Hochschule Flensburg

B A C H E L O R – T H E S I S

Thema: Hashed Ed25519 Signaturverfahren über verteiltes Schwellwert-System

von: Nico Hohm

Matrikel-Nr.: 590305

Studiengang: Angewandte Informatik

Betreuer/in und Erstbewerter/in: Osmanbey Uzunkol

Zweitbewerter/in: Tim Aschmoneit

Ausgabedatum: 25.05.2020

Abgabedatum: 25.07.2020

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Thesis ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen benutzt habe.

_____

Nico Hohm

Contents

1 Begriffsverzeichnis **4**

I Einführung **4**
1.1 Thresholdizing HashEdDSA: MPC to the Rescue 5

II Multiple Party Computing MPC **6**

2 Protokoll daBit **6**
2.1 Umsetzung 7

3 MPC Funktion **8**
3.1 Umsetzung 8

4 Access Structure **10**

III Signatur Algorithmen **10**

5 Einführung **11**
5.1 Sign(message, sk, pk) 11
5.2 verify(signature, m, pk) 11

6 Schnorr Signatur **11**
6.1 Schnorr Sign: 12
6.2 Schnorr Verify((e||s),m): 12

7 Hashed Ed25519 Signatur Algorithmus **12**

8 Edwards Kurve **16**
8.1 Umsetzung 17

9 Diskretes Logarithmus Problem (DSP) **18**
9.1 DSP bei Elliptischen Kurven (ECDLP) 19

IV JavaScript Anwendung **19**

10 Modulo Division **20**

11 Encode/Decode **20**

12 Serverabfragen **21**

V Funktionstests **22**

13 Library Version **22**

14 Eigene Version **23**

1 Begriffsverzeichnis

PK Public Key/Öffentlicher Schlüssel

SK Private Key/Geheimer Schlüssel

q große Primzahl der Edwardskurve

p Ordnung der Edwardskurve bzw. Anzahl der Punkte auf der Kurve

$P1$ Partei 1, initialisiert jeweilige Funktion

MPC Multiple Party Computing/Berechnung über mehrere Parteien

Integer Datentyp für ganze Zahlen

Part I
Einführung

Motivation

Digitale Sicherheit ist in unserer heutigen Zeit wichtiger als je zuvor. Mit der fortschreitenden Digitalisierung des Alltags besitzt so gut wie jeder ein Passwort, einen Account oder eine Datei die es zu schützen gilt. Datenlecks und der unsichere Umgang mit Benutzerdaten sind nun Stoff für Schlagzeilen. Als Durchschnittsbürger, aber vor allem auch als Softwareentwickler ist es deshalb von großer Wichtigkeit sich angemessen mit der Sicherheit seiner Daten auseinanderzusetzen.

Mit der neugefundenen Popularität von Kryptowährungen gelangten selbst bestimmte kryptografische Verfahren wie Blockchain in das öffentliche Bewusstsein, entsprechend finden sich immer mehr Menschen mit Interesse daran mehr über Kryptografie zu lernen.

Meine persönliche Motivation lag vor allem darin nach den Veranstaltungen Kryptografie und IT Security die gelernten Konzepte zum ersten Mal in Code umzusetzen. Die Veranstaltungen waren größtenteils theoretisch gehalten, am interessantesten fand ich dabei simplere kryptografische Abläufe wie den Diffie-Hellman Schlüsselaustausch schriftlich mit konkreten Zahlen durchzurechnen.

Der Reiz am Programmieren kam für mich daher, dass ein Signaturschema hauptsächlich aus einer Folge von relativ simplen mathematischen Operationen besteht. Die Logik des Schemas war also stets transparent, die Herausforderung

bestand darin die Bestandteile effizient umzusetzen und sie richtig zusammenzusetzen. Es kam dabei oft vor, dass mir Zusammenhänge erst wirklich klar wurden, nachdem ich anfing sie im Code umzusetzen.

Beim Aufarbeiten der gegebenen Arbeit hatte ich an vielen Stellen Ideen, zur vereinfachten Darstellung, die ich in dieser Ausarbeitung verwenden könnte.

Ziel der Thesis

Ziel dieser Thesis ist es das Signatur Schema, das in der gegebenen Arbeit "Thresholdizing HashEdDSA: MPC to the Rescue" von Charlotte Bonte, NigelP. Smart und Titouan Tanguy [1] beschrieben ist, in vereinfachter Form umzusetzen und die enthaltenen Vorgänge für mit dem Thema unvertraute Leser zugänglicher zu machen.

Bei der zugehörigen JavaScript Umsetzung wurde der Fokus vor allem auf intuitive Verwendung gelegt um die in der Thesis beschriebenen Konzepte interaktiv nachzuvollziehen. Sie unterscheidet sich Stellenweise von der Beschreibung in der Quelle und ist nicht als eine praktisch einsetzbare Lösung gedacht, die den beschriebenen Sicherheitsstandards entspricht.

Bestenfalls soll der Leser ein grundlegendes Verständnis für Signaturalgorithmen, elliptische Kurven Kryptografie und Multiple Party Computing anhand des Hashed Ed25519 Signatur Algorithmus als Beispiel entwickeln. Dieses Verständnis kann er anschließend nutzen, um die technischen Verläufe der JavaScript Anwendung nachzuvollziehen und sein Wissen mit dieser praktischen Umsetzung zu festigen.

1.1 Thresholdizing HashEdDSA: MPC to the Rescue

Ziel des ursprünglichen Dokuments ist es, basierend auf dem NIST Standard HashEdDSA Protokoll einen digitalen Signatur Algorithmus (DSA) zu konstruieren, der in einem System von mehreren Parteien (multiple partie computing MPC) ausgeführt wird.

Eine Zugriffsstruktur Q2 legt hierzu eine Anzahl von vertrauenswürdigen Parteien fest, die sich am Algorithmus beteiligen müssen, dies hat den Vorteil, dass ein Angreifer eine Mehrzahl von Parteien korrumpieren muss, um eigenständig eine Signatur fälschen zu können.

Der verwendete Signatur Algorithmus, hier Hashed Ed25519 bleibt dabei unverändert. Das Berechnen eines gemeinsamen Private Keys passiert das Protokoll daBit (doppelt authentifizierte Bits), welches MPC Funktionen benutzt, um eine bestimmte Anzahl zufälliger Bits zwischen beliebig vielen Parteien zu berechnen. Das Protokoll generiert sowohl zufällige Bits, als auch zufällig Zahlen modulo q.

Es werden grundlegende MPC Funktionen, wie z.B. Input, Output und Random zu Verfügung gestellt, die voraussetzen, dass alle teilnehmenden Parteien sich mit einem einmaligen Identifier varid ausweisen, um der Ausführung der Funktion zuzustimmen.

Außerdem wird bei der Ausgabe von Punkten des Signatur Algorithmus ein daBit ähnliches Verfahren verwendet, um die Punkte als Teilwerte zu übertragen und hinterher von den Parteien gemeinsam berechnen zu lassen.

Part II
Multiple Party Computing MPC

2 Protokoll daBit

Das Protokoll daBit ermöglicht den Austausch eines von P1 generierten zufälligen Bitstrings zwischen den Parteien, welcher später als Private Key im Signatur Schema genutzt wird. Die Operationen zum Austausch passieren verteilt über die Parteien (MPC).

Die Sicherheit dieses Protokolls wird durch die Übertragung mit Hilfe der MPC Funktionen gewährleistet (siehe Abschnitt 5).

Der Austausch basiert auf dem Prinzip eine Zahl in mehreren Teilen zu übertragen. Jede Partei bekommt einen Teil der Zahl geschickt.

Addieren alle Parteien ihre Parts zusammen (modulo q), so erhalten sie die ursprüngliche Zahl zurück.

