodurch sie doppelt so schnelle Timings erzielen aber auch etwas größeren Speicherverbrauch besitzen. Im Allgemeinen jedoch besitzt dieses Verfahren einen kleineren Speicherverbrauch als CMSS bei derselben Anzahl an Schlüsselpaaren ($2^{40}$). Es ermöglicht sogar effektiv unerschöpfliche $2^{80}$ Schlüsselpaare bei etwas schlechterem Speicherverbrauch als CMSS. GMSS benötigt sehr lange zum Erstellen der initiallen Schlüsselpaare, noch deutlich länger als CMSS, jedoch gilt auch hier, dass ein einmaliges Generieren effektiv unlimitiert Signaturen ermöglicht. Leider können wir, aufgrund der Nutzung eines anderen Prozessors, die Timings nicht direkt mit CMSS vergleichen. Wir können jedoch sagen, dass diese in einem akzeptablem Rahmen scheinen, vor allem da wir größere $w$ in diesem Experiment nutzen und Winternitz für jede Steigerung von $w$ um eins seinen benötigten Rechenaufwand verdoppelt. Die Möglichkeit noch besser zwischen Speicherverbrauch

und guten Timings optimieren zu können ist ein weiterer deutlicher Vorteil. Allgemein scheint GMSS eine moderate Verbesserung des Speicherverbrauchs sowie deutlich bessere Optimierungsmöglichkeit für den Anwender zu bieten. Zusätzlich löst es die in Kapitel 4.4 beschrieben Probleme von CMSS. Vorher kamen wir schon zu dem Schluss, dass CMSS, wenn nötig, ein akzeptabler Ersatz für RSA, DSA und ECDSA sein sollte, aus obigen Punkten ist GMSS eine Verbesserung dessen und somit eine noch bessere Option.

# 5 Fazit

In dieser Arbeit haben wir uns dem Problem der anstehenden Post-Quanten Wende durch die Entwicklung nutzbarer Quantencomputern gewidmet und wie wir dessen negative Auswirkungen für die Kryptographie mindern können. Da die Kryptographie nicht nur möglichst effizient sondern auch für lange Zeit sichere Verfahren bieten muss, ist es wichtig, dass wir die Auswirkungen neuer Erkenntnisse frühzeitig untersuchen und sinnvolle Alternativen für Nutzer vorbereiten. Speziell haben wir uns auf die Auswirkungen bezüglich Signaturverfahren konzentriert, welche essenziell für den reibungslosen Ablauf unserer heutigen digitalen Gesellschaft sind.

Über die letzten Jahre gab es konstante Fortschritte in der Kryptoanalyse der effizientesten Signaturverfahren, womit bereits die Lösung mittels klassischer Computer näher und näher gerückt ist. Hierdurch wurde die Effizienz der führenden Verfahren mehr und mehr gemindert, damit diese sicher bleiben. Kombiniert mit der Existenz von Shor's Algorithmus sieht es so aus, als wären unsere besten momentanen Signaturverfahren lösbar, sobald Quanten-Computer über ausreichend Q-Bits zur Ausführung dieses Algorithmus verfügen.

Die Zeit die uns noch bleibt, bis so ein Quanten-Computer geschaffen wurde, sollten wir nutzen um vielversprechende Alternativen zu finden, zu untersuchen und zu verbessern. Als mögliche Alternative haben wir die Hash-basierten Signaturverfahren vorgestellt, welche auf einem alternativen weniger gelösten Problem basieren und weniger anfällig gegenüber Quanten-Computern scheinen.

Diese Familie erfuhr keine weitverbreitete Nutzung, obwohl sie schon vor 40 Jahren mit dem Lamport-Diffie und Merkle-Baum Verfahren vorgestellt wurde, da ihre Effizienz im Vergleich zu den Vorreitern nicht ausreichend war. Im weiteren Verlauf der Arbeit haben wir jedoch eine Vielzahl an Modifikationen vorgestellt, welche über die Jahre entwickelt wurden und eine deutliche Verbesserung der Rechen- und Speichereffizienz mit sich bringen. Die Verbesserungen modernerer Versionen wie CMSS und GMSS und die Fortschritte in der Lösung von RSA, DSA und ECDSA haben dazu geführt, dass Hash-basierte Signaturverfahren heutzutage konkurrenzfähig scheinen. Aufgrund dessen scheint es sinnvoll zusätzlichen Aufwand in die Kryptoanalyse und Weiterentwicklung der Hash-basierten Signaturverfahren zu investieren. Damit wir eine sicher erscheinende Alternative besitzen, sobald Shor's Quanten-Algorithmus aktiv genutzt werden kann. Auch falls dies zum Finden eines kritischen Angriffsvektors führt müssen wir jetzt anfangen uns mit Post-Quanten Verfahren zu beschäftigen, da uns sonst die Zeit ablaufen wird.

# Literaturverzeichnis


[1] Buchmann, J., Dahmen, E., and Bernstein, D. (2009). Post-quantum cryptography. pages 1–13,35–93.

[2] Buchmann, J. A., Dahmen, E., Klintsevich, E., Okeya, K., and Vuillaume, C. (2007). Merkle signatures with virtually unlimited signature capacity. In *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007, Proceedings*, pages 31–45.

[3] Buchmann, J. A., Dahmen, E., and Schneider, M. (2008). Merkle tree traversal revisited. In *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, pages 63–78.

[4] Buchmann, J. A., García, L. C. C., Dahmen, E., Döring, M., and Klintsevich, E. (2006). CMSS - an improved merkle signature scheme. In *Progress in Cryptology - INDOCRYPT 2006, 7th International Conference on Cryptology in India, Kolkata, India, December 11-13, 2006, Proceedings*, pages 349–363.

[5] Dods, C., Smart, N. P., and Stam, M. (2005). Hash based digital signature schemes. In *Cryptography and Coding, 10th IMA International Conference, Cirencester, UK, December 19-21, 2005, Proceedings*, pages 96–115.

[6] Gamal, T. E. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4):469–472.

[7] Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219.

[8] Jakobsson, M., Leighton, F. T., Micali, S., and Szydlo, M. (2003). Fractal merkle tree representation and traversal. In *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, pages 314–326.

[9] Johnson, D., Menezes, A., and Vanstone, S. A. (2001). The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Sec.*, 1(1):36–63.

[10] Lamport, L. (1979). Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto.

[11] Menezes, A., van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press.

[12] Merkle, R. C. (1989). A certified digital signature. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 218–238.

[13] Rabin, M. O. (1979). Digitalized signatures and public-key functions as intractable as factorization. Technical report, Massachusetts Inst of Tech Cambridge Lab for Computer Science.

[14] Rivest, R. L., Shamir, A., and Adleman, L. M. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

[15] Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 124–134.

[16] Szydlo, M. (2004). Merkle tree traversal in log space and time. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 541–554.