Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Bachelorarbeit Signieren von Dokumenten mit ECDSA und Hashbäumen

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: David Wedekind
geboren am:
geboren in:

Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Andreas Hallof

eingereicht am: ..... verteidigt am: .....

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Digitale Signaturen gewinnen immer mehr an Bedeutung. So werden sie beim Transport von Nachrichten und Dokumenten (beispielsweise beim E-Rezept), bei der Datenarchivierung und weiteren Bereichen verwendet. Für digitale Signaturen werden häufig spezielle Geräte, so genannte Hardware Security Module (HSMs), verwendet. Diese sind sowohl in der Anschaffung als auch im Betrieb ein signifikanter Kostenfaktor. Es werden davon i. d. R. mehrere benötigt, um eine notwendige Anzahl von Signaturen pro Zeiteinheit – insbesondere zu Stoßzeiten – erzeugen zu können. Um die hohen Kosten senken zu können, lohnt es sich, Techniken zu erwägen, welche die Signatur von mehr Dokumenten ermöglichen. Eine naheliegende Möglichkeit ist das Erwerben von mehr Hardware; diese ist jedoch teuer und dieses Vorgehen skaliert nicht gut. Es gibt andere Lösungsansätze, welche keine höheren Investitionen in die Hardware benötigen, sondern versuchen das Problem auf der Softwareseite zu lösen. Eine vielversprechende Möglichkeit ist die Verwendung von Merkle-Bäumen, um viele Dokumente unter einer einzigen Signatur zu vereinen. Die Struktur Merkle-Baum ist schon erprobt und wird in verschiedenen Technologien eingesetzt. So wird sie beispielsweise im RFC 8391 [1] zu XMSS, in Git, dem Certificate-Transparency-Projekt und der Blockchain verwendet. Unter Verwendung eines HSM ist davon auszugehen, dass man ca. 680 Signaturen pro Sekunde [2] erzielen kann. Mit Hilfe von Merkle-Hashbäumen kann man mit nur einem HSM über 80.000 Signaturen pro Sekunde erreichen. Weiterhin skaliert die Lösung mittels Merkle-Hashbäumen sehr gut. Andreas Hallof von der gematik hatte die Idee, diese Technologie für das elektronische Rezept zu verwenden, da dort besonders viele Signaturen mit HSMs benötigt werden.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es, eine Implementierung zu entwickeln, welche es möglich macht, eine sehr große Zahl von Dokumenten mithilfe von nur einer, im HSM erzeugten Signatur, zu signieren und diese auch wieder zu verifizieren. Das soll über die Datenstruktur Merkle-Baum realisiert werden. Nicht vorhandene Algorithmen, die für diese Anwendung gebraucht werden, sollen erstellt werden. Verschiedene Umsetzungsoptionen werden in Bezug auf Performanz und Sicherheit verglichen, um ein möglichst effizientes Programm zu erstellen. Außerdem soll dieses Programm, in

Bezug auf die Geschwindigkeit, mit der Signatur einzelner Dokumente über Hardware Security Module, verglichen werden. Das Vereinen mehrerer Dokumente unter einer Signatur bringt verschiedene Herausforderungen mit sich. Eine für meinen Anwendungsfall sehr relevante ist, dass ein Dokument oder eine Teilmenge verifiziert werden kann, ohne dass potentiell Rückschlüsse auf andere Dokumente gezogen werden können. Außerdem sollen alle Sicherheitsleistungen erhalten bleiben, die eine „normale“ ECDSA-Signatur bereitstellt. Die Authentizität und Nichtabstreitbarkeit für den Signierenden, sowie die Integrität der signierten Dokumente sollen also gewährleistet sein.

1.3 Aufbau der Arbeit

Zunächst werde ich einige technische Grundlagen definieren, welche ich in meiner Implementation verwendet habe. Dann benenne ich bestehende Anwendungen der Struktur Merkle-Baum und warum diese, auch für meinen Anwendungsfall, nützlich ist. Im darauffolgenden Abschnitt beschreibe ich das elektronische Rezept, welches Hauptbestandteil der Motivation meiner Arbeit ist. Dann gehe ich auf meine Proof-of-Concept-Implementierung, sowie damit erstellte Benchmarks, ein. Alle relevanten Dateien, welche ich zum Erstellen dieser Arbeit verwendet habe befinden sich im öffentlichen Git-Repository `https://scm.cms.hu-berlin.de/wedekind/bachelorarbeit` [3]

2 Theoretische Grundlagen

2.1 Kryptografisches Hashing

Eine Hashfunktion ist ein mathematischer Algorithmus, welcher Daten beliebiger Größe abgesehen von maximalen Eingabelängen [4] auf ein Bitarray festgelegter Größe abbildet. Der Algorithmus muss deterministisch sein und um eine kryptografische, kollisionsresistente Hashfunktion zu sein, zusätzlich folgende Eigenschaften erfüllen:

- Einwegfunktion: Es ist praktisch unmöglich, zu einem Hash seinen Eingabewert zu berechnen.
- Schwache Kollisionsresistenz: Es ist praktisch unmöglich, zu einem gegebenem Eingabewert einen zweiten Eingabewert zu finden, der den gleichen Hash ergibt.
- Starke Kollisionsresistenz: Es ist praktisch unmöglich, zwei frei gewählte Eingabewerte zu finden, die den gleichen Hashwert ergeben. [5] [6]

Für meine Anwendung benötige ich vor allem die Eigenschaft der Einwegfunktion und dass der Algorithmus deterministisch ist.

2.2 ECDSA

Der Elliptic Curve Digital Signature Algorithm (ECDSA) [7] ist eine Variante des Digital Signature Algorithm (DSA) [4] mit der Verwendung von elliptischen Kurven über endliche Körper.

Elliptische Kurven Man hat die Kurve E und einen designierten Punkt G auf E, genannt der Basispunkt. Der Basispunkt hat Ordnung n, n ist eine große Primzahl. Die Zahl der Punkte auf der Kurve ist $h * n$. h soll nicht von n geteilt werden können und so klein wie möglich sein, meist 1,2 oder 4. [4]

Schlüsselerzeugung Die Erzeugung von Private/Public Schlüsselpaaren, welche für Signatur und Verifikation in ECDSA verwendet werden, basiert auf den sogenannten Domain Parametern
$(q, FR, a, b, DomainParameterSeed, G, n, h)$. q, FR, a und b beschreiben die Kurve. Der $DomainParameterSeed$ ist eine zufällig erzeugte Zeichenkette, die vorliegt, wenn die Kurve nachweislich zufällig erzeugt wurde. Hierfür wird die Kurve aus

dem $DomainParameterSeed$ in solch einer Weise generiert, dass der Nachweis leicht über dieselbe Rechnung, welche auch zur Generierung verwendet wird, nachvollzogen werden kann. Je nach Methode genügt ein einfacher Vergleich bestimmter Parameter oder eine Modulo-Operation, die bestimmte Parameter beinhaltet, um die zufällige Generierung mit dem $DomainParameterSeed$ nachzuweisen. Dann muss noch aufgezeigt werden, dass der $DomainParameterSeed$ ohne eine besondere Einflussnahme oder für den Einbau von Hintertüren gewählt ist, was zum Beispiel durch Wahl einer sinnvollen Nachkommastelle in einer irrationalen Zahl einfach zu zeigen ist. Weiterhin ist G ein fester Erzeuger der n-Torsionsuntergruppe der Kurve, n die Ordnung des Punktes G und h der Cofaktor.

Für die Schlüsselpaarerzeugung sind dann verschiedene Methoden möglich, z.B. mit Verwendung von extra zufälligen Bits oder mittels Testen von Kandidaten. [4]

Algorithmus für die Schlüsselpaarerzeugung Ein einfacher Algorithmus für die Schlüsselpaarerzeugung mit Generierung von zufälligen Bits kann folgendermaßen erfolgen:

1. Wähle zufällige ganze Zahl $d \in [1, n-1]$.
2. Berechne $Q = dG$.
3. Der öffentliche Schlüssel ist Q; der private Schlüssel ist d. [7]

Signaturerzeugung Für diese Arbeit ist vor allem die eigentliche Signaturerzeugung interessant, weil ich diese in meiner Implementation verwende. Ein einfacher Algorithmus sieht folgendermaßen aus:

Algorithmus für die Signaturerzeugung Benötigt wird:

- Domain Parameter, spezifiziert wie im Paragraphen „Schlüsselpaarerzeugung“ des Abschnitts 2.2.
- Ein privater Schlüssel d, generiert wie im Paragraphen „Schlüsselpaarerzeugung“ des Abschnitts 2.2.
- Eine für jede Nachricht neu erzeugte Geheimzahl.
- Eine zugelassene Hashfunktion.
- Ein zugelassener Zufallszahlengenerator.

Sei m eine zu signierende Nachricht:

1. Wähle eine zufällige ganze Zahl $k \in [1, n-1]$.
2. Berechne $kG = (x_1, y_1)$.
3. Berechne $r = x_1 \bmod n$. Wenn $r = 0$, gehe zu Schritt 1 zurück.
4. Berechne $k^{-1} \bmod n$.
5. Berechne HASH(m) und konvertiere das Ergebnis in eine ganze Zahl e.
6. Berechne $s = k^{-1}(e + dr) \bmod n$. Wenn $s = 0$, gehe zu Schritt 1. zurück.
7. Die Signatur der Nachricht m ist das Paar (r, s). [7]

Algorithmus für die Verifikation Um die Signatur (r, s) von der Nachricht m zu verifizieren, benötigt man die Domain Parameter und den öffentlichen Schlüssel Q des Schlüsselpaares, welches zur Signaturerzeugung verwendet wurde.

1. Verifiziere, dass r und s im Intervall $[1, n-1]$ liegen. Wenn nicht $\rightarrow$ Signatur ablehnen.
2. Berechne $e = \text{HASH}(m)$.
3. Berechne $w = s^{-1} \bmod n$.
4. Berechne $u_1 = ew \bmod n$ und $u_2 = rw \bmod n$.
5. Berechne $X = u_1G + u_2Q$. Wenn $X = 0 \rightarrow$ Signatur ablehnen.
6. Berechne $v = x_1 \bmod n$, wobei $X = (x_1, y_2)$.
7. Akzeptiere die Signatur, wenn $v = r$.[7]

2.3 Merkle-Bäume

Ein Baum besteht aus einer Menge von Knoten und einer Menge von Kanten, welche jeweils zwei Knoten verbinden. Ein bestimmer Knoten wird Wurzel genannt. Jeder Knoten, mit Ausnahme der Wurzel, ist durch eine Kante mit genau einem Elternknoten verbunden. Der Knoten selbst wird dann als Kind bezeichnet. Der Wurzelknoten hat keinen Elternknoten. Ein eindeutiger Pfad verläuft von der Wurzel zu jedem Knoten im Baum. Wenn jeder Knoten im Baum maximal zwei Kinder hat, wird dieser Binärbaum genannt. Ein Binärbaum ist vollständig, wenn alle Blätter die gleiche Tiefe haben und jeder Knoten, außer den Blättern, zwei Kinder hat.

Merkle-Bäume, auch Hashbäume genannt, sind Bäume, dessen Blätter Hashes von Daten sind und alle Knoten, die keine Blätter sind, Hashes aus deren Kindern, wie in Abbildung 1. Die Wurzel ist dann der Master-Hash des Baumes. [8]

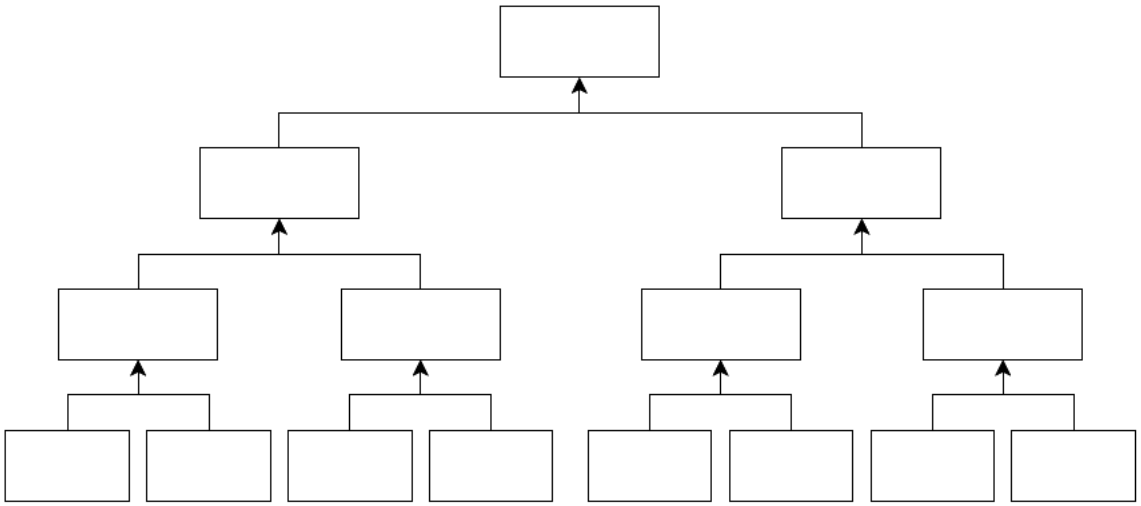

Abbildung 1: Merkle-Baum

Erstellen eines Merkle-Baumes Das Erstellen des Baumes wird in zwei Schritte gegliedert. Erst müssen die Blätter erstellt werden und danach können die höheren Knoten berechnet werden.