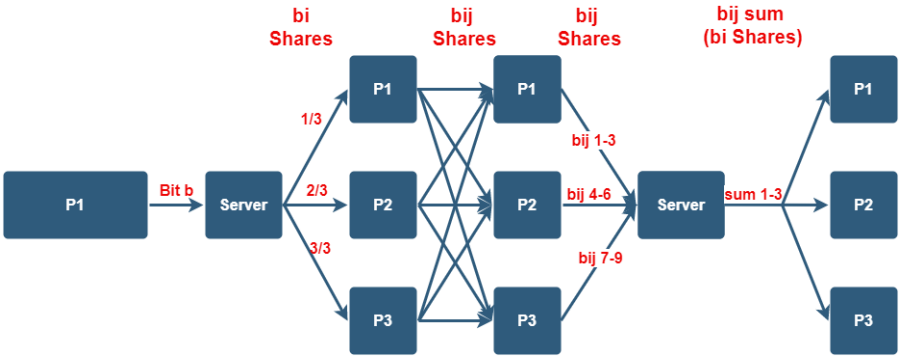

Figure 1: daBit Übermittlung von b mit 3 Parteien in Q1

In der daBit Funktion wird jede Stelle eines Arrays mit zufälligen Zahlen (modulo q) auf diese Weise an die Parteien gegeben. Die erhaltenen Teile der Zahl werden dann erneut auf diese Weise aufgeteilt. Deren erneut gespaltenen Teile werden dann an alle Parteien gesendet. Jede Partei hat dann die benötigten Teile, um mit 2 Zusammensetzungen auf die ursprüngliche Zahl zu kommen. Dieser Vorgang passiert erneut mit jeder Zahl in binär (modulo 2), anschließend wird das Ergebnis mit der *Frand* Funktion überprüft.

2.1 Umsetzung

Variablen:

m: Länge des Bitstring Outputs, 256

Q1: Anzahl kleinste valide Gruppe mit P1

$\triangle$: $q/Q1$

sec: security parameter = 128

γ: $sec + 1 = 129$

Array Sj: Arrays mit den wieder zusammengefügten Werten b (Binär und modulo q)

$P1$ generiert $m+\gamma \cdot sec = 16768$ random Integers b zwischen $0-q$. Jeder Integer wird in $Q1$ Teile, genannt bi gespalten, so dass $\sum_{i \in Q1} bi = b$ ihre Modulo-Summe wieder bi ergibt. Die $biArrays$ werden an die Parteien in $Q1$ gesendet.
ANMERKUNG: In der JavaScript Anwendung werden statt, einzelner Werte zu versenden Arrays, mit allen Werten gefüllt und dann erst versendet.

$P1$ berechnet den Wert Ki
$Ki = \left[\frac{\triangle \cdot \sum_{j \in Q1} hij}{q} \right]$

Anschließend lassen alle Parteien ihren Teil bi erneut teilen, um die Anteile bij zu erhalten, diese werden an alle Parteien in $Q1$ gesendet. Die Parteien senden ihre jeweiligen Anteile zum Server. Der Server addiert die Teile, um die ursprünglichen bi Werte zu erhalten und sendet diese an alle Parteien in $Q1$. Jede Partei ist nun im Besitz von $Q1$ Anteilen, um den ursprünglichen Integer b zusammen zu addieren.

Der gesamte Prozess wird wiederholt, allerdings gibt $P1$ statt dem random Integer b zu Beginn den Wert $(bi1 - ki * q)\%2$ ein, welcher einer binären Version von b, $b2$ entspricht.
Das ursprüngliche Bit b wird vom Server zu keinem Zeitpunkt weiterverschickt. Der letzte Schritt, um das Bit b zu erhalten ist es, die Summen aus bij (welche den bi Shares entsprechen) zu bilden. In der JavaScript Umsetzung wird diese letzte Summe im Gegensatz zu den anderen Summenberechnungen auf Clientseite ausgeführt.

Zum Überprüfen der erhaltenen Daten wird die Funktion $Frand$ verwendet. Für jedes b im $Array\,Sj$ werden sec random Bits rij generiert. Die Summe von $rij \cdot bi$ wird in einem Array der Größe sec gespeichert.

Die Funktion wird jeweils mit Ergebnis Arrays von $\langle bi \rangle_q$ und $\langle bi \rangle_2$ ausgeführt. Das Ergebnis sind Zwei Arrays der Größe sec mit Bits bzw. Zahlen modulo q. Es wird überprüft, ob jede Stelle des $qArrays$ modulo 2 dem Bit an der entsprechenden Stelle des Bit Arrays entspricht. Sollte dies der Fall für alle Stellen j Sein sind b und b_2 valide. Die Funktion gibt die ersten m Stellen von b_2 aus.

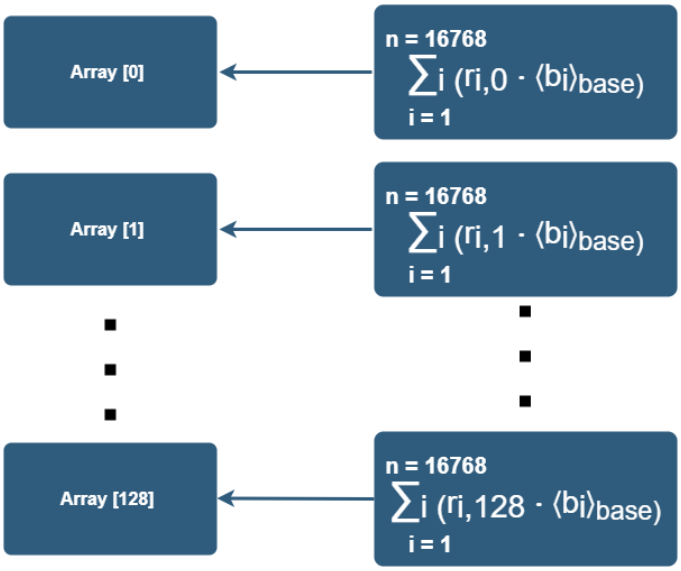

Figure 2: Berechnung desr frand Arrays der Länge sec

Zur Erinnerung, Bit b_2 sollte stets b modulo 2 entsprechen. Außerdem ist die Ordnung der Edwards Kurve q auf Grund der Kurvensymmetrie immer gerade.

Die Definition von daBit wurde der ursprünglichen Arbeit [1] entnommen.

3 MPC Funktion

Unter MPC sind eine Reihe von Funktionen definiert, die zur Sicherheit beim Übertragen von Daten innerhalb des MPC Systems dienen. Beim Aufruf von einer Partei $P1$ benötigen die Funktionen zusätzlich den Input aller anderen Parteien im System. Zur Identifizierung wird jeder teilnehmenden Partei ein Set mit Identifikationsschlüsseln *varid* zur Verfügung gestellt. Bei jedem Input und Funktionsaufruf müssen die Parteien ein bisher unbenutztes *varid* mitübergeben.

3.1 Umsetzung

Variablen

varid: Identifikationsschlüssel

varidOld: Ein bereits verwendetes varid wird übergeben, um einen gespeicherten Wert abzurufen

domain: Der Wert des Modulo Raums (hier entweder q oder 2)

x,y: Wert einer Zahl (modulo domain)

Cf: Eine Funktion $Cf(x) => y$, hier Hash Funktion Sha512

Anmerkung: Die nach den Funktionen benannten Parameter (Init, Input, Random, etc.) dienen zur Identifikation des Funktionsaufrufs. Zum Verständnis können sie als Strings betrachtet werden, die vom Server abgefragt werden, um die erhaltenen Nachricht (Input) einzuordnen.
Werte x werden stets in der Form (varid, domain, x) gespeichert.

Bei der JavaScript Umsetzung wurden viele MPC Funktionen geändert oder ausgelassen, da sie entweder überflüssig wurden, oder um das Programm zu vereinfachen.

Initialize: Bei Input (Init) von allen Parteien werden die F mpc Funktion aktiviert, vorherige Inputs werden verworfen.

Änderung: Alle Parteien senden eine Nachricht zur Initialisierung an den Server, allerdings überprüft dieser nicht die Vollständigkeit aller Parteien.