Um den Baum zu erstellen, werden in meinem Fall alle Dokumente, welche dem Baum hinzugefügt werden sollen, gehasht und als Blätter hinzugefügt. Jedes so hinzugefügte Dokument bekommt ein zufällig generiertes Partnerblatt. Dann werden so viele Dummydokumente, ebenfalls mit Partnerblatt, hinzugefügt, sodass die Blätteranzahl einer Zweierpotenz entspricht. Anschließend wird von benachbarten

Blättern ein Hash berechnet, wie in Abbildung 1 zu sehen. Das passiert rekursiv so lange, bis nur noch ein Hash in der obersten Ebene übrig bleibt. Dieser Knoten ist die Wurzel des Hashbaumes. [8] Man könnte auch direkt die Dokumente als Blätter verwenden, ohne sie zu hashen und auf Privatsphäre verzichten. Ich verwende Hashes, weil es, für meinen Anwendungsfall, keine Option ist, auf Privatsphäre zu verzichten; außerdem kann man so die Knotengröße vereinheitlichen und der Baum hat, sollte man ihn speichern wollen, eine einheitliche und in vielen Fällen deutlich kleinere Größe.

Authentifizierung eines Dokuments im Merkle-Baum Die Authentifizierung eines einzelnen Dokuments oder einer Teilmenge von Dokumenten kann erfolgen, ohne dass der ganze Baum benötigt wird. Man braucht nur den Pfad vom Blatt des Dokuments zur Wurzel zu finden und muss dann immer den jeweiligen Geschwisterknoten der Knoten auf diesem Pfad speichern. Diese Geschwisterknoten reichen aus, um einen Teilbaum zu erstellen, der die Authentifizierung möglich macht. Ich nenne diesen Pfad weiterhin Verifizierungspfad. Dann kann man, angefangen mit dem ersten Knoten aus dem Verifizierungspfad, wie beim Erstellen des Baumes, aus diesem Knoten und dem Dokumentenhash einen neuen Hash berechnen. Mit dem neuen Hash und dem nächsten Knoten aus dem Verifizierungspfad wird wieder ein neuer Hash berechnet. Das passiert für alle Knoten aus dem Verifizierungspfad. Den entstehenden Hash kann man dann mit dem bekannten Wurzelhash des Merkle-Baumes vergleichen. Wie das genau gemacht wird, ist aber implementationsspezifisch. [8] [9]

2.4 Hardware Security Module

Ein Hardware Security Modul (HSM) ist ein physisches Gerät, welches digitale Schlüssel schützt und verwaltet, Ver- und Entschlüsselung, digitale Signatur, starke Authentifizierung und andere kryptografische Funktionen ausführen kann. In der Regel können private Schlüssel in ein HSM importiert oder im HSM sicher erzeugt und verwendet, jedoch nicht exportiert werden. HSMs können Features haben, die Manipulation erkennen, Manipulation so schwer wie möglich machen und auch auf Manipulation reagieren, zum Beispiel indem Schlüssel bei erkannter Manipulation gelöscht werden. Kryptografische Algorithmen, die typischerweise in HSMs implementiert werden sind:

1. Hashfunktionen
2. Symmetrische Ver- und Entschlüsselung
3. Asymmetrische Kryptosysteme (RSA, ECDSA)
4. Erzeugung von Zufallszahlen, Schlüsseln und Pins

HSMs werden aufgrund ihrer kritischen Rolle von international anerkannten Standards wie „FIPS 140-2“ zertifiziert. Das stellt Nutzern des HSMs eine unabhängige Zusicherung zur Verfügung, dass die Hardware und implementierten Algorithmen verlässlich und die privaten Schlüssel vor Zugriff geschützt sind. [10]

3 Bestehende Verwendung von Hashbäumen und Merkle-Bäumen

Blockchain Hashbäume werden in der Blockchain verwendet. Zum Beispiel bei Bitcoin, Ethereum, IOTA und Apache Cassandra. Im Fall von Bitcoin bekommt jede Transaktion seinen eigenen Hash. Dann werden 1MB an Transaktionen gesammelt, bis ein Block voll ist. Jetzt wird ein Hashbaum aufgebaut und für den ganzen Block bleibt noch ein Hash übrig. Die Wurzel des aufgebauten Baumes wird dann verwendet, um einen einzigartigen Hashblock zu erzeugen und zwar mit Hilfe vom Hash des vorherigen Blocks, eines Zeitstempels, der Software-Version und mehr nötigen Informationen. Eine Transaktion zu fälschen ohne aufzufallen ist praktisch unmöglich, da man gleich alle Transaktionen eines Blockes fälschen müsste. [11]

Git Git speichert allen Inhalt in Baum- oder Blobobjekten. Ein einzelnes Baumobjekt besteht aus ein oder mehreren Einträgen, welche alle Hashs von Blobs oder Subbäumen sind, mit zusätzlich unter anderem Mode, Typ und Dateiname. Es wurden lange SHA-1-Hashs verwendet; da diese aber nicht mehr kryptographisch stark sind, wird mittlerweile SHA-256 verwendet. Der Baum, welcher alle Bäume und Blobs enthält, ist dann ein Merkle-Baum mit nur einem Hash als Wurzel. Wenn man pullt oder pusht überprüft Git, ob die Wurzel des Merkle-Baums verändert ist. Wenn ja, wird immer das linke und rechte Kind überprüft, bis genau die Blätter gefunden sind, die verändert wurden. Dann kann nur das Delta dieser Dateien über das Netzwerk übertragen werden. So schafft Git es, riesige Repositories mit einer sehr langen Historie einfach zu verwalten. [12]

BitTorrent: Merkle hash torrent extension BitTorrent ist ein Peer-to-Peer-Netzwerk zum Austausch von Dateien. [13] Eine auszutauschende Datei wird in kleine Stücke aufgeteilt. Peers schicken sich dann gegenseitig die einzelnen Stücke der Datei. Um die Korrektheit eines erhaltenen Stückes zu verifizieren, nutzt BitTorrent Hashes der Stücke, welche in einer sogenannten Torrent-Datei zu finden sind. Diese Datei wird von jedem Peer, vor dem Beitritt ins Netzwerk, meist von einem WebServer, heruntergeladen. Die Torrent-Dateien können teilweise sehr groß werden, was eine große Last für die WebServer bedeutet. Eine Möglichkeit, um die Torrent-Dateien sehr viel kleiner zu machen, ist die Verwendung von Merkle-Bäumen. Die Hashes der Stücke werden dann als Blätter in einen Merkle-Baum eingefügt. Damit ein vollständiger Baum entsteht, werden so viele „filler hash values“ hinzugefügt, dass

die Blätteranzahl einer Zweierpotenz entspricht. Dann wird der Baum, genauso wie im Paragraph „Erstellen eines Merkle-Baumes“ im Abschnitt 2.3, aufgebaut. Die Torrent-Datei muss jetzt nur noch den entstandenen Wurzelhash, die Dateigröße, sowie die Größe der Stücke enthalten. Um ein erhaltenes Stück zu überprüfen, muss der Client zunächst den Hash des erhaltenen Stückes berechnen. Außerdem benötigt der Client den Wurzelhash aus der Torrent-Datei, sowie den Verifizierungspfad des Stückes, erstellt wie im Paragraphen „Authentifizierung eines Dokuments im Merkle-Baum“ im Abschnitt 2.3, welcher vom Absender mitgesendet werden muss. Schließlich wird, wie im eben genannten Paragraphen beschrieben, aus dem Hash des Stückes sowie dem Verifizierungspfad ein Hash berechnet, welcher mit dem bekannten Wurzelhash verglichen werden kann, um bei Gleichheit die Integrität des Stückes zu bestätigen. [14]

Certificate-Transparency-Projekt CT (Certificate-Transparency) ist ein Teil der Web PKI. Die Web PKI enthält alles Nötige, um Zertifikate, die im Internet für TLS verwendet werden, auszustellen und zu verifizieren. Zertifikate binden einen öffentlichen Schlüssel an einen Namen. Zertifikate werden von Certificate Authorities (CAs) ausgestellt. Man muss diesen CAs vertrauen können, dass sie die Domains an den korrekten Besitzer der Domain binden, also an den korrekten öffentlichen Schlüssel. Jetzt stellt sich aber die Frage, was passiert, wenn man einer CA nicht vertrauen kann. Wenn diese zum Beispiel gehackt wurde oder unsauber gearbeitet hat. Darum braucht man unabhängige, zuverlässige Logs. Diese Logs werden als Merkle-Bäume repräsentiert. Hier sind vor allem die Eigenschaften der öffentlichen Verifizierbarkeit und der Manipulationssicherheit am wichtigsten.[1] [15]

CT verwendet Merkle-Bäume als Datenstruktur für die Logs. Die Blätter eines Baumes sind Hashes von Zertifikaten, welche an das Log angehangen wurden. Wenn der Logserver den Wurzelhash signiert, entsteht ein so genannter „Signed Tree Head“ (STH). Periodisch fügt ein Log alle neuen Zertifikate zu sich hinzu. Es wird ein seperater neuer Merkle-Baum mit den neuen Zertifikaten erstellt und dieser dann mit dem alten Merkle-Baum kombiniert, um einen neuen Merkle-Baum zu erstellen. Mit der Wurzel des neuen Merkle-Baumes wird dann ein neuer STH erstellt. [16]

[1] `https://certificate.transparency.dev/`

TR-ESOR TR-ESOR ist ein Projekt, für beweiserhaltende Langzeitspeicherung von Dokumenten, der Bundesrepublik Deutschland. Es soll versichert werden, dass elektronisch signierte Daten und Dokumente über lange Zeiträume bis zum Ende der Aufbewahrungsfristen im Sinne eines rechtswirksamen Beweiswerterhalts vertrauenswürdig gespeichert werden können. [17] Das wird unter Verwendung von Merkle-Bäumen in Verbindung mit Zeitstempeln erreicht, genauer gesagt mit dem Evidence Record Syntax.

ERS Die Evidence Record Syntax (ERS), definiert in RFC 4998[18], kodiert in ASN.1 oder in XML, erlaubt es, ein oder mehrere Dokumente oder ein oder mehrere Dokumentengruppen parallel aufzubewahren. Dafür wird ein Merkle-Baum angelegt, wo jede Gruppe von Blättern den Hashwerten einer Gruppe von Dokumenten entspricht. Die Wurzel des Hashbaums wird initial durch einen einzelnen Zeitstempel-Token geschützt.
Es gibt zwei Wege den Schutz der Dokumentengruppe zu erneuern:

1. Zeitstempel-Erneuerung: Diese Methode wird eingesetzt, wenn der Hash-Algorithmus zur Erzeugung des Merkle-Baumes immer noch stark ist, aber der kryptografische Algorithmus des Zeitstempel-Tokens schwach werden sollte oder bevor der Zeitstempel nicht mehr erfolgreich validiert werden kann. In diesem Fall wird das gesamte Validierungsmaterial des vorherigen Zeitstempels zur ERS hinzugefügt und ein neuer Zeitstempel generiert, der das Validierungsmaterial und den vorherigen Zeitstempel schützt. Damit ist wieder der ganze Baum geschützt.

2. Hash-Baum-Erneuerung: Diese Methode wird verwendet, bevor der Hash-Algorithmus des neuesten Zeitstempel-Tokens vom neuesten Merkle-Baum schwach wird. In diesem Fall wird der ganze Merkle-Baum mit einem stärkeren Hash-Algorithmus komplett neu berechnet. Dazu gehören die gesamten Original-Dokumente und Dokumentengruppen als auch der vorherige Zeitstempel und das zugehörige Validierungsmaterial. Der neue Merkle-Baum wird mit einem neuen Zeitstempel geschützt. [19]

Die ERS erlaubt es, alle Informationen, welche nötig sind, um die Integrität und Existenz einzelner Dokumente oder Dokumentengruppen seperat von allen anderen Dokumenten oder Gruppen zu kombinieren. Dafür wird ein reduzierter Hash-Baum verwendet. [17]

4 Das elektronische Rezept

Das elektronische Rezept (E-Rezept) ist eine seit Mitte 2021 eingeführte Alternative des klassischen ausgedruckten Papier-Rezepts. Damit können Ärzte Patienten digital Rezepte über die Telematikinfrastruktur und per App bereitstellen. Es wird dabei sichergestellt, dass nur berechtigte Akteure auf personenbezogene Daten vom E-Rezept zugreifen können. [20]

Funktionsweise des E-Rezeptes Ein Arzt kann mithilfe seines Primärsystems, also einem Praxisverwaltungssystem, Apothekenverwaltungssystem, Krankenhausinformationssystem oder Laborinformationssystem und dem elektronischen Heilberufsausweis qualifiziert elektronisch signieren. Das E-Rezept wird dann an den Fachdienst E-Rezept übertragen, wo es bis zu 100 Tage nach Einlösen aufbewahrt wird. Das E-Rezept kann in jeder Apotheke in Deutschland eingelöst werden, indem es digital einer Wunschapotheke zugewiesen wird. Danach wird eine für die Abrechnung mit den Krankenkassen vom Fachdienst signierte Quittung ausgestellt. [21]

Die Signaturen werden in einer speziellen vertrauenswürdigen Ausführungsumgebung (VAU) ausgeführt. Dazu gehört zum Beispiel, dass alle Komponenten des E-Rezept-Fachdienstes vertraulich miteinander kommunizieren und auch der Schutz der E-Rezepte. Das heißt, der E-Rezept-Fachdienst muss sicherstellen, dass E-Rezepte während der Verarbeitung vor dem Zugriff, auch durch den Anbieter, technisch geschützt sind. Dazu werden in der VAU auch HSMs verwendet. [22]

Verwendung von HSMs HSMs werden vom E-Rezept-Fachdienst zur Generierung, Speicherung und Verwendung von Schlüsselmaterial genutzt.