Input: Bei Input (Input, Pi, varid, domain,x) von Pi und (Input, Pi, varid, domain) von allen anderen Parteien mit einem unbenutzten varid wird (varid, domain, x) von Pi gespeichert.

Änderung: Es wurde auf die zusätzlichen Inputs der anderen Parteien verzichtet, P1 sendet Nachrichten direkt zum Server ohne weitere Abfragen.

Random: Bei Input (Random, varid, domian) von allen Parteien mit unbenutzten varid wird eine zufällige ganze Zahl x (modulo domain) generiert und unter (varid, domain,x) gespeichert.

Änderung: Random Values werden auf Client Seite generiert.

Evaluate: Bei Input (varid, varidOld, domain,Cf) von allen Parteien, wenn varidOld bereits vorhanden ist, wird der zugehörige Wert x von varidOld abgerufen. Für jedes abgerufene x speichert die Funktion (varid, domain, y) mit $y <= Cf(x)$.

Änderung: Da es sich bei dem Parameter Cf immer um die Hashfunktion Sha512 handeln wird, stellt der Server stattdessen eine Hash Funktion zur Verfügung.

Output: Bei Input (Output, varidOld, domain, type) von allen Parteien:
Bei type = 0: Öffentlicher Output, sende (varid, y) an alle Parteien.
Bei type = -1: Kein Output, speichere (varid, y) und fahre ohne Output fort.
Bei type > 0: Privater Output, sende y and Pi mit i = type

Änderung: Die verschiedenen Output Typen wurden von Socket.io Funktionen zur Verfügung gestellt und es wurde auf zusätzliche Inputs der anderen Parteien verzichtet.

Convert: Bei Input (Convert, varidOld, domain1, varid, domain2) einer Partei wird der Wert x mit varidOld und domain1 aufgerufen, ist dieser vorhanden wird (varid,domain2,y) mit $y <= (x)\%domain2$ gespeichert.

Änderung: Es wurde auf eine Convert Funktion verzichtet, da die verwendeten Variablen keinen separaten Datentypen für Binär und Dezimal benutzen.

Allgemein: Statt einem System, bei dem jeder verifizierte Nutzer ein Set I mit Identifiern erhält, von denen er einen unbenutzten Identifier mit jeder Nachricht benutzt, wurde ein vereinfachtes System basierend auf Socket.io Ids benutzt.

MPC Funktionen zum Generieren von random Integern bzw. Bits wie GenBit() und Rand() wurden mit dem JavaScript Random Number Generator auf Clientseite ersetzt.

Die Definition der MPC Funktionen wurde der ursprünglichen Arbeit [1] und einer der darin enhaltenen Quellen [8] entnommen.

4 Access Structure

Um die Anzahl der benötigten Parteien zu bestimmen, die für die verteilten Berechnungen nötig sind wird eine Zugangsstruktur $Q2$ definiert. Diese berechnet einen Grenzwert t (threshold) anhand der Anzahl von Parteien n. Der Grenzwert t ist stets $t < n/2$ und es werden mindestens $t+1$ vertrauenswürdige Parteien benötigt, um eine Operation durchzuführen, sprich mehr als die Hälfte der Parteien muss vertrauenswürdig sein. Außerdem muss die Anzahl von vertrauenswürdigen Parteien über 2 liegen.

Zur vereinfachten Darstellung und leider auch aus zeitlichen Gründen benutzt die JavaScript Anwendung eine vereinfachte Version von $Q2$. Parteien weisen sich dabei nicht mit einem *varid* aus, sondern können im Client selbst entscheiden, ob sie korrupt oder vertrauenswürdig sind, der oben beschriebene Grenzwert überprüft, ob genug Parteien vertrauenswürdig sind, um daBit auszuführen.

Die Definition der AccessStructure wurde der ursprünglichen Arbeit [1] entnommen.

Part III
Signatur Algorithmen

5 Einführung

Grundlegend geht es bei einem Signaturalgorithmus darum zu überprüfen, ob eine Nachricht von einer zur Signatur berechtigten Partei kommt und bei der Übertragung unverändert geblieben ist. Anders als bei einer Verschlüsselung ist es nicht das Ziel den Inhalt der Nachricht geheim zu halten, Nachricht und Signatur können also öffentlich in Klartext übertragen werden. Hierzu werden Zwei Funktionen Definiert:

5.1 Sign(message, sk, pk)

Eine gewählte Nachricht wird genutzt, um eine Signatur zu generieren, diese Funktion ist nur möglich, wenn die Partei sowohl den Public Key pk, als auch den Private Key sk kennt, welcher nur ausgewählten Parteien bekannt sein soll. Optimal gäbe es zu jeder möglichen Nachricht eine einzigartige Signatur. Haben Zwei Nachrichten dieselbe Signatur nennt man dies Kollision.

5.2 verify(signature, m, pk)

Nachricht und Signatur werden überprüft, falls die Nachricht geändert wurde wird sie nicht mehr zur Signatur passen und wird nicht verifiziert. Diese Funktion steht jedem zur Verfügung, der den Public Key kennt, also theoretisch jeder Partei, da dieser bei Definition öffentlich ist.
Eine Signatur ist also eine Art Prüfsumme zur Nachricht, allerdings beweist sie auch die Herkunft der Nachricht, da ein Angreifer ohne sk nicht in der Lage ist sie zu generieren. Ziel einer Implementierung eines Signatur Schemas sollte also sein, dass eine Signatur möglichst einzigartig zu jeder Nachricht ist und die Eingabe bei kleinsten Änderungen an Nachricht oder Signatur nicht mehr verifiziert wird. Es soll einem Angreifer außerdem unmöglich sein von den öffentlichen Werten wie dem Public Key oder der Signatur auf den Private Key zurückzuschließen.

6 Schnorr Signatur

Die Grundlage von EdDSA ist die elliptische Kurven Version der Schnorr Signatur [6].

Sei $G(p)$ eine eine elliptische Kurve wobei p eine Primzahl ist. Der Private Key x ist ein Random Integer mit $0 < x < p$. Der Public Key Y ergibt sich aus $x \cdot G$, mit Generatorpunkt G als ein valider Punkt auf der Kurve $G(p)$. Außerdem wird eine Hashfunktion H mit dem Wertebereich 0–$(q-1)$ zur Verfügung gestellt.

6.1 Schnorr Sign:

- k = Random Integer mit $0 < k < p$
- Jedes k sollte nur einmal verwendet werden, da es dem Angreifer sonst erleichtert den Wert nachzuvollziehen.
- $R = k * G$
- $R.x$ = x Koordinate des Punktes R
- $e = H(R.x||m)$
- $s = (k + (x * e))\%p$
- e und q werden zur Basis 2 konvertiert. Falls e mehr Stellen als q hat werden die überschüssigen Stellen von der rechten Seite gestrichen.
- Die Signatur ergibt sich aus $(e||s)$

6.2 Schnorr Verify((e||s),m):

- $R = (s * G)–(e * Y)$
- Durch die Subtraktion von $e * Y$ kann auf $R = k * G$ zurückgeschlossen werden.
- $ev = H(R.x||m)$
- die Nachricht gilt als verifiziert, wenn $ev = e$

Bei zu niedrig gewähltem p oder einem mehrfach verwendeten Wert für k besteht die Gefahr, dass ein Angreifer den Wert k nachvollziehen kann. Aus der Formel $s = (k + (x * e))\%p$ wird deutlich, dass k neben x die einzige unbekannte ist, da e und s die Signatur bilden. Bei bekannten k könnte ein Angreifer als x mit $x = (s - k)/e$ ermitteln und so die Sicherheit des Schemas brechen.

Die Schnorr Signatur ist zwar kein Teil Anwendung, allerdings sind bereits viele Konzepte vorhanden, die im Hashed Ed25519 Algorithmus wichtig sind, wie z.B. die Berechnung der Punkte über G. Unvertrauten Lesern wird also nahegelegt sich zunächst mit dem Schnorr Algorithmus vertraut zu machen.