Speichern von Schlüsselmaterial Der Anbieter des E-Rezept-Fachdienstes muss das private Schlüsselmaterial für kryptografische Verfahren (Entschlüsselung, Signaturen) in einem HSM speichern, dessen Eignung durch eine erfolgreiche Evaluierung nachgewiesen wurde. Als Evaluierungsschema kommt der Federal Information Processing Standard (FIPS) in Frage. Die Prüftiefe muss mindestens FIPS 140-2 Level 3 entsprechen. [23]

Erzeugen von Schlüsselmaterial Ein HSM der VAU des E-Rezept-Fachdienstes muss eine Schnittstelle zur Ableitung von symmetrischen Schlüsseln für die Persistierung von E-Rezept-Daten bereitstellen.

Der E-Rezept-Fachdienst hat die folgenden privaten Schlüssel in einem HSM zu erzeugen und anzuwenden:

- TI-Fachdienst-Identität zur Authentisierung des Dienstes gegenüber dem Primärsystem des Leistungserbringers (TLS),
- TI-Fachdienst-Identität für die Ende-zu-Ende verschlüsselte Kommunikation zwischen E-Rezept-VAU und Primärsystem des Leistungserbringers (sicherer Kanal auf Anwendungsebene [gemSpec_Krypt# 7 Kommunikationsprotokoll zwischen E-Rezept-VAU und E-Rezept-Clients]). [23]

5 Proof-of-Concept Implementierung

Um mein Programm in der Praxis zeigen zu können, habe ich mich für zwei Kommandozeilenanwendungen entschieden. Eine für das Signieren und eine für das Verifizieren von den erstellten Signaturen.

5.1 Allgemeine Informationen

Auswahl der Programmiersprache Ich habe mich dazu entschlossen, C zu verwenden, da Performance der Hauptgrund für die Motivation meiner Arbeit ist und ich außerdem mit OpenSSL eine in C geschriebene Kryptobibliothek verwende. Konzeptuell kann man alle Programmiersprachen verwenden, welche eine geeignete Kryptobibliothek besitzen und den Geschwindigkeitsansprüchen genügen.

Verwendete Bibliotheken Ich verwende die C POSIX Standardbibliotheken stdio, string, stdlib, time und dirent, sowie OpenSSL bzw. libcrypto[2] [24] und JSON-C[3] [25]. OpenSSL benutze ich für hashen, base64, ECDSA, dem generieren von zufälligen Bits und ASN.1. JSON-C verwende ich zum Erstellen und Parsen von JSON für die JSON Web Signature.

Verwendete Hashalgorithmen Ich habe SHA-256 verwendet, da es alle von mir geforderten Sicherheitseigenschaften erfüllt und vom BSI als sicher bewertet wird.[6] Das Verfahren kann aber mit beliebigen als sicher eingestuften Hashalgorithmen verwendet werden, die die benötigten Sicherheitseigenschaften erfüllen. In Zukunft kann also leicht ein stärkerer Hashalgorithmus, wie zum Beispiel, SHA-512 oder SHA-3, verwendet werden.

Verwendete eliptische Kurve Für den ECDSA habe ich Brainpool-Kurven verwendet. Eine andere Möglichkeit wäre eine NIST-Kurve. NIST-Kurven sind schneller, aber Brainpool-Kurven verwenden vollständig (pseudo)zufällige Domainparameter (Kurvenparameter), daher ist die Verteidigung bei Brainpool-Kurven gegen Seitenkanalangriffe besser. Brainpool wird vom BSI auch für verschiedene Projekte verwendet und vorzugsweise zu NIST empfohlen. [26] Da die Signatur mit der Kurve, in der Praxis, sowieso im HSM passieren soll, kann dann eine sinnvolle, vom HSM unterstützte Kurve verwendet werden.

[2] `https://www.openssl.org/`
[3] `https://github.com/json-c/json-c`

Verwendeter Zufall Um echten Zufall möglichst gut abzubilden, habe ich in meiner Implementation die OpenSSL-Funktion `RAND_bytes` verwendet. Es handelt sich um einen kryptografisch sicheren Pseudozufallsgenerator, welcher aber eine zuverlässige Entropiequelle des Betriebssystems verwendet, um sich zu seeden.[4] Ich verwende `RAND_bytes` nur ein Mal zum Generieren des Seeds für den Pseudozufall im Programm. [24]

Repräsentation der Signaturen Es gibt viele Möglichkeiten, eine Signatur zu repräsentieren. Wichtige davon sind JSON-Web-Signaturen, ASN.1-Signaturen und XMLDSig-Signaturen. Ich habe mich dafür entschieden, JWS und ASN.1/DER zu unterstützen. Ich nutze diese beiden, weil sie relevant [27] [28] und relativ einfach zu verwenden sind.

5.2 Programm zur Signaturerzeugung

Parameter Als Eingabe gibt man einen Ordner an, dessen enthaltene Dokumente unter einem Merkle-Baum signiert werden. Die Ausgabe der einzelnen Signaturen passiert in einem zweiten angegebenen Ordner. Außerdem benötigt das Programm den privaten Schlüssel des Signierenden, sowie das zugehörige Zertifikat. Das Programm beherrscht sowohl JSON Web Signature als auch ASN.1/DER Signaturen.

-i Pfad zum Ordner, der die zu signierenden Dokumente enthält,

-o Pfad zum Ordner, in dem die Signaturen abgelegt werden,

-k Pfad zum privaten Schlüssel des Signierenden,

-c Pfad zum dem Schlüssel zugehörigen Zertifikat des Signierenden,

-m Auswahl der zu erstellenden Signatur zwischen „jws“ und „asn1“.

Repräsentation des Baumes im Speicher In meinem Programm arbeite ich nur mit vollständigen Merkle-Bäumen. Der Baum ist im Speicher in einem einfachen Array abgelegt. Jeder Eintrag im Array ist der Inhalt eines Knotens im Baum und die Position dieses Knotens im Baum kann man durch den Index im Array ermitteln. Die einzelnen Ebenen des Baumes liegen hintereinander im selben Array, je Ebene von links nach rechts, wie in Abbildung 2 zu sehen. Im gezeigten Array ist der Baum aus Abbildung 1 repräsentiert. Ich habe mich gegen eine abstraktere Darstellung

[4] `https://www.openssl.org/docs/manmaster/man3/RAND_bytes.html`

des Baumes mit Hilfe von Nodes entschieden. Dies hätte aufgrund von schlechterer Cachelokalität die Performance des Programmes verschlechtert und es gibt auch einfache Algorithmen, wenn der Baum in einem Array repräsentiert ist.

Abbildung 2: Repräsentation eines Merkle-Baumes als Array

Erstellung der Dummydokumente und der zufälligen Partnerblätter Die Größe der Blätter im Baum ist für alle Blätter außer der Partnerblätter (und theoretisch Dummydokumenten) durch die Größe der Ausgabe der Hashfunktion festgelegt. In meinem Fall sind das, durch die Verwendung von SHA-256, 32 Byte. Da diese Einschränkung besteht, habe ich mich dazu entschieden, den Partnerblättern die gleiche Größe zu geben, damit alle Blätter die gleiche Größe haben. Das macht manche Algorithmen einfacher, allgemein kann aber eine beliebige Größe gewählt werden, die den Sicherheitsanforderungen genügt. Für die Erstellung der Partnerblätter habe ich zwei Möglichkeiten getestet.

Zufällige Partnerblätter Die erste Möglichkeit, die ich getestet habe, ist sehr intuitiv, nämlich einfach (für SHA-256) 32 Byte zufällig für jedes Blatt zu generieren. Es hat den Vorteil, dass es sicherheitstechnisch keine Bedenken gibt. Der große Nachteil ist natürlich, dass echten Zufall zu generieren sehr aufwendig ist und einen Großteil der Zeit des Kreierens des Baumes ausmachen würde.

Pseudozufällige Partnerblätter Die zweite Möglichkeit ist es, einmalig hinreichend viele Zufallsbits zu generieren und diese dann zu verwenden, um mit dem echt zufällig gewähltem Seed so viele pseudozufällige Blätter pz_b wie nötig zu erstellen. Ich verwende die Position des Blattes im Baum b_n, konkateniert mit dem vorher generiertem Seed s, als Eingabe für die Hashfunktion, also $pz_b = \text{HASH}(b_n|s)$. Der entstehende Hash ist dann das Blatt mit der jeweiligen Blattnummer. Die Größe des Seeds habe ich von der verwendeten Hashfunktion abhängig gemacht. SHA-256 wird in Blöcken von 512 Bits (64 Byte) verarbeitet. Die Position im Baum ist ein Integer, also 32 Bit (4 Byte). Deshalb habe ich als Größe für den Seed 60 Byte = 64 Byte - 4 Byte gewählt. Der Seed ist also möglichst groß, um soviel Sicherheit wie möglich zu erzeugen, aber klein genug, dass SHA-256 nur einen Block verarbeiten

muss und das Programm dadurch schneller wird. Wenn ein größerer Seed benötigt, oder eine andere Hashfunktion verwendet wird, lohnt es sich, je nach Anwendung, auf die Größe der von der Hashfunktion verarbeiteten Blöcke zu achten, um einen optimalen Sicherheits-Performance-Tradeoff zu finden. Außerdem kann dann pro Merkle-Baum viel Speicherplatz eingespart werden (die Hälfte der Blätter), indem nur der Seed gespeichert wird. Die zufälligen Partnerblätter können ja jederzeit aus dem Seed deterministisch berechnet werden. Da die Partnerblätter in den Signaturen sowieso enthalten sind, beinhaltet der Seed keine kritischen, geheimen Informationen. In meinem Programm verwende ich pseudozufällige Partnerblätter.

Dummydokumente Für die Dummydokumente besteht auch die Möglichkeit, diese per komplettem Zufall zu erstellen. Das hat zwei große Nachteile. Einmal kostet es natürlich wieder viel Performance und außerdem lassen sich diese Blätter dann schlecht als Dummydokument identifizieren. Da auch die Dummydokumente ein Partnerblatt bekommen, welches zufällig generiert ist, muss das Blatt des Dummydokumentes gar nicht zufällig sein, um die Integrität des Baumes zu gewährleisten. Der Wert des Dummydokumentes wird ja sowieso mit dem zufälligen Blatt konkateniert und dann gehasht. Da wir von der Einwegeigenschaft der Hashfunktion ausgehen und nur mit dem gehashten Wert weitergerechnet wird, können wir das Dummydokument beliebig wählen. [9] Ich habe mich entschlossen den Hash eines leeren Dokumentes zu verwenden. Diesen kann man schon einmalig vorausberechnen, kostet deswegen nur zur Compile-Zeit Performanz. Außerdem lässt sich so sehr leicht erkennen, welche Blätter von echten Dokumenten sind und welche von Dummydokumenten. Zusätzlich lassen sich die Dummydokumente dann in der Speicherung sparen, da diese schon feststehen und jederzeit eingefügt werden können. In meinem Programm verwende ich den Hash eines leeren Dokumentes als Dummydokumente.

Auswirkung auf die Speicheranforderung Die Speicheranforderung ist natürlich nur relevant, falls ein späteres Aufbauen des kompletten Baumes möglich sein soll. Falls der Baum keine Dummydokumente enthält, kann man den kompletten Baum nur mithilfe der Signaturen wieder aufbauen.

Wenn Dummydokumente verwendet werden, benötigt man diese, sowie die zugehörigen Partnerblätter, da die Dummydokumente keine Signatur erhalten. Wenn Zugriff auf die Signaturen der Dokumente besteht, lassen sich also bei d Dummydokumenten, Blättergröße s und Seedgröße q, $2 * d * s - q$ Byte sparen, wenn der Hash eines leeren Dokumentes als Dummydokumente und geseedeter Zufall für die Partnerblätter verwendet wird.

Wenn man keinen Zugriff mehr auf die Signaturen, aber auf die signierten Dokumente hat, lassen sich bei Verwendung geseedeten Zufalls bei den Random Leaves und dem Hash eines leeren Dokuments als Dummydokument bei n Blättern mit Blättergröße s und Seedgröße q minimum $(n * s)/2 - q$ Bytes sparen, wenn keine Dummydokumente verwendet wurden. Maximal lassen sich $\frac{3}{4}n * s - s - q$ Bytes sparen, wenn man genau $\frac{1}{4}n + 1$ viele Dokumente hat und somit $\frac{1}{4}n - 1$ Dummydokumente, was der maximalen Anzahl entspricht, da die Anzahl der Dokumente + die Anzahl der Dummydokumente immer $\frac{1}{2}n$ entspricht. Für Institutionen, die sehr viele Bäume aufbewahren müssen, lässt sich somit Speicherplatz sparen, wenn die sowieso schnelleren Methoden verwendet werden.