7 Hashed Ed25519 Signatur Algorithmus

Hashed Ed25519 ist ein bestimmter Signatur Algorithmus basierend auf der Edwardskurve, die über die Primzahl 2^{255}–19 definiert wurde. Die genutzte Hashfunktion SHA512 gibt Bitstrings mit einer Länge von 64 Oktetten (512Bits) aus. Die Kodierungen von Punkten und Integern sind jeweils 32 Oktette/ 256 Bits lang.

KeyGen(b,sec)

- Die Funktion wird mit dem Output von daBit b und $sec = 128$ aufgerufen, der BitString b dient als Secret Key.
- Der Public Key wird aus dem sha512 Hash H von sk generiert, in dem die erste Hälfte (bit 0-255) genommen wird
- -bit 0,1,2 und 255 auf 0 gesetzt werden -bit 254 auf 1 gesetzt wird.
- Der entstehende Bitstring wird zu einem Integer sq (modulo q) konvertiert.
- Der Basepoint auf der Kurve wird mit diesem Integer sq multipliziert.
- Der Public Key ist die Encodierung des entstehenden Punktes mit einer Länge von 256 Bits.

Sign(m,sk,pk)

- Die 2. Hälfte (bit 256-512) von H wird zusammen mit dem sha512 Hash der Nachricht m zusammen erneut in mit der sha512 Funktion zum Bitstring r gehasht.
- Der Bitstring wird zum Integer rq (mod q) konvertiert und der Basepoint wird mit diesem multipliziert.
- Der entstehende Punkt wird zum Bitstring R konvertiert.
- Der Integer aus $rq + sha512(R||pk|sha512(m)) * sq$ wird zum Bitstring S konvertiert.
- Die Signatur entsteht aus $R||S$. Da die Signatur aus 2 Kodierungen besteht, ist sie insgesamt 512 Bits lang.

Verify(R||S,m,pk)

- Die Signature wird in R und S aufgespalten, R und pk werden als Punkt dekodiert.
- S wird als Integer dekodiert.
- Ein Bitstring wird aus sha512$(R||pk||sha512(m))$ gebildet und zu einem Integer t konvertiert.
- Die Signature gilt als verifiziert, wenn die Gleichung $[2^3 * S]G = [2^3]R + [2^3 * t]PK$ zutrifft.

Zum besseren Verständnis des Ablaufs ist es hilfreich sich zunächst auf die Verwendeten Variablen in der Gleichung zur Verifizierung $[2^3 * S]G = [2^3]R + [2^3 * t]pk$ zu konzentrieren.

Die Variable $S = (r + sha512(R||pk|sha512(m)) * s)\%q$ enthält dabei alle Variablen die wichtig sind, rq, R, m und sq in Form von pk.
Es gilt zu beachten, dass Punkte wie R und PK über den selben Basepoint definiert sind, ihr nutzen besteht also darin die Variable mit der der Basepoint multipliziert wurde sicher zu übermitteln (siehe Abschnitt 10.1 ECDLP), hier also rq für R und $pk(int)$ für PK. Betrachtet man die rechte Seite der Gleichung fällt auf, dass die selben Variablen enthalten sind, rq und sq in Form des Punktes PK. Der Integer t, gebildet aus dem Bitstring

Bitstring HashData = sha512$(R||pk||sha512(m))$

ergänzt exakt den fehlenden Teil von S nachdem rq und sq bereits vorhanden sind. Zur Erinnerung: $S = (r + sha512(R||pk||sha512(m))s)\%q)$

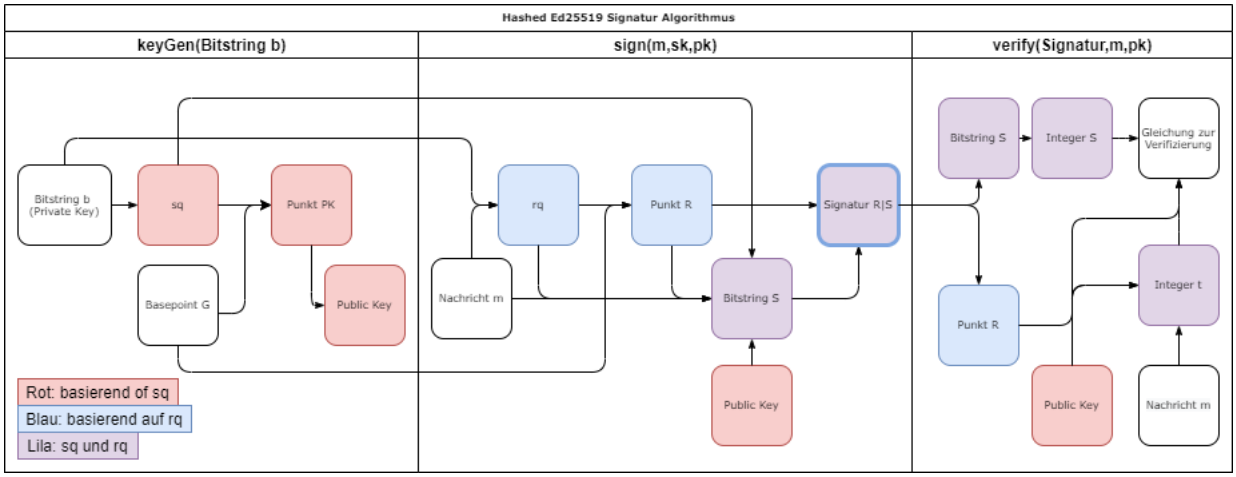

Figure 3: Diagramm zum Verlauf der Variablen

Anmerkung: Signatur R|S besteht aus den Werten Bitstring S (lila) und Punkt R (blaue), weshalb sie mit einem blauen Rand versehen wurde.

In Abbildung 3 soll ein Überblick zu den wichtigen Variablen im Algorithmus geschaffen werden. Der wichtigste Vorgang ist hier zunächst die Zusammensetzung von Bitstring S. Diese enthält alle anderen relevanten Variablen, sprich die Nachricht m und sq, rq sowie deren Punktformen PK und Punkt R, die durch multiplikation mit dem Basepoint G entstanden sind. In der verify() Funktion wird der Bitstring S zusammen mit Punkt R, PK und m übergeben. Diese Variablen werden von verify() benutzt, um den Interger S effektiv nachzubauen, stimmt der selbst gebaute Integer S mit dem Integer S aus der Signatur überein gilt die Signatur als verifiziert.

Stark vereinfacht passiert im Schema also Folgendes:

1. -KeyGen generiert sq
2. -Sign generiert rq aus sq und der Message m
3. -Sign generiert S aus sq, rq und m
4. -Verify erhält sq, rq, m und S
5. -Verify generiert ein eigenes S aus rq, sq und m
6. -Verify vergleicht sein S mit dem S aus Sign, stimmen beide überein wird true ausgegeben

Der Rest des Schemas dient zur sicheren Übertragung dieser Variablen und den Werten, die für ihre Generierung nötig waren.

Im Vergleich zum Grundlegenden Schnorr Schema fallen einige Unterschiede auf. Durch die Decode/Encode Funktionen für Integer und Punkte ist es möglich beide als Bitstring zu übertragen, weshalb an mehreren Stellen Integer vor der Übertragung mit dem Basepoint G Multipliziert und encodiert werden.

Statt lediglich die X-Koordinate eines Punktes zu benutzen, um einen Integer zu erhalten können ganze Punkte genutzt werden.

Außerdem fällt auf, dass die Sign Funktion keine Random Integer enthält, weshalb eine Signature mit Gleichem Input (m, sk, pk) im Gegensatz zum Schnorr Schema stets die gleiche Signature ausgeben wird. Statt des Random Integers in der Schnorr Signatur, der verhindert, dass aus der Signatur auf den Private Key geschlossen werden kann, wird die 2. Hälfte des Private Keys zusammen mit dem Hash der Message erneut gehasht um einen Integer r zu erhalten, der wiederum zur sicheren Übertragung mit dem Basepoint multipliziert wird.