Algorithmus zur Konstruktion des Baumes Vor dem Bauen des Baumes gehen wir davon aus, dass wir schon alle Hashs der Dokumente in einem Array zur Verfügung haben. Aus der Anzahl der Dokumentenhashs können wir uns, mit der Formel Blätteranzahl $= 2^{\lceil \mathbf{log_2}(num\_hashs) \rceil}$, die Anzahl der Blätter berechnen, die der Baum später haben wird. Effizienter geht es noch, indem wir Bitshifts folgendermaßen verwenden:

```
int calculate_least_covering_power_of_two(int number){
    int num = 1;
    while(num < number){
        num = num << 1;
    }
    return num;
}
```

Die gesamte Baumgröße können wir uns dann leicht mit Blätteranzahl $* 2 - 1$ ausrechnen. Wir erzeugen nun ein Array aus Pointern mit der eben berechneten Baumgröße. Die Dokumentenhashs werden abwechselnd mit 32 Byte geseedetem Zufall, angefangen mit den Dokumentenhashs von Index 0, in das Baumarray 5.2 eingefügt. Falls die berechnete Blätteranzahl noch nicht erreicht wurde, erzeugen wir die Dummydokumente, wie im Paragraphen „Dummydokumente“ im Abschnitt 5.2 beschrieben. Diese werden im gleichen Prinzip, anstelle der Dokumentenhashs, eingefügt, bis die Blätteranzahl erreicht ist. Damit befinden sich alle Blätter des Merkle-Baumes im Array. Nun kümmern wir uns um die eigentliche Erstellung des Baumes. Es werden immer zwei benachbarte Knoten konkateniert und dann gehasht, wie im Paragraphen „Erstellen eines Merkle-Baumes“ im Abschnitt 2.3 beschrieben. Für zwei benachbarte Knoten $K1$ und $K2$ wird der berechnete Knoten Kn also mit $Kn = \text{HASH}(K1|K2)$

berechnet. Für einen ganzen Baum ist das zu sehen in Abbildung 1. Der neu entstandene Knoten wird dann als Elternknoten im Baum in das Array eingefügt. Aus Ebene e, angefangen mit 1 bei den Blättern und der Blätteranzahl ba, lässt sich leicht, mit der Formel $2^{(log_2(2ba)-e)} * (2^e - 2)$, die Position der Ebene e im Baumarray berechnen. Für $e + 1$ erhält man dann die Ebene der Elternknoten. Wenn man die Kinder von links nach rechts durchgeht, kann man auch die Eltern in der neuen Ebene leicht von links nach rechts einfügen. Das wird so lange gemacht, bis wir den Wurzelknoten des Baumes berechnet und somit den ganzen Baum zur Verfügung haben. Die eben genannte Prozedur kann entweder rekursiv erfolgen, oder in einer verschachtelten Schleife. Ich habe mich für die Variante mit den Schleifen entschieden, weil es so für mich leichter nachzuvollziehen war, dass alle Knoten wirklich in den Grenzen des Arrays bleiben.

```
for(int j = 1; j < log_base_2(final_leave_list_size*2); j++) {
    int current_step =
        power_of_two(log_base_2(final_leave_list_size*2)-j)*(power_of_two(j)-2);
    int next_step =
        power_of_two(log_base_2(final_leave_list_size*2)-(j+1))*(power_of_two(j+1)-2);
    int l=0;
    for(int k = current_step; k < next_step; k = k + 2) {
        hash_two_nodes(merkle_tree[k], merkle_tree[k + 1],
            merkle_tree[next_step+l]);
        l++;
    }
}
```

Der Codesnippet zeigt meinen Algorithmus zum Bauen des Baumes. Man sieht das Anwenden der eben gezeigten Formel im Code, zusammen mit dem Hashen der beiden Kinder und dem Einfügen des Elternknotens. Die äußere for-Schleife beschreibt, dass das alles für jede Ebene, bis zu der Ebene unter dem Wurzelknoten gemacht wird.

Algorithmus zur Bestimmung des Verifizierungspfades Der Verifizierungspfad ist, wie schon im Paragraphen „Authentifizierung eines Dokuments im Merkle-Baum“ im Abschnitt 2.3 erwähnt, nötig, um die Verifikation der Signatur möglich zu machen. Das Ziel ist es, alle Knoten zu finden, die nötig sind, um von nur dem Blatt, welches das zu verifizierende Dokument repräsentiert, über die Methode zur Konstruktion des Baumes die Wurzel berechnen zu können. Das heißt, man benötigt das Partnerblatt, sowie alle Knoten, die den in jedem Schritt berechneten Knoten gegenüberliegen, in Abbildung 3 grün und gestrichelt dargestellt.

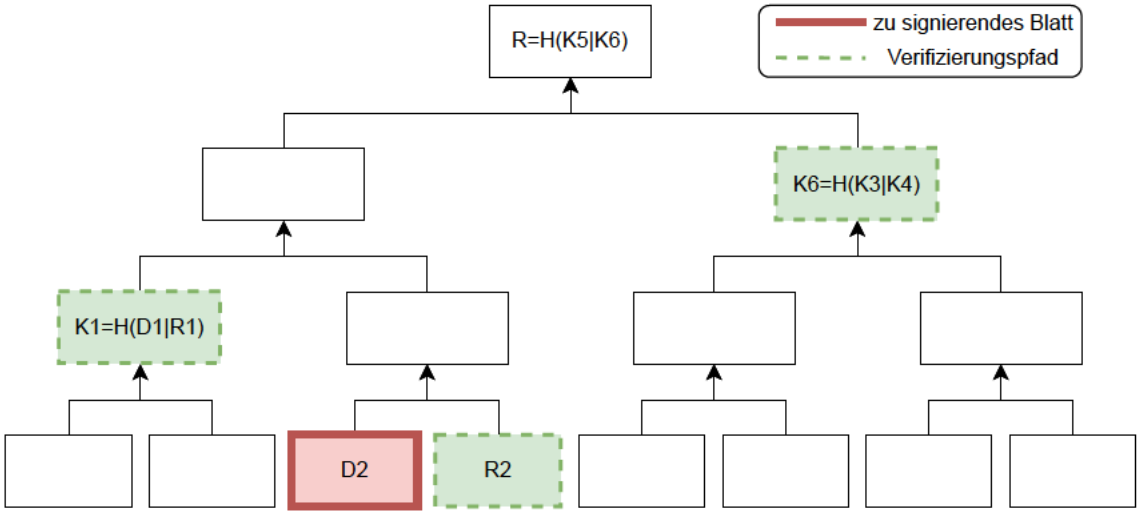

Abbildung 3: Verifizierungspfad der Datei D2

Der Algorithmus muss für beliebige positive Baumgrößen und für alle Blätter mit Dokumentenhashs funktionieren. Da jedes zweite Blatt ein zufälliges Partnerblatt ist, muss der Algorithmus für diese nicht unbedingt funktionieren, es schadet aber natürlich nicht. Je nach verwendeter Datenstruktur zur Repräsentation des Baumes kann es unterschiedliche Algorithmen geben, um den Verifizierungspfad für ein Dokument, welches ja als Hash als Blatt im Baum repräsentiert ist, zu finden. Die von mir gewählte Datenstruktur ist ein Array, wie schon inm Paragraphen „Repräsentation des Baumes im Speicher“ im Abschnitt 5.2 beschrieben. Für den Algorithmus gehe ich davon aus, dass die Nummerierung des Array bei 1 beginnt und die Ebene mit dem Wurzelknoten Ebene 1 ist. Außerdem betrachte ich jede Ebene einzeln. Wenn ein Knoten eine ungerade Position besitzt, wird die Position+1 für den Algorithmus verwendet, weil dieser sich für beide Blätter gleich verhält. Außerdem wird dem zum Verifizierungspfad hinzugefügten Knoten ein „-“ vorangeschrieben, wenn dieser sich

in seiner Ebene auf einer ungeraden Position befindet. Das wird gemacht, damit der Algorithmus zum Verifizieren der Signatur, beschrieben im Paragraphen „Verifikation der JSON Web Signature“ im Abschnitt 5.3, unterscheiden kann, wie die Knoten konkateniert werden müssen. Da die Knoten Hashes sind und so jedes beliebige Zeichen beinhalten können, base64-kodiere ich diese. So können die Knoten keine problematischen Zeichen, wie Zeilenumbrüche, enthalten. Base64 enthält außerdem kein „-“.

Algorithmus

1. Merken der Position der aktuellen Ebene e und der eins höheren Ebene n_e im Baum
2. Hinzufügen des Partnerblattes zum Verifizierungspfad
3. Wenn die aktuelle Ebene $e < 3 \rightarrow$ fertig
4. Überprüfe, ob die Position des aktuellen Knotens p gerade oder ungerade ist, wenn ungerade $\rightarrow p = p + 1$
5. Berechne $m = p \bmod 4$.
6. Wenn $m == 0 \rightarrow p = n_e + (p - \text{lastposition})/2 - 1$,
   sonst $\rightarrow p = n_e + (p - lastposition)/2 + 1$
7. Hinzufügen des Knotens an der neu berechneten Position zum Verifizierungspfad, wenn $m == 0$, dann füge dem Knoten von links ein „-“ hinzu
8. Aktualisieren der Positionen der Ebenen auf jeweils eine Ebene höher im Baum
9. gehe zu 3.

Aus dem entstehenden Verifizierungspfad kann man die ursprüngliche Position des zugehörigen Dokumentes im Merkle-Baum berechnen. Daraus lassen sich potentiell Informationen ableiten. Zum Beispiel, welche Dokumente zu einer Person gehören, wenn man schon die Anzahl und die Position mindestens eines Dokumentes kennt. Wenn man das verhindern möchte, muss man die Dokumentenhashes in einer zufälligen Reihenfolge den Blättern hinzufügen, oder sogar auf verschiedene Merkle-Bäume verteilen. Ich habe die Reihenfolge nicht randomisiert, da dies nicht nötig ist, um meine Schutzziele zu erreichen.

Aufbau der Signaturen Beide vorgestellten Signaturvarianten erhalten ein X.509-Zertifikat des Signierenden, mit welchem die Authentizität gewährleistet wird. Außerdem enthalten sie den Verifizierungspfad, sowie die ECDSA-Signatur, was beides benötigt wird, um die Integrität der Daten und Nichtabstreitbarkeit für den Signierenden zu gewährleisten.

Aufbau der erstellten JSON Web Signature Die JSON Web Signature hat einen festgelegten Aufbau: `Header.Payload.Signature`. Der `Header` muss immer den verwendeten Algorithmus angeben. Den Namen des verwendeten Algorithmus mit SHA-256 als Hashalgorithmus schlage ich als „HTES256 “ vor. Außerdem kommt das X.509-Zertifikat des Signierenden im `Header`, angegeben mit dem Tag „x5c“, unserer JSON Web Signature vor. Dafür wird das Zertifikat base64-kodiert und dann dem `Header` hinzugefügt. In `Payload` kann entweder das gesamte Dokument, oder der Hash des zu signierenden Dokumentes abgelegt werden. Ich habe zu Demonstrationszwecken das gesamte Dokument abgelegt. Das macht einen Unterschied für die Verifikation der Signatur, da dafür das gehashte Dokument benötigt wird. `Signature` enthält den Verifikationspfad der base64-kodierten Hashes und die base64kodierte ECDSA-Signatur. `Header`, `Payload` und `Signature` müssen jeweils base64url-kodiert werden. [29]

Der `Header`-Teil sieht folgendermaßen aus:

```
{"alg":"HTES256",
"x5c": base64(Zertifikat des Signierenden)}
```

Der `Signature`-Teil sieht wie folgt aus:

```
{"ht_path": Verifizierungspfad}
{"ecdsa_sig": base64(ECDSA-Signatur)}
```

Eine von mir, mit dem Programm, erstellte JWS, mit base64url-dekodiertem `Header`, `Payload` und `Signature`, ist im Anhang 8 zu finden.

Aufbau der erstellten ASN.1/DER Signatur Für den Aufbau der ASN.1 Signatur habe ich mich am X.509 Zertifikat orientiert. [30]

Der Aufbau ist wie folgt:

```
Signatur ::= SEQUENCE {
    ecdsaSig        ecdsa-with-SHA256, (OID 1.2.840.10045.4.3.2)
    HashSequence    SEQUENCE OF UTF8STRING,
    Certificate     X509V3SignedCertificate
}
```

`ecdsaSig` enthält die ECDSA-Signatur und die Information, mit welchem Hash-Algorithmus signiert wurde. In meinem Fall SHA-256. `HashSequence` enthält den Verifizierungspfad als Sequence von UTF8-Strings. Ich habe UTF8-Strings verwendet, es ist aber auch möglich andere Stringarten, wie OCTET-Strings zu verwenden. `Certificate` enthält das Zertifikat des Signierenden. In meinem Fall ein X.509-Zertifikat. Dieser ASN.1-Signatur kann man eine neue OID geben, aus welcher schon erkannt werden soll, welchen Hash-Algorithmus man benötigt (hier SHA-256). Eine von meiner Implementation erstellte Signatur findet man im Anhang 8.

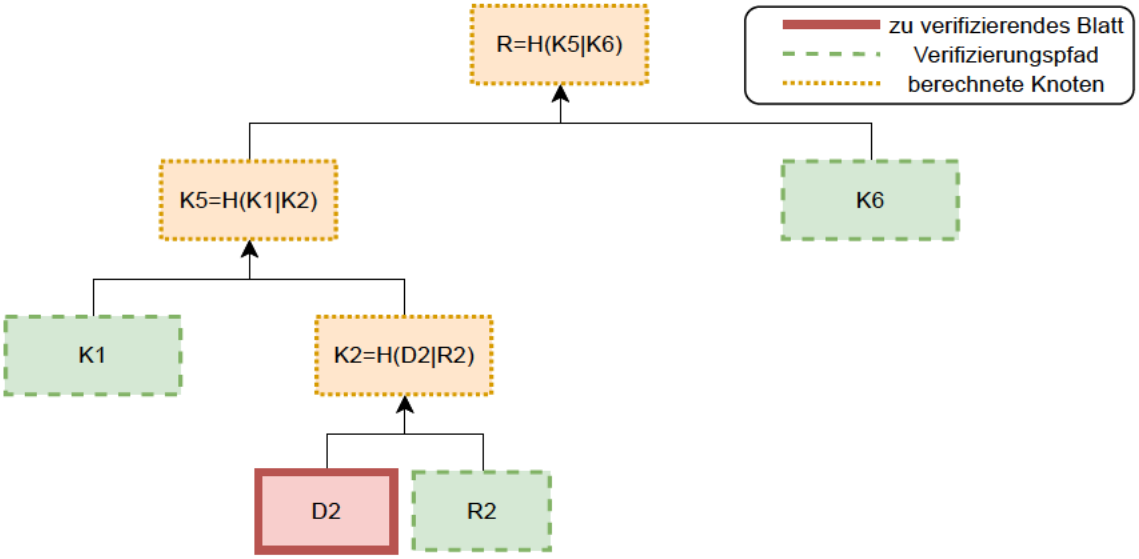

Abbildung 4: Verifizierung einer Signatur des Dokumentes D2

5.3 Programm zur Signaturverifikation

Parameter Als Eingabe benötigt das Programm lediglich die Signatur, welche verifiziert werden soll und den Modus, um welche Signatur es sich handelt. Da das Programm zum jetzigen Stand nur JWS unterstützt, muss der Modus nicht angegeben werden.

-i Pfad zur Signatur,

-m Auswahl der Repräsentation der Signatur

Verifikation der JSON Web Signature Zum Verifizieren der JSON Web Signature muss die Signatur in ihre Bestandteile `Header`, `Payload` und `Signature` aufgespalten werden. Alle drei Teile sind base64url-kodiert und werden dekodiert, bevor der Algorithmus mit diesen weiterarbeitet. Der `Header` besteht aus zwei Teilen:
Dem „alg“- und „x5c“-Teil. Die Verifikation überprüft erst, ob in „alg“ „HTES256“ steht. Wenn nicht, dann wurde die Signatur nicht mit dem korrekten Verfahren kodiert und kann nicht mit diesem Algorithmus verifiziert werden. In „x5c“ befindet sich das base64-kodierte Zertifikat des Signierenden, welches dekodiert wird und danach der öffentliche Schlüssel aus dem Zertifikat entnommen wird. Die restlichen Informationen aus dem Zertifikat sind für den reinen Algorithmus nicht notwendig, müssen aber natürlich überprüft werden, um die Authentizität des öffentlichen Verifikationsschlüssels zu überprüfen. Im `Payload` liegt, je nachdem, das Dokument, oder der Hash des Dokumentes. In meinem Fall das komplette Dokument. Im `Signature`-Teil befindet sich das Array mit den Knoten vom Verifizierungspfad und die ECDSA-Signatur, zu sehen als grün gestrichelte Knoten in Abbildung 4.

Bei der Verifikation startet man mit dem Hash des Dokumentes als Ergebnishash, welches man aus dem Payload bekommt. In Abbildung 4 ist das der rote, dick umrandete Knoten. Wenn es sich um das ungehashte Dokument handelt, muss das natürlich noch gehasht werden. Dann nimmt man immer, von links nach rechts, einen Knoten aus dem Array des Verifizierungspfades und guckt, ob dieser an erster Stelle ein „-“ hat, oder nicht. Wenn sich dort ein „-“ befindet, heißt das, dass der neue Hash an den aktuellen Ergebnishash von links konkateniert wird. Wenn kein „-“ existiert, wird der Hash von rechts konkateniert. Der entstehende String wird dann wieder gehasht und es entsteht ein neuer Ergebnishash. Das wiederholt man für alle Knoten aus dem Verifizierungspfad und erhält schließlich die Wurzel des Merkle-Baumes als finalen Ergebnishash. Alle berechneten Ergebnishashes in einem Beispiel mit Verifizierungspfadlänge drei sind als orange, gepunktete Knoten in Abbildung 4

zu sehen. Mit Hilfe der Wurzel und dem öffentlichen Schlüssel aus dem Zertifikat kann man dann die ECDSA-Signatur überprüfen. Ich mache das mit der OpenSSL-Funktion `ECDSA_do_verify`. Wenn die Verifikation der ECDSA-Signatur erfolgreich ist, dann ist auch die Verifikation der gesamten Signatur abgeschlossen und erfolgreich. Ein Algorithmus könnte folgendermaßen aussehen:

Algorithmus zur Verifikation der JSON Web Signature

1. Header, Payload und Signaturkomponente aus JWS extrahieren
2. Alle Komponenten base64url-dekodieren
3. „alg“-Feld im Header überprüfen
4. Zertifikat aus dem „x5c“-Feld im Header base64-dekodieren
5. Öffentlichen Schlüssel *pubkey* aus dem Zertifikat extrahieren
6. Zertifikat bis zum Vertrauensanker überprüfen
7. Verifizierungspfad aus der Signaturkomponente extrahieren und base64-dekodieren
8. Setze $Ergebnisknoten$ auf den Hash des Dokumentes aus dem Payload
9. Für jeden Knoten im Verifizierungspfad:
   a) base64-dekodiere den Knoten als $Aktueller - Knoten$
   b) Wenn $Aktueller - Knoten$ mit einem „-“ beginnt $\rightarrow$ $Ergebnisknoten = \text{HASH}(Aktueller - Knoten | Ergebnisknoten)$
   c) sonst $\rightarrow$ $Ergebnisknoten = \text{HASH}(Ergebnisknoten | Aktueller - Knoten)$
10. $Wurzelknoten = Ergebnisknoten$
11. Extrahiere die ECDSA-Signatur aus der Signaturkomponente und base64-dekodiere diese
12. Ganz normale Verifikation der ECDSA-Signatur mit *pubkey* und $Wurzelknoten$ durchführen

Nummer 6 im Algorithmus, also das Überprüfen des Zertifikats bis zum Vertrauensanker, habe ich in meiner Implementation nicht umgesetzt. Nummer 9 im Algorithmus beschreibt das Berechnen der Wurzel aus dem Dokumentenhash und dem Verifizierungspfad.

Mein geschriebener Code dazu sieht folgendermaßen aus:

```
unsigned char *current_result_node = message_hash;
unsigned char *current_new_node;
unsigned char *root;
for (int i = 0; i < ht_path_length; i++)
{
    json_object *current_node_json = json_object_array_get_idx(ht_path_json,
        i);
    const char *current_node_encoded =
        json_object_get_string(current_node_json);
    if(current_node_encoded[0] == '-'){
        current_new_node = base64_decode(current_node_encoded+1,
        (int) strlen(current_node_encoded)-1);
        hash_two_nodes(current_new_node, current_result_node,
            current_result_node);
    }
    else{
        current_new_node = base64_decode(current_node_encoded,
        (int) strlen(current_node_encoded));
        hash_two_nodes(current_result_node, current_new_node,
            current_result_node);
    }
    free(current_new_node);
}
root = current_result_node;
```

6 Benchmarking

Benchmarking ist nötig, um konkurrierende Methoden innerhalb des Programms und die Geschwindigkeit des Programms im Vergleich mit der Signatur einzelner Dokumente testen zu können. Da es bei der Motivation dieser Arbeit nicht nur um die Geschwindigkeit der Algorithmen, sondern auch darum geht, die Kosten niedrig zu halten, wurden die Benchmarks auf einem Computer mit folgenden relevanten Bauteilen durchgeführt: Intel(R) Core(TM) i7-6700K und CORSAIR Vengeance LPX 16GB (2 x 8GB) 288-Pin PC RAM DDR4 2133 (PC4 17000) Desktop Memory Model CMK16GX4M2A2133C13. Verzichtet wurde deshalb auf Benchmarks auf großen, leistungsstarken Servern, wie denen der Humboldt-Univerität zu Berlin. Die Benchmarks wurden auf Linux Mint 20.3 durchgeführt. Da der Input über Dokumente aus einem Ordner funktioniert, habe ich einen Ordner mit 3.000.000 zufälligen 32-Byte großen Dokumenten generiert und im Programm eine Obergrenze `MAX_DOCUMENTS` eingeführt, mit der ich einstellen kann, wie viele Dokumente verwendet werden. Ich habe mich für diese Anzahl an Dokumenten entschieden, weil es die größtmögliche Menge war, für die ich noch aufgrund von limitiertem Arbeitsspeicher und der Nutzung von einer Ramdisk vernünftige Benchmarks durchführen konnte. Außerdem bestimmt die Variable `NUMBER_OF_RUNS`, wie häufig die zu testende Methode laufen soll. So lässt sich leicht eine Test-Menge für eine bestimmte Dokumentenanzahl definieren. Zur Zeitmessung verwende ich `clock()` aus `<time.h>`, welches laut POSIX eine gut geeignete Abschätzung der genutzten CPU-Zeit des Programmes angeben soll. [31] Das ist wichtig, um eine Messung unabhängig der Arbeitslast des verwendeten Computers zu ermöglichen. Eine Abhängigkeit würde entstehen, wenn die Messung Echtzeit anstatt verwendete CPU-Zeit verwenden würde. Für alle Benchmarks, bei denen Dateien eingelesen werden, habe ich eine Ramdisk verwendet, damit der Zugriff auf das Dateisystem einen möglichst kleinen Einfluss auf den Benchmark hat. Ich habe die Ramdisk mit dem Befehl

```
sudo mount -t tmpfs -o size=12G myramdisk ~/ramdisk,
```

für eine Ramdisk der Größe 12 GB erstellt. So liegen die Dateien schon bei Start des Benchmarks im Arbeitsspeicher und müssen nicht erst von der Festplatte in den Arbeitsspeicher kopiert werden. [32] Ein möglichst kleiner Einfluss des Dateisystems ist besonders wichtig, da die zu signierenden Dateien in der Praxis von komplett unterschiedlichen Quellen in den Arbeitsspeicher kommen können, wie z.B. HDDs, SSDs, dem Netzwerk oder anderen und die Benchmarks möglichst allgemein gültig

sein sollen. In allen folgenden Benchmarks ist das Ziel, dass die Tests möglich schnell sind, also die durchschnittliche Zeit möglichst klein. Um zu zeigen, dass die Messungen aussagekräftig sind, habe ich für jede Messung, zusätzlich zu der durchschnittlichen Zeit, auch die Varianz berechnet. Diese ist im Anhang in den Tabellen 1 bis 10 immer mit eingetragen. Bei manchen Messungen war die Auflösung von `clock()` nicht groß genug, weswegen für die Varianz nur eine 0 steht. Für diese Messungen kann ich nicht gewährleisten, dass sie aussagekräftig sind. Die Werte waren für mich aber immer plausibel.

6.1 Echter Zufall versus Pseudozufall bei der Generierung der Partnerblätter

Wie schon in den Paragraphen „Zufällige Partnerblätter“ und „Pseudozufällige Partnerblätter“ im Abschnitt 5.2 beschrieben, kann man die Partnerblätter entweder mit echtem Zufall oder per Pseudozufallsgenerator erstellen. Wenn man entscheidet, welche von beiden Methoden man verwenden möchte, sollte auch der Zeitunterschied in Betracht gezogen werden, da eine höhere Geschwindigkeit die Motivation für das Verwenden von geseedetem Zufall ist. Der Benchmark enthält auch das Lesen und Hashen der Dokumente aus dem Arbeitsspeicher. Ich sehe einen großen Zusammenhang zwischen diesen Schritten und dem Hinzufügen der Partnerblätter, da genau ein Partnerblatt pro eingelesenem und gehashtem Dokument generiert wird. So kann man gut sehen, was für einen Unterschied das für den gesamten Schritt des Hinzufügens von Dokumenten zum Baum macht. Die Werte zu den Benchmarks befinden sich im Anhang in den Tabellen 1 und 2.

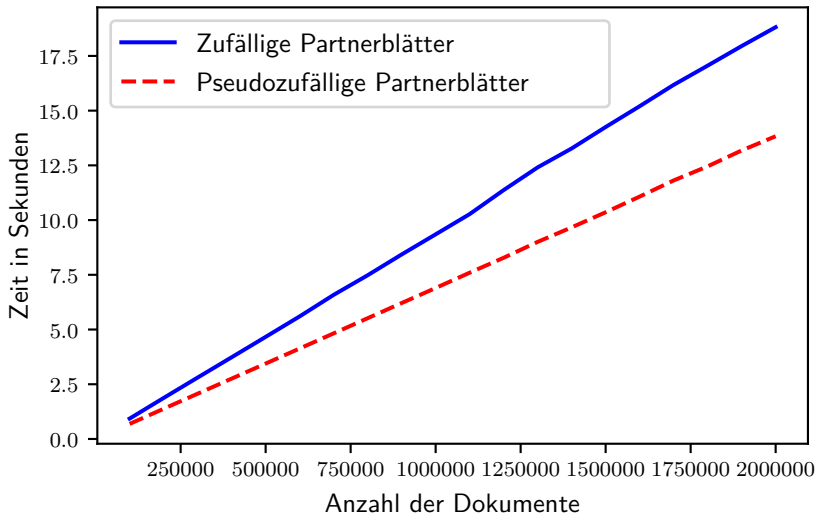

Abbildung 5: Benchmark Partnerblätter

Wie man in Abbildung 5 sehen kann, ist der Abstand zwischen den beiden Methoden größer, je größer die Anzahl der Dokumente ist. Wie zu erwarten ist die Methode mit geseedetem Zufall schneller. Der größte absolute Abstand befindet sich bei 2.000.000 Dokumenten mit einer Zeit von 13,83 s für die pseudozufälligen und 18,81 s für die zufälligen Partnerblätter. Das macht einen absoluten Abstand von 4,98 s und einen relativen Abstand von etwa 23,78 %. Bei einer noch größeren Anzahl von Dokumenten wird der absolute Abstand, also der Zeitgewinn für das Nutzen von pseudozufälligen Partnerblättern, vorraussichtlich noch größer sein. Da der Abstand unabhängig der Anzahl von Dokumenten pro Baum nur um 25 % liegt, kann es sich je nach Anwendung lohnen, echten Zufall zu verwenden. Bei Verwendung eines sicheren Pseudozufallsgenerators sollte aber der Pseudozufall aufgrund der Zeitersparnis vorgezogen werden.