Die Definition für den Hashed Ed25519 Signatur Algorithmus wurde der ursprünglichen Arbeit [1] entnommen.

Hash Funktion

Bei einer Hasfunktion wird ein Schlüssel eingegeben und auf einen Wert von bestimmter Länge abgebildet. Bei unserer Hashfunktion Sha512 wird ein String auf eine Bitstring der Länge 512 abgebildet.

Optimal ergibt jede valide Eingabe einen einzigartigen Bitstring, vom diesem Bitstring kann nicht auf den Eingabewert zurückgeschlossen werden. Dies kann praktisch unter anderem dazu genutzt werden zu beweisen, dass Zwei Parteien denselben Wert x kennen, ohne x öffentlich zu machen, in dem beide Parteien ihr x hashen und den entstehende Bitstring miteinander vergleichen.

8 Edwards Kurve

Eine Edwards Kurve ist eine Art elliptische Kurve [7] über die Gleichung $x^2 + y^2 = 1 + dx^2y^2$. Die Kurve wird als Körper Fp definiert, wobei p eine Primzahl ist, die den Moduloraum der Kurve bestimmt. Die Form der Kurve ist dabei von dem Faktor d abhängig. Dieser muss so gewählt werden, dass er nicht als Quadrat einer Zahl in p gebildet werden kann. Andernfalls ist die Addition nicht vollständig, das heißt, dass es Punkte auf der Kurve geben würde, die nicht mit dem bekannten Additionsverfahren addiert werden können.

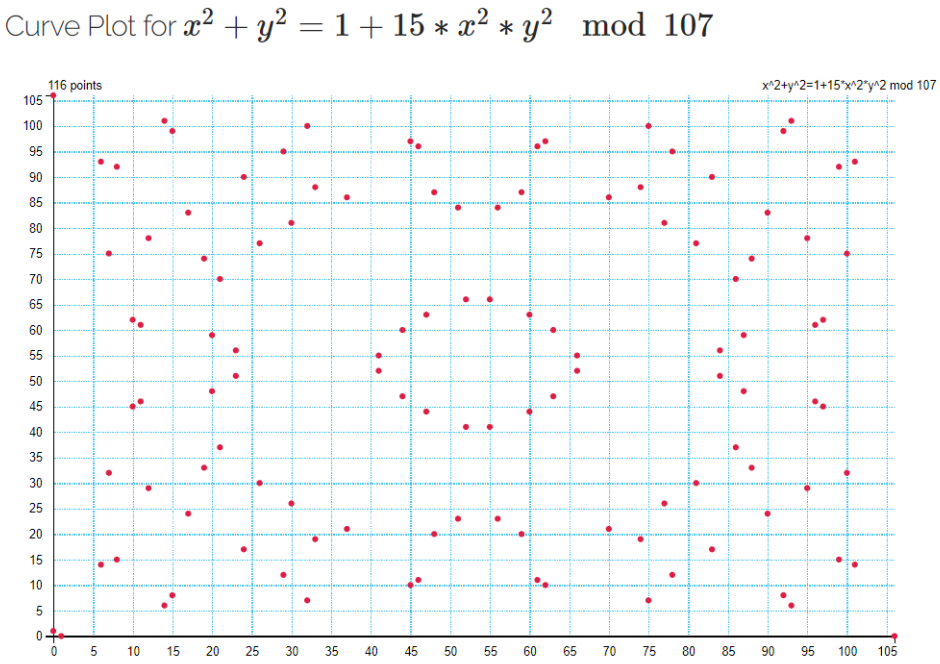

Figure 4: Edwardskurve mit p = 107 und d = 15

Abbildung erstellt auf graui.de/code/ffplot/ [5]

Beim Betrachten der Beispielkurve in der Abbildung fällt auf, dass die Kurve in der Mitte, also der Hälfte des Raumes über p, horizontal und vertikal symmetrisch ist. Die Ordnung der Kurve q ist definiert durch die Anzahl der Punkte auf der Kurve, auf Grund dieser Symmetrie gilt stets $q\%4 = 0$.

Bei der Umsetzung der Anwendung fiel auf, dass eine ungerade Zahl für q zu Fehlern führen würde, beispielsweise bei der Summenberechnung in $Frand$.

Wie bei elliptischen Kurven addiert man Zwei Punkte, in dem eine Gerade durch die Punkte gezogen wird, das Ergebnis ist der horizontal gegenüberliegende Punkt des Punktes, der mit auf dieser Tangente liegt.

Die Formel zur Addition lautet:

$$(x3|y3) = \left(\frac{x1 \cdot y2 + x2 \cdot y1}{1 + d \cdot x1 \cdot x2 \cdot y1 \cdot y2}, \frac{y1 \cdot y2 - x1 \cdot x2}{1 - d \cdot x1 \cdot x2 \cdot y1 \cdot y2}\right)$$

Das Besondere an der Edwards Kurve ist, dass die selbe Formel auch zur Verdopplung eines Punktes gilt, hierzu wird statt eine Geraden durch Zwei Punkte die Tangente des Punktes verwendet. Das neutrale Element liegt bei $(0|1)$.

Die erwähnte Unvollständigkeit bei einem d, welches ein Quadrat in p ist entsteht, in dem der Divisor-Teil der enthaltenen Division bei 0 liegen kann, was natürlich zu einem Fehler führt.

Innerhalb der Anwendung wird ein sich auf der Kurve befindlicher Basepoint als Generator festgelegt. Alle Punkte in der Anwendung entstehen durch die Multiplikation des Basepoints mit einem Integer. Die Multiplikation eines validen Basepoints mit den Integern $0, \ldots, (q-1)$ ergibt q verschiedene Punkte, es kann also vom Basepoint durch Multiplikation auf jeden anderen Punkt auf der Kurve gelangt werden.

Die Werte dieser Punkte sind abhängig vom Faktor d. Die Multiplikation mit allen Integern in p verschiedener valider Basepoints erzeugt also dieselbe Menge an Punkten, allerdings in anderer Reihenfolge, dieses Verhalten nennt man Permutation.

Anmerkung: In der Quelle zu Edwards Kurven [7] wirden spezifisch nur Twisted Edwards Curves erwähnt die einen Faktor a zur Gleichung hinzufügen. Bei Definition ist jede Edwards Kurve auch eine Twisted Edwards Kurve wenn dieser Faktor $a = 1$ ist, die in der Quelle genannten Formeln sind also auch hier zutreffend, da der Faktor a heraugekürzt werden konnte.

8.1 Umsetzung

In der Anwendung war es nicht nötig die Kurve als eigenes Objekt zu definieren. Von Testfunktionen abgesehen sind der Anwendung nur die Werte p, q, d und der Basepoint bekannt, allerdings nicht die Menge an validen Punkten auf der Kurve oder die Gleichung der Kurve selbst.

Zur Grundlegenden Umsetzung der für das Signaturverfahren benötigten Funktionen ist lediglich die Funktion $addPoints(x1, y1, x2, y2)$ zur Addition von 2 Punkten nötig. Es wird davon ausgegangen, dass der Basepoint ein valider Generator ist und da jeder Punkt der Anwendung aus Vielfachen dieses Punktes gebildet wird können auch diese als valide angenommen werden.

Eine umfassende Kenntnis der Kurve war also innerhalb der Anwendung nicht nötig.

Multiplikation wird in der Anwendung durch mehrfaches Addieren und Verdoppeln umgesetzt.

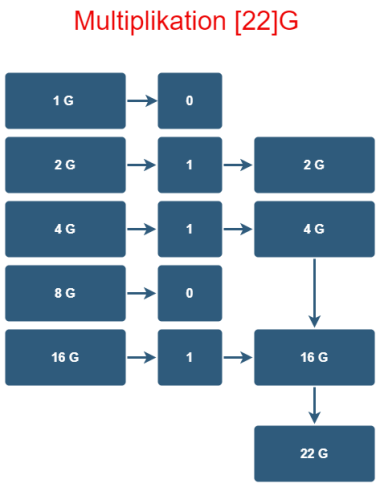

Figure 5: Algorithmus zur Multiplikation von G mit 22

In der Anwendung wird der Multiplikator zunächst in eine Binäre Zahl konvertiert, hier 22 => 10110. Der Basepoint wird verdoppelt mal der Länge dieser Zahl -1. Ist diese an der entsprechenden Stelle gleich 1 wird das Ergebnis der Verdopplung zum Output addiert.