6.2 Zufällige versus fest gesetzte Dummydokumente

Wie Dummydokumente erstellt werden, kann einen großen Unterschied auf die Performance des Programmes haben, da sie bei n Blättern bis zu $\frac{n}{4} - 1$ Blätter ausmachen können, die dem Baum hinzugefügt werden. Der Benchmark zeigt das Bauen des Baumes, bei dem im ersten Schritt die Dummydokumente hinzugefügt werden. Die Werte zu den Benchmarks befinden sich im Anhang in den Tabellen 3, 4, 5 und 6. Abbildung 6 zeigt, dass der feste Hash, wie zu erwarten, in allen Fällen schneller ist. In Abbildung 7 ist gut zu sehen, dass die beiden Werte immer annähernd gleich sind, wenn die Anzahl der Dokumente eine Zweierpotenz erreicht. Das ist der Fall, da immer so viele Dummydokumente hinzugefügt werden, bis die Blätteranzahl eine Zweierpotenz erreicht (siehe Ërstellen eines Merkle-Baumesïm Abschnitt 2.3). Nehmen wir an, $n = 2^m$ und $m \in \mathbb{N}$: Bei n Dokumenten gibt es natürlich genau $2n$ oder 2^{m+1} Blätter, da jedes gehashte Dokument ein Partnerblatt erhält. Dann müssen keine Dummydokumente generiert werden. Bei $n + 1$ Dokumenten müssen immer am meisten Dummydokumente generiert werden, nämlich $n - 1$ viele, weshalb man beim nächstliegenden Wert im Graphen die lokalen Maxima sieht. Dadurch entsteht ein Sägezahnmuster für beide Methoden, welches aber deutlich ausgeprägter für die zufälligen Dummydokumente ist. In Abbildung 6 sieht man, dass das auch für eine kleinere Dokumentenzahlen signifikant ist. Wenn nicht auf die Anzahl der Dokumente geachtet wird, bevor ein Baum aufgebaut wird, empfehle ich deshalb, immer festgesetzte Dummydokumente zu verwenden. Bei einer ungünstigen Anzahl von Dokumenten kann nämlich ein großer Zeitverlust entstehen. Wenn man zufällige Dummydokumente verwendet, sollte man die Anzahl der Dokumente immer überprüfen, bevor man einen Baum aufbaut und möglichst genau mit n, oder möglichst nah vor n starten. Wir können uns durch die Einwegeigenschaft der Hashfunktion und den pseudozufälligen Partnerblättern darauf verlassen, dass auch ein pseudozufälliger Elternknoten, des Dummydokumentes mit seinem Partnerblatt, entsteht. Dadurch sehe ich den Geschwindigkeitsvorteil groß genug und verwende den Hash eines leeren Dokumentes als Dummydokumente.

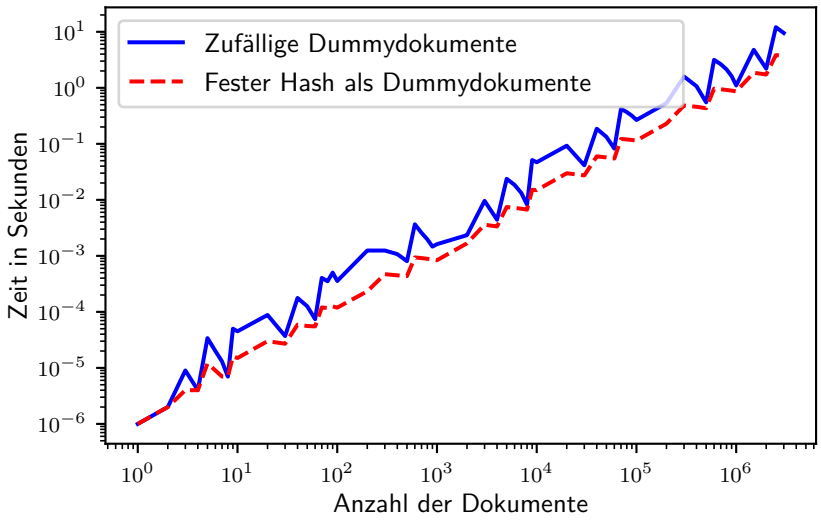

Abbildung 6: Benchmark Bauen des Baumes

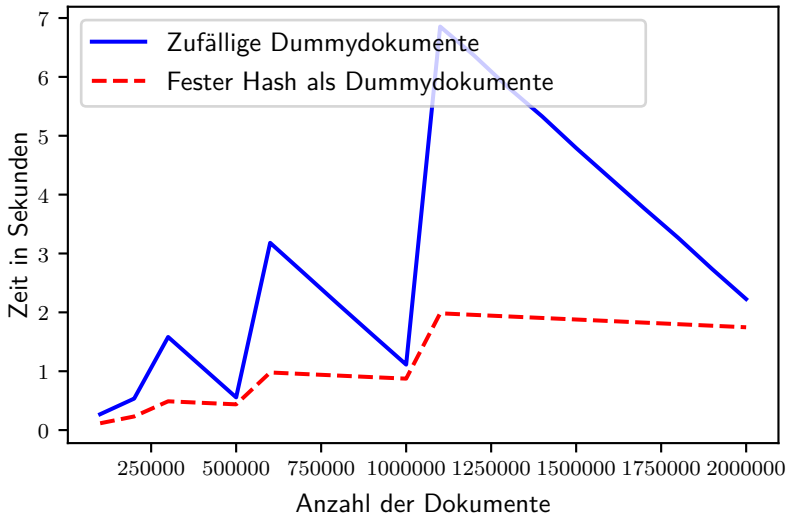

Abbildung 7: Benchmark Bauen des Baumes

6.3 Implementation versus einzelne Signaturen

Ein Hauptziel der Arbeit war es, eine schnellere und dennoch vergleichbar sichere Alternative zum einfachen Signieren jedes Dokumentes zu finden. Im Folgenden vergleiche ich die Geschwindigkeit eines ausgewähltem HSMs mit der meiner Implementation. Ich habe das Programm sowohl mit der Erstellung von JSON Web Signaturen, als auch ASN.1-Signaturen getestet. Die Werte zum Benchmark mit JWS befinden sich im Anhang in der Tabelle 7, mit ASN.1 in der Tabelle 8.

Vergleichsobjekt HSM Als Vergleichsobjekte habe ich mir die HSMs von utimaco entschieden. Der Wert des HSMs, der uns interessiert, ist die 256 Bit Elliptic Curve signature generation. Laut dem von mir verwendeten Datasheet, hat der bestmögliche HSM „Se1000“ eine Rate von 680 Transaktionen pro Sekunde (tps).[5] [2] Das sind in diesem Fall 680 signierte Dokumente pro Sekunde. Der genannte HSM beherrscht ECDSA mit NIST und Brainpool Kurven, bei den Performancedaten steht aber leider nicht, für welche Kurve die 680 tps stehen. Darum rechne ich $\frac{1}{680}$ s bei der durchschnittlichen Programmlaufzeit obendrauf, da die Wurzel der Merkle-Bäume sowieso jeweils mit einem HSM signiert werden müsste, was ich nicht vernachlässigen möchte. Da ich die ECDSA-Signatur in meinem Programm mitberechne, braucht es in den Benchmarks minimal länger als in der Praxis, wenn die Signatur nur im HSM erzeugt würde. Zusätzlicher Zeitaufwand meiner Implementation ist kein Problem, da ich mit diesem Benchmark versuche zu zeigen, dass das Programm deutlich schneller ist, als einzelne Signaturen mit HSMs.

[5] https://hsmsecurity.pl/wp-content/uploads/2016/08/DataSheet-Seria-Se.pdf

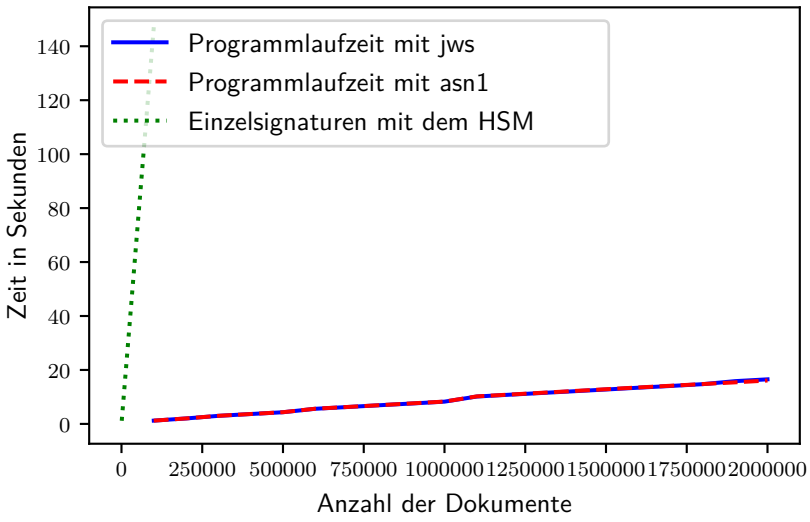

Abbildung 8: Benchmark des kompletten Programmes

Wie man in Abbildung 8 deutlich erkennen kann, ist das Erstellen der Signaturen mit Hilfe des Merkle-Baumes um ein Vielfaches schneller, als jedes Dokument einzeln zu signieren. So braucht man mit Hilfe des Merkle-Baumes für 100.000 Dokumente, einschließlich der Signatur im HSM, nur 1,213622 s, während die Erzeugung von 100.000 einzelnen Signaturen im HSM 147,06 s benötigen würde. Damit ist mein Programm etwa 121 mal so schnell. Das zeigt, dass meine Implementation eine deutlich schnellere Alternative ist. Ob die Signatur als JWS oder ASN.1/DER-Signatur repräsentiert wird, macht, was die gesamte Programmlaufzeit angeht, keinen signifikanten Unterschied.

6.4 Zeitaufwand bei der Verifikation

Bei einer Signatur ist natürlich nicht nur der Zeitaufwand für das eigentliche Signieren, sondern auch für die Verifikation wichtig. Je nach Anwendungsfall muss diese ja auch mehrfach für eine Signatur ausgeführt werden. Da die vorgeschlagene Signatur ausschließlich Zusatzinformationen enthält und bei der Verifikation mehr Schritte notwendig sind, ist die Verifikation natürlich langsamer als bei einer einfachen Signatur ohne Verifizierungspfad. Die Benchmarks sind für die Verifikation verschiedener JWSs mit unterschiedlich langem Verifizierungspfad. Einmal messe ich, wie lange das Berechnen der Wurzel aus dem Verifizierungspfad dauert und einmal die gesamte Verifikation. Die Werte zu den Benchmarks befinden sich im Anhang in den Tabellen 9 und 10. Die Messung in Abbildung 9 zeigt das Berechnen der Wurzel aus dem Verifizierungspfad. Zu sehen ist, dass dieser Schritt der Verifikation etwas 0,001 ms pro Hash im Verifizierungspfad dauert. Bei einer Länge des Verifizierungspfades von 22, was einer Dokumentenanzahl, bei der Signatur, von bis zu $2^{22} - 1 = 4.194.303$ entspricht, sind es 0,024 ms. Etwa 40.000 Verifikationen brauchen also, in diesem Fall, eine Sekunde mehr, nur für den Schritt des Berechnens der Wurzel. In Abbildung 10 sieht man, dass die Länge des Verifizierungspfades noch einen etwas größeren Einfluss auf die gesamte Verifikation hat. Bei der Verifizierungspfadlänge 1 habe ich 0,142 ms gemessen, bei der Länge 22 0,193 ms. Das macht eine Differenz von 0,051 ms, also etwa doppelt so viel, wie nur für das Berechnen der Wurzel. Für die gesamte Verifikation macht die Pfadlänge hier einen Unterschied von etwa 0,002 ms pro Knoten aus. Den niedrigeren Wert bei Verifizierungspfadlänge 7, kann ich nur durch eine Messungenauigkeit erklären, da der Algorithmus bei einem längeren Pfad auch immer länger dauern müsste. Wenn die Signatur nicht sehr häufig verifiziert werden muss, sehe ich diese Zeit nicht als ein Problem an.