Wie bereits beim Schnorr Algorithmus ist die Sicherheit durch den Diskrete Logarithmus Probleme bei elliptischen Kurven (ECDLP) gewährleistet.

9 Diskretes Logarithmus Problem (DSP)

Die Sicherheit der Signaturverfahren kann mit einer Form des diskreten Logarithmus Problem (DLP) begründet werden.

Das ursprüngliche DLP besagt folgendes:

Sei A ein Generator in einer zyklischen Gruppe der Ordnung n. Bei der Gleichung $A^x = B\%n$ würde man mit $x = dlog(A)B$ lösen, sprich x ist gleich dem diskreten Logarithmus von B zur Basis A.

Eine zyklische Gruppe F ist gegeben, wenn alle Elemente mit einem Generator A^xmit x als natürliche Zahl erzeugt werden können. Dies ist stets der Fall, wenn F über einen Modulo Raum mit Primzahl definiert wurde.

Das DLP besagt, dass x bei einer zyklischen Gruppe mit ausreichend groß gewähltem n praktisch nicht berechnet werden kann, da bisher kein Algorithmus bekannt ist, bei dem die Laufzeit nicht exponentiell mit der Größe von n steigt.

So können Beispielsweise 2 Parteien herausfinden, ob ein Integer den gleichen Wert hat, ohne diesen öffentlich zu machen, in dem sie ihn ein einer passenden zyklischen Gruppe mit gemeinsamem Generator für x einsetzen und B veröffentlichen.

Kryptografische verfahren wie die Elgamal Verschlüsselung oder der Diffie-Hellman Schlüsselaustausch machen sich dieses Prinzip zu nutzen.

9.1 DSP bei Elliptischen Kurven (ECDLP)

Der Hashed Ed25519 Signatur Algorithmus macht sich eine Art von DLP für elliptische Kurven (ECDLP) zu nutze. Bei der Skalarmultiplikation eines Generator-Punktes G mit einem Integer x $[xp]G = P$ kommt es zum gleichen Effekt. Wie bereits bekannt ist unsere elliptische Kurve wie eine zyklische Gruppe über eine Moduolozahl p definiert und die Multiplikation eines Generators führt wie bei der Exponierung in der zyklischen Gruppe zur Permutation der Punkte auf Fp. Deshalb entsteht bei der Multiplikation des Basepoints G auf einer Edwards Kurve mit einem ausreichend groß gewähltem q auch das diskrete Logarithmus Problem.

Bei $[xp]G = P$ mit ausreichend groß gewähltem p ist es also praktisch nicht möglich bei bekanntem G und P auf x zurückzuschließen.

Da die Addition von 2 Punkten auf einer elliptischen Kurve deutlich mehr Operationen benötigt als eine einfache Potenzierung im Moduloraum, ist die Berechnugn des diskreten Logarithmus natürlich auch komplexer. Dies hat zur Folge, dass Verfahren basierend auf dem ECDLP eine niedrigere Schlüssellänge im Vergleich zu DLP basierten Verfahren benötigen um als sicher zu gelten.

Part IV
JavaScript Anwendung

Die Anwendung wurde in Zwei Versionen umgesetzt. Eine Version, in der das Signatur Schema, daBit und MPC Funktionen von Grund auf selbst umgesetzt wurde und eine Version, bei der das eigene Signaturschema durch die Funktionalität der Library indutny/elliptic [3] zur Verfügung gestellt wird.

Aus zeitlichen Gründen benutzt die selbsterstellte Version eine Edwards Kurve mit $p = 107$, $q = 116$, $d = 15$ und $Basepoint = (8|15)$ statt der 25519 Kurve. Trotz dieser Abweichung wird es unvertrauten Lesern empfohlen, vorerst mit dieser Version zu testen, da die Konsolenausgaben und niedrigeren Zahlen den Ablauf deutlich nachvollziehbarer machen und auch manuell nachgerechnet werden können.

Anmerkung: Wenn nicht ausdrücklich anders erwähnt, handelt es sich in den folgenden Abschnitten stets um die selbst umgesetzte Anwendung ohne die indutny/elliptic Library.

In den folgenden Abschnitten werden verschiedene Funktionalitäten der JavaScript Anwendung beschrieben, die für die Hashed Ed25519 Signatur oder daBit benötigten werden, allerdings kein direkter Bestandteil dieser Algorithmen sind.

10 Modulo Division

Um Zahlen innerhalb des Moduloraumes zu dividieren muss das modulare Inverse des Divisors b gefunden werden und mit dem Dividenden a multipliziert werden. Hierzu wird ein erweiterter Euklidischer Algorithmus [4] benutzt, dabei ist es nötig, dass Divisor und Modulo miteinander teilerfremd sind, also ihr größter gemeinsamer Teiler bei 1 liegt.

Die Modulare Division wird im Code in den Funktionen $addPoints()$ und $getXfromY()$ benutzt, die beidem im Moduloraum p arbeiten. Da p bei Definition eine Primzahl ist, ist sie mit allen anderen Zahlen in ihrem Moduloraum außer 0 teilerfremd.

Im Algorithmus wird die zu invertierende Zahl b und die Modulozahl benutzt:

- -Die größere wird durch die kleinere Zahl geteilt (Quotient und Rest).
- -Der Divisor dieser Division wird durch den Rest geteilt.
- -Wiederholung bis der Rest 0 ergibt.
- -Der Divisor der letzten Gleichung ist der größte gemeinsame Teiler.
- -Zusätzlich wird mit jeder Division ein Wert $x = xi - 2 - xi - 1qi - 2$ berechnet, wobei $x1 = 0$ und $x2 = 1$ festgelegt ist.
- -Nach der letzten Division wird ein weitere x nach dieser Formel berechnet, dieses x ist das gesuchte modulare Inverse i.

Zur anschließende Überprüfung kann $i * b == 1$ berechnet werden.

11 Encode/Decode

Im Signatur Algorithmus wird an mehreren Stellen ein Integer oder Punkt zur Übertragung zu einem Bitstring kodiert und zu einem späteren Zeitpunkt wieder dekodiert, z.B. Besteht die Signatur aus den Enkodierungen des Punktes R und des Integers S.

Dieser Vorgang ist für Integer recht simpel. Der gewählte Integer wird mit der $encodeInt()$ Funktion zu einem binären String konvertiert und mit zusätzlichen Nullen auf die gewünschte Länge (hier 256) gebracht.

Dieser binäre String kann dann mit der $decodeInt()$ Funktion wieder dekodiert werden. Um unnötig große Rechnungen zu vermeiden, wird der entsprechende Modulowert (hier q oder p) in den Funktionsparametern mitübergeben.

Bei der Kodierung eines Punktes werden zunächst die X- und Y-Koordinaten als Integer auf die länger 256 kodiert. Die Kodierung des Punktes entsteht, indem das MSB der Y Kodierung durch das LSB der X Kodierung ersetzt wird.

Es fällt auf, dass lediglich 1 Bit der X Koordinate in der Kodierung enthalten ist. Die Dekodierung wird durch den Aufbau der Elliptischen Kurve ermöglicht. Es gibt stets Zwei Punkte auf jeder vorhandenen Y Koordinate auf Grund der

Symmetrie der Kurve, das LSB der X Kodierung wird hier genutzt, um zwischen diesen beiden Punkten zu unterscheiden.