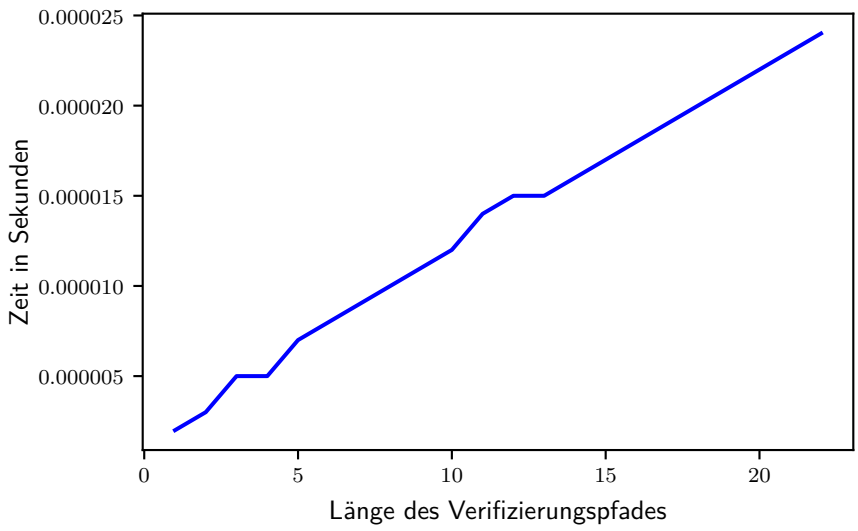

Abbildung 9: Benchmark Berechnung der Wurzel aus dem Verifizierungspfad

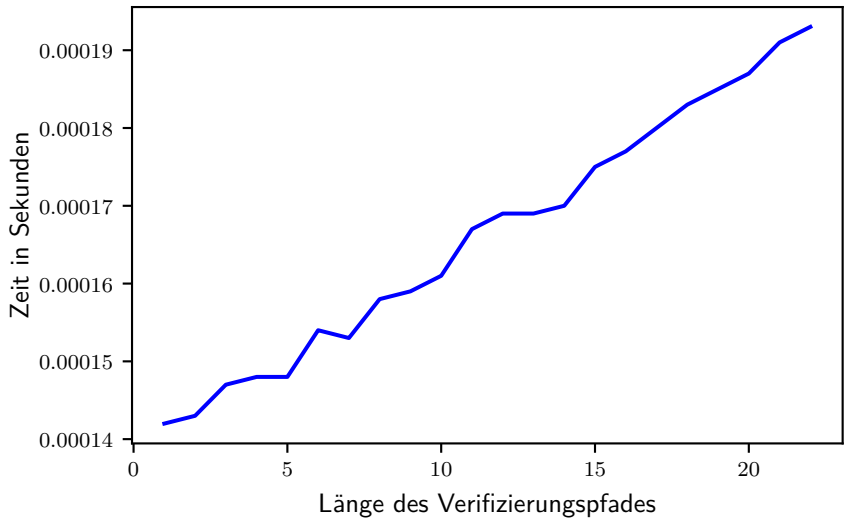

Abbildung 10: Benchmark der kompletten Verifikation

7 Zusammenfassung und Ausblick

Die Arbeit hat gezeigt, dass eine Verwendung von Merkle-Bäumen sinnvoll ist, um viele Dokumente in einer kurzen Zeit zu signieren. Die Datenstruktur Merkle-Baum ist bereits sehr erprobt und wird in den verschiedensten Bereichen erfolgreich eingesetzt. Auch für den Anwendungsfall E-Rezept dieser Arbeit, zeigen die Benchmarks eindeutig, dass die Verwendung, gerade durch den benötigten Einsatz von HSMs, einen Geschwindigkeitsvorteil um einen Faktor von 100 und mehr bringt. Außerdem bleiben alle von mir gewünschten Sicherheitseigenschaften erhalten. Nachteile sind, dass die einzelnen Signaturen mehr Speicherplatz verbrauchen und die Verifikation etwas länger dauert, was aber nicht signifikant passiert und gerade für den Anwendungsfall E-Rezept kein Problem ist, da die Zertifikate nicht oft verifiziert werden müssen.

Es gibt einige wichtige Aspekte, die in dieser Arbeit nicht betrachtet wurden und für zukünftige Arbeiten interessant sein könnten. Mit dem Aufbauen des Merkle-Baumes werden Teile der Signatur aus der sicheren Umgebung des HSMs ausgelagert. Eine Nutzung von sicheren Umgebungen, wie zum Beispiel Intel SGX [33], könnte sinnvoll sein. Dabei ist auch die Auswirkung auf die Performanz interessant. Für eine Dokumentensignierung wie im E-Rezept, wäre außerdem eine praktische Implementierung mit Nutzung eines echten HSMs, ein Schritt in eine reale Verwendung dieser Technologie.

8 Anhang

Beispiel JSON WEB Signature


```
{"alg":"HTES256","x5c":"MIICPzCCAeWgAwIBAgIUbvojgcSuEMVE70MnfBtCoX+HSH8wCgYIKoZIzj0
    EAwIwdTELMAkGA1UEBhMCREUxDzANBgNVBAgMBkJlcmxpbjEPMA0GA1UEBwwGQmVybGluMRAwDgYDVQ
    QKDAdnZW1hdGlrMRAwDgYDVQQLDAdnZW1hdGlrMSAwHgYDVQQDDBdIYXNodHJlZS1TaWduZXItRXhhb
    XBsZTAeFw0yMTA0MjAxNTE5MzRaFw0yNjA0MTkxNTE5MzRaMHUxCzAJBgNVBAYTAkRFMQ8wDQYDVQQI
    DAZCZXJsaW4xDzANBgNVBAcMBkJlcmxpbjEQMA4GA1UECgwHZ2VtYXRpazEQMA4GA1UECwwHZ2VtYXR
    pazEgMB4GA1UEAwwXSGFzaHRyZWUtU2lnbmVyLUV4YW1wbGUwWTATBgcqhkjOPQIBBggqhkjOPQMBBw
    NCAAQyznSs1COkntQx4mP/x6AuGbaj8Qk5LlN0e98EefU9vKu1pd12xkWWVocpca91Kvg1gVj67zvgl
    pEHM8Gp49V3o1MwUTAdBgNVHQ4EFgQUusy3zmQdvQQDNFZ7WQyg/NNZ1iIwHwYDVR0jBBgwFoAUusy3
    zmQdvQQDNFZ7WQyg/NNZ1iIwDwYDVR0TAQH/BAUwAwEB/zAKBggqhkjOPQQDAgNIADBFAiEA7a2R1b9
    XclYCKk2sElHW7LfnrnYcRz/gOw/dcGeNUncCIE452YKIOShdl+FQLjKC4FBpphhTZBHxVF3fdy6/GN
    pn"}.
Beispieldokument.
{"ht_path":["26vUSCpvz/vmJJGBTWMtK+w8UYKAHD5qb1rUwJJwsJk=",
"RM/ybgwXtf7etsW4Dw03kSdNdYGtnI6K4dueVaVQNeM=",
"fLcJYyrzSXCHlMCnJdyKdzAOw5LEls2ZzWimguDbjr0=",
"-oGj3kc81ouCxuNGjXFFE2IeryrDgKZET9mQ0moaK010="],
"ecdsa_sig":"MEYCIQDUaTbcbOknpp3REdEqpCZdrEThURF+Yf1W672vHWPpPgIhAKlHNjwvpgMQzps9Zjj
k4atG0lfSyaQ79LGJ/VBiVf3o"}
```

Beispiel ASN.1/DER-Signatur

```
    0:d=0  hl=4 l= 837 cons: SEQUENCE
    4:d=1  hl=2 l=  68 cons: SEQUENCE
    6:d=2  hl=2 l=  32 prim: INTEGER
        :7D2EE6E261537C4D065176BCE7F5D90D5710A8E878644E78A2DAA0BD6BB0C252
   40:d=2  hl=2 l=  32 prim: INTEGER
       :3959AEE6E72AE6D01D658963B890E117253FC69AFD58D96D8B52DA2293495E70
   74:d=1  hl=3 l= 185 cons: SEQUENCE
   77:d=2  hl=2 l=  44 prim: UTF8STRING
       :FddZi1q4KMNs6GcFOnSmIF/ZBSwgncGGWywnJS18iP0=
  123:d=2  hl=2 l=  44 prim: UTF8STRING
      :/Mm9XPqKj/xP0mNuCXy+Gpf0trD6+KNPvXK9ye1WHdY=
  169:d=2  hl=2 l=  44 prim: UTF8STRING
      :SF+G0DOpS4ucRnsthu0aSTB0MLa8nnnOVLq16y2YuXI=
  215:d=2  hl=2 l=  45 prim: UTF8STRING
      :-39TzZlbjT2ZztcVGmXG0VqXxYDehraPKVkAlkSu/fIo=
  262:d=1  hl=4 l= 575 cons: SEQUENCE
  266:d=2  hl=4 l= 485 cons: SEQUENCE
  270:d=3  hl=2 l=   3 cons: cont [ 0 ]
  272:d=4  hl=2 l=   1 prim: INTEGER           :02
  275:d=3  hl=2 l=  20 prim: INTEGER
      :6EFA2381C4AE10C544EF43277C1B42A17F87487F
  297:d=3  hl=2 l=  10 cons: SEQUENCE
  299:d=4  hl=2 l=   8 prim: OBJECT            :ecdsa-with-SHA256
  309:d=3  hl=2 l= 117 cons: SEQUENCE
  311:d=4  hl=2 l=  11 cons: SET
  313:d=5  hl=2 l=   9 cons: SEQUENCE
  315:d=6  hl=2 l=   3 prim: OBJECT            :countryName
  320:d=6  hl=2 l=   2 prim: PRINTABLESTRING   :DE
  324:d=4  hl=2 l=  15 cons: SET
  326:d=5  hl=2 l=  13 cons: SEQUENCE
  328:d=6  hl=2 l=   3 prim: OBJECT            :stateOrProvinceName
  333:d=6  hl=2 l=   6 prim: UTF8STRING        :Berlin
  341:d=4  hl=2 l=  15 cons: SET
  343:d=5  hl=2 l=  13 cons: SEQUENCE
  345:d=6  hl=2 l=   3 prim: OBJECT            :localityName
  350:d=6  hl=2 l=   6 prim: UTF8STRING        :Berlin
  358:d=4  hl=2 l=  16 cons: SET
  360:d=5  hl=2 l=  14 cons: SEQUENCE
  362:d=6  hl=2 l=   3 prim: OBJECT            :organizationName
  367:d=6  hl=2 l=   7 prim: UTF8STRING        :gematik
  376:d=4  hl=2 l=  16 cons: SET
  378:d=5  hl=2 l=  14 cons: SEQUENCE
  380:d=6  hl=2 l=   3 prim: OBJECT            :organizationalUnitName
  385:d=6  hl=2 l=   7 prim: UTF8STRING        :gematik
  394:d=4  hl=2 l=  32 cons: SET
  396:d=5  hl=2 l=  30 cons: SEQUENCE
  398:d=6  hl=2 l=   3 prim: OBJECT            :commonName
  403:d=6  hl=2 l=  23 prim: UTF8STRING        :Hashtree-Signer-Example
  428:d=3  hl=2 l=  30 cons: SEQUENCE
  430:d=4  hl=2 l=  13 prim: UTCTIME           :210420151934Z
  445:d=4  hl=2 l=  13 prim: UTCTIME           :260419151934Z
```

```
460:d=3  hl=2 l= 117 cons: SEQUENCE
462:d=4  hl=2 l=  11 cons: SET
464:d=5  hl=2 l=   9 cons: SEQUENCE
466:d=6  hl=2 l=   3 prim: OBJECT            :countryName
471:d=6  hl=2 l=   2 prim: PRINTABLESTRING   :DE
475:d=4  hl=2 l=  15 cons: SET
477:d=5  hl=2 l=  13 cons: SEQUENCE
479:d=6  hl=2 l=   3 prim: OBJECT            :stateOrProvinceName
484:d=6  hl=2 l=   6 prim: UTF8STRING        :Berlin
492:d=4  hl=2 l=  15 cons: SET
494:d=5  hl=2 l=  13 cons: SEQUENCE
496:d=6  hl=2 l=   3 prim: OBJECT            :localityName
501:d=6  hl=2 l=   6 prim: UTF8STRING        :Berlin
509:d=4  hl=2 l=  16 cons: SET
511:d=5  hl=2 l=  14 cons: SEQUENCE
513:d=6  hl=2 l=   3 prim: OBJECT            :organizationName
518:d=6  hl=2 l=   7 prim: UTF8STRING        :gematik
527:d=4  hl=2 l=  16 cons: SET
529:d=5  hl=2 l=  14 cons: SEQUENCE
531:d=6  hl=2 l=   3 prim: OBJECT            :organizationalUnitName
536:d=6  hl=2 l=   7 prim: UTF8STRING        :gematik
545:d=4  hl=2 l=  32 cons: SET
547:d=5  hl=2 l=  30 cons: SEQUENCE
549:d=6  hl=2 l=   3 prim: OBJECT            :commonName
554:d=6  hl=2 l=  23 prim: UTF8STRING        :Hashtree-Signer-Example
579:d=3  hl=2 l=  89 cons: SEQUENCE
581:d=4  hl=2 l=  19 cons: SEQUENCE
583:d=5  hl=2 l=   7 prim: OBJECT            :id-ecPublicKey
592:d=5  hl=2 l=   8 prim: OBJECT            :prime256v1
602:d=4  hl=2 l=  66 prim: BIT STRING
670:d=3  hl=2 l=  83 cons: cont [ 3 ]
672:d=4  hl=2 l=  81 cons: SEQUENCE
674:d=5  hl=2 l=  29 cons: SEQUENCE
676:d=6  hl=2 l=   3 prim: OBJECT            :X509v3 Subject Key Identifier
681:d=6  hl=2 l=  22 prim: OCTET STRING      [HEX
    DUMP]:0414BACCB7CE641DBD040334567B590CA0FCD359D622
705:d=5  hl=2 l=  31 cons: SEQUENCE
707:d=6  hl=2 l=   3 prim: OBJECT            :X509v3 Authority Key Identifier
712:d=6  hl=2 l=  24 prim: OCTET STRING      [HEX
    DUMP]:30168014BACCB7CE641DBD040334567B590CA0FCD359D622
738:d=5  hl=2 l=  15 cons: SEQUENCE
740:d=6  hl=2 l=   3 prim: OBJECT            :X509v3 Basic Constraints
745:d=6  hl=2 l=   1 prim: BOOLEAN           :255
748:d=6  hl=2 l=   5 prim: OCTET STRING      [HEX DUMP]:30030101FF
755:d=2  hl=2 l=  10 cons: SEQUENCE
757:d=3  hl=2 l=   8 prim: OBJECT            :ecdsa-with-SHA256
767:d=2  hl=2 l=  72 prim: BIT STRING
```

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
100000	0,930918	0,000002
200000	1,872791	0,000080
300000	2,805082	0,000202
400000	3,731741	0,000439
500000	4,662943	0,000116
600000	5,599445	0,000194
700000	6,582235	0,001204
800000	7,478881	0,000779
900000	8,428580	0,000988
1000000	9,348334	0,000771
1100000	10,272402	0,002986
1200000	11,369445	0,001956
1300000	12,407254	0,390596
1400000	13,271628	0,338491
1500000	14,251312	0,362921
1600000	15,201031	0,408057
1700000	16,177382	0,283987
1800000	17,055716	0,285772
1900000	17,950099	0,139814
2000000	18,812003	0,015483