Für die Dekodierung wird zunächst Y aus der Kodierung entnommen, der Hauptteil der Funktion besteht darin, X nachzuvollziehen. Es gilt die Formel $x^2 = (y^2 - 1)/(d * y^2 + 1)$ wobei die Modulo Division verwendet werden muss. Abhängig von p können verschiedene Formeln verwendet werden, um die Wurzel von x^2 zu ziehen.

Abschließend wird $x = p - x$ gesetzt, sollte das LSB von X nicht x modulo 2 entsprechen, x ist also positiv oder negativ im Moduloraum. Der Punkte (x|y) wird ausgegeben.

Es gilt zu beachten, dass die Encode/Decode Funktionen mit little-endian binären Strings arbeiten. Das bedeutet, dass anders als gewohnt die Wertigkeit der Bits ganz links mit der niedrigsten Wertigkeit (Least Significant Bit LSB) beginnt und das Bit mit der höchsten Wertigkeit (Most Significant Bit MSB) an der Stelle ganz rechts steht, die Zahl wird also andersherum gelesen.

Bei diesen Funktionen, besonders bei der Dekodierung von Punkten wurde sich stark an der Ed25519 Python Umsetzung von S. Josefsson und N. Moeller [2] orientiert.

12 Serverabfragen

Die Anwendung verwendet einen node.js Server mit der Library Socket.io. Socket.io dient zur erleichterten Kommunikation zwischen Server und Client. Ein Objekt io wird mit

```
var io = require('socket.io')(http);
```

auf dem Server definiert. Server und Client können mit

```
io.emit("keyword",JsonData);
```

einen Broadcast und mit

```
io.to(socket.id).emit("keyword", JsonData);
```

ein JSON Objekt an eine bestimmt Client ID versenden. Der String "keyword" wird vom Empfänger für einen Listener benutzt.

```
socket.on("keyword",(data)=>{
    //DO SOMETHING
});
```

Die Variable data ist hier das gesendete JSON Objekt, das in der ausführenden Funktion verarbeitet werden kann.

In der Anwendung stellt der Server einige Funktionen, wie z.B. die Hash Funktion zur Verfügung, die vom Client an verschiedenen Stellen genutzt wird. Hier sendet der Client selbst einen String "dest" mit im Objekt, unter dem der Server die Antwort schicken wird.

```
    socket.emit("hash", {
        value: bitString,
        dest: "keyGen"
    });
}
socket.on("keyGen", function (data) {
```

(Client)

```
io.to(socket.id).emit(data.dest, output);
```

(Server)

Part V
Funktionstests

Folgende Tests wurden im Browser Google Chrome ausgeführt. Die Zeiten wurden mit der performance.now() Funktion von JavaScript gemessen. In beiden Versionen wurde der Schlüssel nicht von daBit generiert, das heißt in der keyGen Funktion ist das Generieren eines 256 lange random Bitstrings beinhaltet, der andernfalls bereits in daBit generiert werden würde.

13 Library Version

In dieser Version wurden nur die Funktionen unterschiedlich zur eigenen Version getestet, sprich die Signaturfunktionen keyGen(), sign() und verify(). Die Funktionen wurden getestet, in dem 1000 zufällige Strings mit Länge 10, 1000 und 100.000 generiert wurden und mit ihnen der gesamte Signaturprozess durchgeführt wurde.

Es kann unter eigenen Umständen mit dem Test-Button am Ende der Seite getestet werden. Zur Änderung der String Länge und der Anzahl der Durchläufe können die Variablen *stringLength* und *runs* in der Funktion *test*() geändert werden. In der Konsole werden Arrays mit allen Laufzeiten und die Durchschnittswerte angezeigt. Da

Ergebnisse:

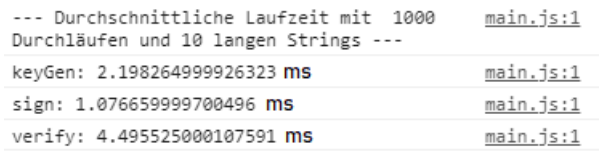

Figure 6: Ergebnis mit Stringlänge 10

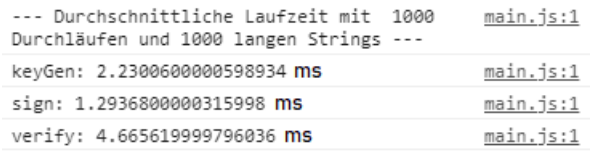

Figure 7: Ergebnis mit Stringlänge 1000

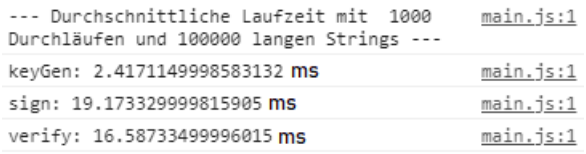

Figure 8: Ergebnis mit Stringlänge 100000

Beobachtung:

Die Laufzeit von *keyGen*() bleibt zwischen den Durchläufen konstant, da diese Funktion nicht mit dem generierten String arbeitet. Beim Betrachten der Werte im Array fällt außerdem auf, dass die ersten 2-5 Werte stehts um ein vielfaches über dem Durchschnitt liegen.

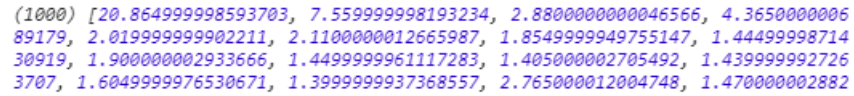

Figure 9: Ersten 16 Laufzeiten von keygen bei Stringlänge 1000

Bei allen Durchläufen war die ersten Laufzeiten bei weitem am höchsten. Dieser Effekt tritt nur auf, wenn die jeweilige Funktion seit dem Seitenaufruf nicht aufgerufen wurde, es kann also davon ausgegangen werden, dass er durch die Deklaration neue Variablen in der Funktion entsteht.

14 Eigene Version

Im Gegensatz zur Libraryversion verfügen die Clients über keine Interne Hash Funktion, stattdessen nutzen sie die entsprechende MPC Funktion um Strings auf Serverseite hashen zu lassen, die folgenden Ergebnissen sind also abhängig von der Übertragunszeit zwischen Client und Server. Um das asynchrone Verhalten der Funktionen zu berücksichtigen wurde eine rekursive Testmethode gewählt, bei dem eine Funktion erst nachdem sie durchgelaufen ist die nächste aufruft. Diese Testmethode wurde aus zeitlichen Gründen nur provisorisch eingefügt, weswegen sie leider nicht in der Abgabeversion vorhanden ist.

Ergebnisse:

Figure 10: Ergebnis mit Stringlänge 10

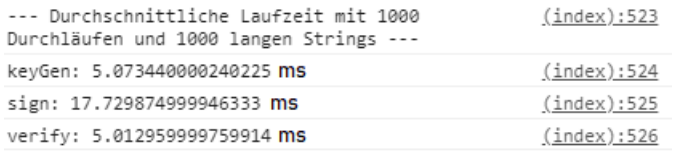

Figure 11: Ergebnis mit Stringlänge 1000

Figure 12: Ergebnis mit Stringlänge 100000

Beobachtung:

Im Vergleich zur Library Version fällt auf, dass die Laufzeiten bei sign() ähnlich skalieren, allerdings skaliert verify merkbar langsamer mit einer ~30-fachen Laufzeit zwischen Strings mit 1000 und 100000 Zeichen. Vor allem fällt aber auf, dass keyGen mit der Stringlänge skailiert, obwohl diese Funktion unabhängig davon sein sollte. Beim betrachten der Array fällt auf, dass die meisten Werte im erwarteten Bereich von ~4-5ms liegen, allerdings in unregelmäßigen Abständen Gruppen von Durchläufen mit ~900ms vorkommen.

In einer weiteren Reihe von Durchläufen wurde die Zeit zwischen Serveranfrage und Antwort von der Laufzeit abgezogen, dabei kam es nicht zu merkbar längeren Laufzeiten und die Ergebnisse blieben konstant.

Die folgenden Testergebnisse führen die Zeiten auf Clientseite mit und ohne der Zeit zwischen Serveranfrage und Antwort auf, sowie die Laufzeiten der Funktionen auf Serverseite.

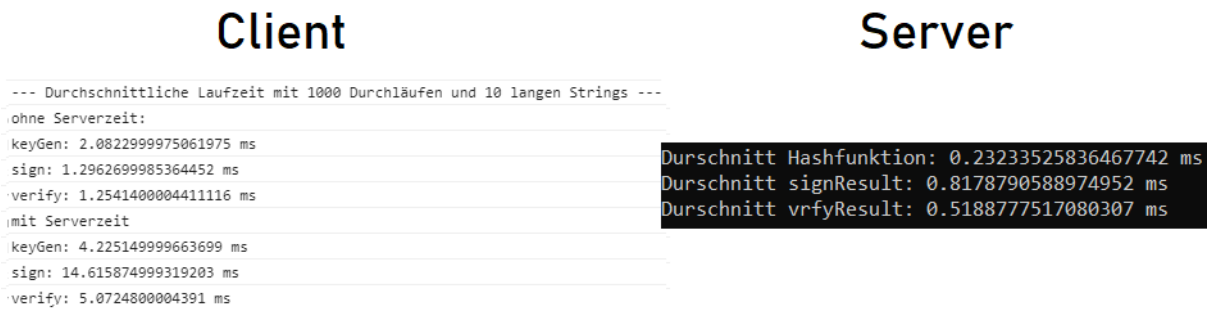

Figure 13: Ergebnis mit Stringlänge 10

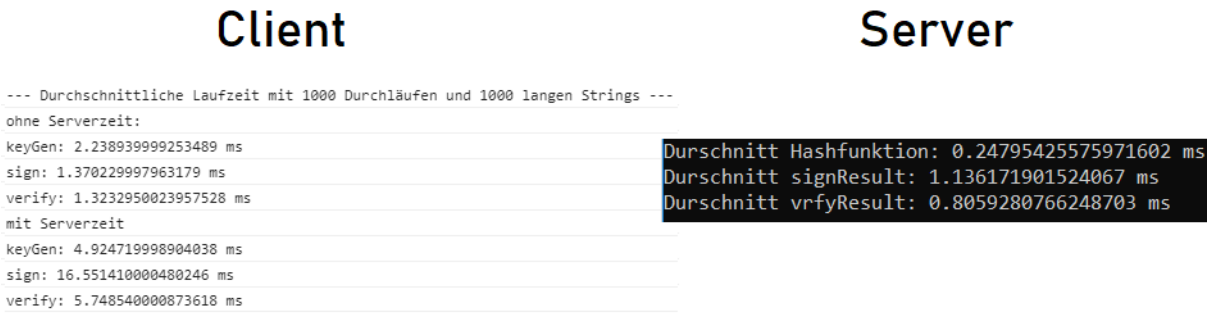

Figure 14: Ergebnis mit Stringlänge 1000

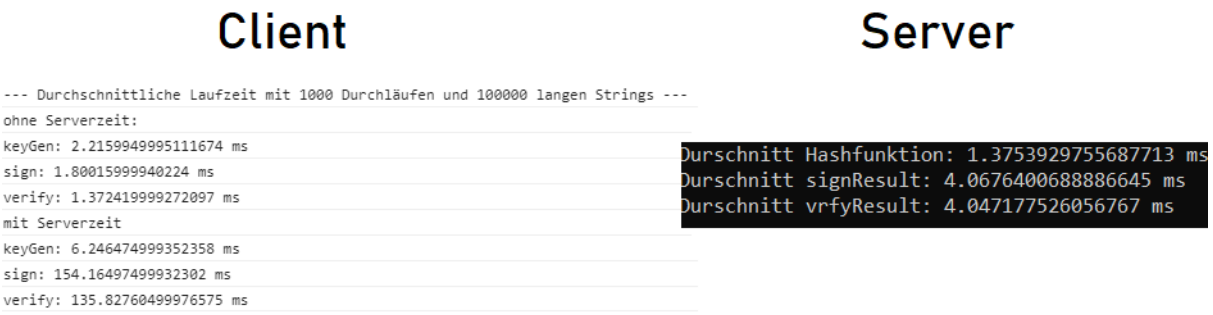

Figure 15: Ergebnis mit Stringlänge 100000

Die Signaturfunktionen benutzen folgende Serverfunktionen mit jedem Durchlauf:

keyGen: 1x Hash

sign: 1x signResult, 2x Hash

verify: 1x vrfyResult

Anmerkung: signResult und vrfyResult beinhalten selbst jeweils Zwei Aufrufe der Hash Funktion, wobei jede Funktion je ein Hash der Nachricht (also des Strings mit länger 10,1000 und 100000) enthält.

Die Hash Funktion in KeyGen erhält stets einen Bitstring der Länge 256, was erklärt, warum bei Stringlänge 100000 die Durchschnittslaufzeit von keyGen unter dem Durchschnitt der Hashfunktion liegt, obwohl keyGen eine Hashfunktion enthält.

Beobachtung: Die Laufzeit skalliert auf Clientseite nur gering mit der Stringlänge, die steigende Laufzeit auf Serverseite kann auf Grund des Aufbaus von signResult und vrfyResult damit begründet werden, dass die Hash Funktion bei bei großen Strings merkbar länger läuft, da der beinhaltete Hash zu Beginn der Funktionen passiert und hinterher mit dem Hash String gearbeitet wird, der stets 512 Bits lang ist.

Vor allem fällt aber der Unterschied auf Clientseite bei String der Länge 100000 auf. Die Laufzeiten bei sign und verify sind deutlich höher, wenn man die Serverzeiten mitberechnet, obwohl die separaten Funktionszeiten auf Client- und Serverseite im Vergleich gering bleiben. Der enorme Anstieg in der Laufzeit muss also durch die Übertragungszeit zwischen Server und Client kommen. Es wird also vermutet, dass die größte Schwachstelle der Laufzeitoptimierung dadurch verursacht wird, dass die Übertragungsgeschwindigkeit von JSON Objekten stark von deren größe abhängig ist, da dies der einzige Faktor zu sein scheint, der die Übertragungszeit deutlich beeinflusst.

Verwendete Technologien

Sprachen

- JavaScript
- HTML
- CSS

Libraries/Laufzeitumgebungen

- Node.js v12.16.1
- js-sha512 Library v0.8.0 https://github.com/emn178/js-sha512

- indutny elliptic Library v6.5.3 https://github.com/indutny/elliptic
- socket.io v2.3.0
- webpack v4.43.0
- npm 2.15.12

Programme

- IDE: Visual Studio Code 1.47
- Browser: Google Chrome Version 84.0.4147.89 (64-Bit)

Quellen

- [1] Thresholdizing HashEdDSA: MPC to the Rescue von Charlotte Bonte[1], NigelP. Smart[1,2] und Titouan Tanguy[1]
  [1]imec-COSIC, KU Leuven, Leuven, Belgium.
  [2]University of Bristol, Bristol, UK.
- [2] EdDSA and Ed25519 by Simon Josefsson and Niels Moeller https://tools.ietf.org/html/draft-josefsson-eddsa-ed25519-03#section-4.1
- [3] Indutny elliptic JavaScript Library https://github.com/indutny/elliptic
- [4] Extended Euclidean Algorithm http://www.doc.ic.ac.uk/~mrh/330tutor/ch03.html
- [5] Plot Curves over Finite Fields https://graui.de/code/ffplot/
- [6] Elliptic Curve Cryptography, Bundesamt für Sicherheit in der Informationtechnik
  https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_V-2-0_pdf.pdf?__blob=publicationFile&v=2
- [7] Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters
  https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186-draft.pdf
- [8] Low Cost Constant Round MPC Combining BMR and Oblivious Transfer von Carmit Hazay[1], Peter Scholl[2], Eduardo Soria-Vazquez[3]
  [1]Bar-Ilan University, Israel
  [2]Aarhus University, Denmark. Work done whilst at University of Bristol
  [3]University of Bristol