Tabelle 1: Benchmark zufällige Partnerblätter (jeweils 50 Durchläufe)

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
100000	0,688023	0,000000
200000	1,379485	0,000037
300000	2,068095	0,000009
400000	2,749134	0,000002
500000	3,441168	0,000011
600000	4,134949	0,000073
700000	4,831345	0,000376
800000	5,512308	0,000105
900000	6,208901	0,000302
1000000	6,896439	0,000506
1100000	7,595647	0,000382
1200000	8,277472	0,000374
1300000	9,003504	0,000573
1400000	9,666620	0,000873
1500000	10,357738	0,001495
1600000	11,073031	0,001075
1700000	11,810290	0,012336
1800000	12,454754	0,000933
1900000	13,180336	0,004194
2000000	13,829987	0,000797

Tabelle 2: Benchmark pseudozufällige Partnerblätter (jeweils 50 Durchläufe)

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
100000	0,269122	0,000000
200000	0,534759	0,000000
300000	2,068095	0,000000
400000	1,581736	0,000004
500000	1,068765	0,000000
600000	0,555389	0,000003
700000	3,181108	0,000548
800000	2,661197	0,000002
900000	2,139075	0,000000
1000000	1,623946	0,000000
1100000	1,112295	0,000002
1200000	6,854665	0,000007
1300000	5,822050	0,000006
1400000	5,328743	0,000027
1500000	4,791610	0,000014
1600000	4,279110	0,000014
1700000	3,766989	0,000208
1800000	3,266134	0,000004
1900000	2,735466	0,000005
2000000	2,225578	0,000001

Tabelle 3: Benchmark Bauen des Baumes mit zufälligen Dummydokumente (jeweils 50 Durchläufe)

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
1	0,000001	0,000000
2	0,000002	0,000000
3	0,000009	0,000000
4	0,000004	0,000000
5	0,000034	0,000000
6	0,000020	0,000000
7	0,000013	0,000000
8	0,000007	0,000000
9	0,000050	0,000000
10	0,000045	0,000000
20	0,000088	0,000000
30	0,000037	0,000000
40	0,000177	0,000000
50	0,000126	0,000000
60	0,000074	0,000000
70	0,000404	0,000000
80	0,000352	0,000000
90	0,000502	0,000000
100	0,000356	0,000000
200	0,001245	0,000001
300	0,001245	0,000000
400	0,001075	0,000000
500	0,000804	0,000000
600	0,003661	0,000005
700	0,002574	0,000000
800	0,001974	0,000000
900	0,001464	0,000000
1000	0,001617	0,000001
2000	0,002344	0,000001
3000	0,009590	0,000014
4000	0,004415	0,000005
5000	0,023742	0,000016
6000	0,018468	0,000017
7000	0,013304	0,000012
8000	0,008201	0,000009
9000	0,051320	0,000008
10000	0,046910	0,000016
20000	0,092593	0,000017
30000	0,041417	0,000018
40000	0,186032	0,000016
50000	0,133107	0,000003
60000	0,082273	0,000018
70000	0,421179	0,000017
80000	0,368698	0,000018
90000	0,317529	0,000018
2500000	12,087048	0,000080
3000000	9,520731	0,000074

Tabelle 4: Benchmark Bauen des Baumes mit zufälligen Dummydokumente (jeweils 50 Durchläufe)

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
100000	0,115684	0,000000
200000	0,231371	0,000000
300000	2,068095	0,000000
400000	0,462918	0,000000
500000	0,437239	0,000000
600000	0,978569	0,000000
700000	0,952120	0,000000
800000	0,926029	0,000001
900000	0,899436	0,000000
1000000	0,873866	0,000000
1100000	1,983927	0,000001
1200000	1,956947	0,000000
1300000	1,929996	0,000000
1400000	1,903548	0,000000
1500000	1,878064	0,000001
1600000	1,852686	0,000001
1700000	1,826869	0,000005
1800000	1,799570	0,000000
1900000	1,772847	0,000001
2000000	1,746531	0,000000

Tabelle 5: Benchmark Bauen des Baumes mit festem Hash als Dummydokumente (jeweils 50 Durchläufe)

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
1	0,000001	0,000000
2	0,000002	0,000000
3	0,000004	0,000000
4	0,000004	0,000000
5	0,000012	0,000000
6	0,000009	0,000000
7	0,000007	0,000000
8	0,000007	0,000000
9	0,000016	0,000000
10	0,000015	0,000000
20	0,000030	0,000000
30	0,000027	0,000000
40	0,000059	0,000000
50	0,000056	0,000000
60	0,000055	0,000000
70	0,000120	0,000000
80	0,000125	0,000000
90	0,000125	0,000000
100	0,000119	0,000000
200	0,000233	0,000000
300	0,000471	0,000000
400	0,000449	0,000000
500	0,000435	0,000000
600	0,000938	0,000000
700	0,000912	0,000000
800	0,000887	0,000000
900	0,000862	0,000000
1000	0,000836	0,000000
2000	0,001676	0,000000
3000	0,003603	0,000000
4000	0,003347	0,000000
5000	0,007453	0,000000
6000	0,007200	0,000000
7000	0,006944	0,000000
8000	0,006690	0,000000
9000	0,015142	0,000000
10000	0,014885	0,000000
20000	0,029841	0,000000
30000	0,027301	0,000000
40000	0,059726	0,000000
50000	0,057250	0,000000
60000	0,054693	0,000000
70000	0,122248	0,000000
80000	0,119698	0,000000
90000	0,117155	0,000000
2500000	3,843015	0,000004
3000000	3,715021	0,000006

Tabelle 6: Benchmark Bauen des Baumes mit festem Hash als Dummydokumente (jeweils 50 Durchläufe)

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
100000	1,219732	0,000602
200000	2,023177	0,000144
300000	2,990753	0,000163
400000	3,6637	0,000034
500000	4,328182	0,000303
600000	5,607684	0,000712
700000	6,263219	0,000512
800000	6,931243	0,000581
900000	7,573547	0,000334
1000000	8,251615	0,000581
1100000	10,15239	0,001390
1200000	10,812945	0,000717
1300000	11,461879	0,000755
1400000	12,104806	0,000690
1500000	12,752283	0,001045
1600000	13,421755	0,001618
1700000	14,066962	0,001335
1800000	14,738203	0,001242
1900000	15,857494	0,001832
2000000	16,524762	0,001488

Tabelle 7: Benchmark des kompletten Programmes mit jws (jeweils 10 Durchläufe)

Anzahl Dokumente	durchschnittliche Zeit in s	Varianz in s^2
100000	1,215093	0,000047
200000	2,032724	0,000025
300000	3,009558	0,000117
400000	3,668856	0,000055
500000	4,341472	0,000207
600000	5,636794	0,000604
700000	6,285723	0,000177
800000	6,950541	0,000213
900000	7,62448	0,001476
1000000	8,282158	0,000519
1100000	10,203104	0,001090
1200000	10,866922	0,000218
1300000	11,504612	0,000623
1400000	12,187555	0,000526
1500000	12,842788	0,000533
1600000	13,531476	0,001191
1700000	14,171153	0,000768
1800000	14,757907	0,002387
1900000	15,408175	0,001121
2000000	16,086108	0,001364

Tabelle 8: Benchmark des kompletten Programmes mit asn1 (jeweils 10 Durchläufe)

Länge des Verifizierungspfades	durchschnittliche Zeit in s	Varianz in s^2
1	0,000002	0,000000
2	0,000003	0,000000
3	0,000005	0,000000
4	0,000005	0,000000
5	0,000007	0,000000
6	0,000008	0,000000
7	0,000009	0,000000
8	0,000010	0,000000
9	0,000011	0,000000
10	0,000012	0,000000
11	0,000014	0,000000
12	0,000015	0,000000
13	0,000015	0,000000
14	0,000016	0,000000
15	0,000017	0,000000
16	0,000018	0,000000
17	0,000019	0,000000
18	0,000020	0,000000
19	0,000021	0,000000
20	0,000022	0,000000
21	0,000023	0,000000
22	0,000024	0,000000

Tabelle 9: Benchmark des Berechnens der Wurzel aus dem Verifizierungspfad (jeweils 100.000 Durchläufe)

Länge des Verifizierungspfades	durchschnittliche Zeit in s	Varianz in s^2
1	0,000142	0,000000
2	0,000143	0,000000
3	0,000147	0,000000
4	0,000148	0,000000
5	0,000148	0,000000
6	0,000154	0,000000
7	0,000153	0,000000
8	0,000158	0,000000
9	0,000159	0,000000
10	0,000161	0,000000
11	0,000167	0,000000
12	0,000169	0,000000
13	0,000169	0,000000
14	0,000170	0,000000
15	0,000175	0,000000
16	0,000177	0,000000
17	0,000180	0,000000
18	0,000183	0,000000
19	0,000185	0,000000
20	0,000187	0,000000
21	0,000191	0,000000
22	0,000193	0,000000

Tabelle 10: Benchmark der kompletten Verifikation (jeweils 100.000 Durchläufe)

Literatur

[1] Andreas Hülsing, Denis Butin, Stefan Gazdag, Joost Rijneveld, and Aziz Mohaisen. Xmss: extended merkle signature scheme. Technical report, 2018.

[2] Utimaco IS GmbH. Securityserver se. `https://hsmsecurity.pl/wp-content/uploads/2016/08/DataSheet-Seria-Se.pdf`. Accessed: 26.01.2023.

[3] David Wedekind. Bachelorarbeit. `https://scm.cms.hu-berlin.de/wedekind/bachelorarbeit/-/commit/4ca8b821e9686be0318f558dd8614b9847b62eb4`, 2023.

[4] Cameron F Kerry and Patrick D Gallagher. Digital signature standard (dss). *FIPS PUB*, pages 186 4, 2013.

[5] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[6] Bundesamt für Sicherheit in der Informationstechnik (BSI). Kryptographische verfahren: Empfehlungen und schlüssellängen. *Technische Richtlinie BSI TR-02102-1, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Deutschland*, 2023.

[7] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36 63, 2001.

[8] Ralph C Merkle. Protocols for public key cryptosystems. In *Secure communications and asymmetric cryptosystems*, pages 73 104. Routledge, 2019.

[9] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218 238. Springer, 1989.

[10] Stathis Mavrovouniotis and Mick Ganley. Hardware security modules. In *Secure Smart Embedded Devices, Platforms and Applications*, pages 383 405. Springer, 2014.

[11] Dejan Vujičić, Dijana Jagodić, and Siniša Ranđić. Blockchain technology, bitcoin, and ethereum: A brief overview. In *2018 17th international symposium infoteh-jahorina (infoteh)*, pages 1 6. IEEE, 2018.

[12] Scott Chacon and Ben Straub. *Pro git*. Springer Nature, 2014.

[13] Arnaud Legout, Guillaume Urvoy-Keller, and Pietro Michiardi. Understanding bittorrent: An experimental perspective. 2005.

[14] Arno Bakker. Bittorrent - merkle hash torrent extension. `http://bittorrent.org/beps/bep_0030.html`, 2016. Accessed: 26.01.2023.

[15] Certificate transparency. https://certificate.transparency.dev/. Accessed: 26.01.2023.

[16] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure logging schemes and certificate transparency. In *European Symposium on Research in Computer Security*, pages 140 158. Springer, 2016.

[17] Bundesamt für Sicherheit in der Informationstechnik (BSI). Beweiswerterhaltung kryptographisch signierter dokumente (tr-esor). *Technische Richtlinie BSI TR-03125, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Deutschland*, 2022.

[18] Tobias Gondrom, Ralf Brandner, and Ulrich Pordesch. Evidence record syntax (ers). Technical report, 2007.

[19] European Telecommunications Standards Institute (ETSI). Etsi sr 019 510:ëlectronic signatures and infrastructures (esi)". Technical report, 2017.

[20] Nationale Agentur für Digitale Medizin (gematik). Spezifikation implementierungsleitfaden primärsysteme e-rezept. Technical report, 2022.

[21] gematik fachportal - elekrtonisches rezept. https://fachportal.gematik.de/anwendungen/elektronisches-rezept. Accessed: 26.01.2023.

[22] Nationale Agentur für Digitale Medizin (gematik). Systemspezifisches konzept e-rezept 1.1.0. Technical report, 2020.

[23] Nationale Agentur für Digitale Medizin (gematik). Spezifikation e-rezept-fachdienst 1.5.0. Technical report, 2022.

[24] Openssl. https://www.openssl.org/. Accessed: 26.01.2023.

[25] json-c. https://github.com/json-c/json-c. Accessed: 26.01.2023.

[26] Bundesamt für Sicherheit in der Informationstechnik (BSI). Kryptographische vorgaben für projekte der bundesregierung teil 4: Kommunikationsverfahren in anwendungen. *Technische Richtlinie BSI TR-03116*, 2022.

[27] Michael B Jones. The emerging json-based identity protocol suite. In *W3C workshop on identity in the browser*, pages 1 3, 2011.

[28] Russ Housley, William Polk, Warwick Ford, and David Solo. Rfc3280: Internet x. 509 public key infrastructure certificate and certificate revocation list (crl) profile, 2002.

[29] Michael Jones, John Bradley, and Nat Sakimura. Json web signature (jws). Technical report, 2015.

[30] Russell Housley, Warwick Ford, William Polk, and David Solo. Internet x. 509 public key infrastructure certificate and crl profile. Technical report, 1999.

[31] IEEE and The Open Group. Ieee std 1003.1-2017. Technical report, 2017.

[32] Tobias Kind. Ramdisk benchmarks. *University of California*, 52, 2011.

[33] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 27. Januar 2023 David Wedelind