Compiler für Zero-Knowledge Proofs of Knowledge

Diplomarbeit

Jörn Heissler

Juli 2010

Der Inhalt der beiliegenden Compact Disc, welcher die vorliegende Arbeit und die entwickelte Software umfasst, steht auch im Internet unter `http://zkcompiler.tutnicht.de/` zum Abruf bereit.

Betreuer:

Dr. Klaus-Peter Kossakowski

Prof. Dr. Dieter Gollmann

Dipl.-Ing. Tobias Jeske

Technische Universität Hamburg-Harburg

Sicherheit in verteilten Anwendungen

`http://www.sva.tu-harburg.de/`

Harburger Schloßstraße 20

21079 Hamburg

Deutschland

Inhaltsverzeichnis

1. Einleitung

1.1. Thema der Diplomarbeit

In unserer vernetzten Welt, in der der Schutz der persönlichen Daten immer wichtiger wird, müssen neue technische Verfahren etabliert werden, die dem Gebot der Datensparsamkeit folgen und gleichzeitig die berechtigten Sicherheitsinteressen aller beteiligten Parteien wahren.

Derzeit verbreitete Authentisierungsverfahren haben eine gemeinsame Schwäche: Sie gewährleisten nicht die Anonymität des Benutzers und verletzen somit das Gebot der Datensparsamkeit. So kann beispielsweise nachvollzogen werden, wer eine Grenze passiert, wer bargeldlos bezahlt oder wer bestimmte Dienste nutzt.

Zero-Knowledge Proofs of Knowledge (Wissensbeweise ohne Weitergabe von Wissen) bieten einen vielversprechenden Ansatz, den scheinbaren Widerspruch zwischen Anonymitätsbedürfnis und Sicherheitsinteressen aufzulösen.

Der Themenbereich der Zero-Knowledge Proofs of Knowledge gliedert sich in zwei Bereiche: den theoretischen und den praktischen.

Die Theorie wird in verschiedenen Publikationen sehr unterschiedlich dargestellt und es fließen häufig neue Entwicklungen ein. Auch zeichnet sich die gesamte Thematik durch komplexe mathematische Verfahren aus.

Für die praktische Anwendung muss diese Theorie so formalisiert werden, dass sie auf existierenden Datenverarbeitungsanlagen zum Einsatz kommen kann.

Im Anschluss an die Formalisierung stellt sich das Problem der tatsächlichen Implementierung dieser Verfahren, um sie in der Praxis nutzen zu können.

1.2. Zielsetzung

Im Rahmen dieser Diplomarbeit sollen zwei Ziele verfolgt werden:

- Die Theorie der Zero-Knowledge Proofs of Knowledge soll in Hinblick auf eine praktische Umsetzung formalisiert werden.
- Die benötigten Werkzeuge, mit denen Zero-Knowledge-Beweise tatsächlich auf Rechnern eingesetzt werden können, sind zu programmieren. Die Entwicklung dieser Werkzeuge erfolgt als Gruppenprojekt in Zusammenarbeit mit Alexander Klein.

1.3. Motivation

1.3.1. Zero-Knowledge Proof of Knowledge

Mit einem Zero-Knowledge Proof of Knowledge kann man beweisen, dass man ein Geheimnis kennt und dass dieses bestimmte mathematische Eigenschaften besitzt, ohne das Geheimnis selbst preiszugeben.

Wiederholt auftretende Datenschutzskandale wie bei Mastercard ([Wel09]) zeigen die Relevanz von Datenschutz und Datensparsamkeit. Die wirklichen Ursachen dieser Skandale liegen aber nicht im Fehlverhalten einzelner Personen oder Unternehmen, sondern in den ungenügenden Konzepten.

Anonymität sollte immer dann gewahrt bleiben, wenn eine persönliche Identifizierung nicht absolut erforderlich ist. Mit Zero-Knowledge-Beweisen lässt sich ein Maximum an Datensparsamkeit bei gleichzeitiger Sicherheit erreichen.

Zero-Knowledge-Beweise basieren auf Verfahren, die mit der Kryptographie verwandt sind. Die Verwendung von ungelösten mathematischen Problemen, wie zum Beispiel dem diskreten Logarithmus-Problem, bietet die notwendige Sicherheit.

1.3.2. Anwendungsbeispiele

Zero-Knowledge-Beweise sind vielfältig einsetzbar:

- **Zugangskontrollen** (Grenzkontrolle, Mitgliederbereiche on- und offline, etc.)

  Bei einem Grenzübertritt oder der Nutzung eines Mitgliederbereichs ist die Identität der Person in der Regel nicht relevant. Es reicht zu wissen, dass die Person das Nutzungsprivileg hat.

  Es muss möglich sein, entsprechende Mitgliedschaften oder das Recht auf Anonymität, beispielsweise wenn ein Haftbefehl vorliegt, zu entziehen.

  Bei derzeitigen Lösungen wird entweder die Anonymität in den Vordergrund gerückt (z. B. Grenzen innerhalb der Schengen-Zone) und Sicherheit nahezu aufgegeben oder die Anonymität aufgehoben, um größtmögliche Sicherheit zu gewährleisten (Grenze zu Nordkorea).

- **Altersverifikation**

  Eine Altersverifikation soll in der Regel sicherstellen, dass nur volljährige Menschen einen bestimmten Dienst (Zigarettenautomaten, Internetangebote für Erwachsene, etc.) nutzen können.

  Derzeit ist häufig eine Preisgabe der persönlichen Daten notwendig: Die Verifikation geschieht mittels der Personalausweisnummer oder des Führerscheins.

- **Gruppensignaturen**

  Mit einer Gruppensignatur kann ein Mitglied einer Gruppe stellvertretend für diese Gruppe eine Signatur aussstellen. Außenstehende können so überprüfen, dass ein Mitglied der Gruppe unterschrieben hat. Sie haben jedoch keine Möglichkeit, die Identität des Signierers zu ermitteln.

  Unter gewissen Umständen soll es aber möglich sein, dass eine fest definierte interne Partei die Identität aufdecken kann. Ebenso muss es möglich sein, neue Mitglieder in die Gruppe aufzunehmen und andere aus der Gruppe zu entfernen. (Siehe [ACJT00] für ein Beispielprotokoll)

- **Gutscheine / Elektronisches Geld**

  Wie auch die Äquivalente aus Papier sollen Gutscheine und elektronisches Geld zum einen anonym benutzbar sein. Zum anderen muss auch sichergestellt sein, dass sie nicht doppelt eingelöst bzw. ausgegeben werden können. Ein Problem hierbei ist, dass sich elektronische Daten problemlos kopieren lassen.

  Bestehende Lösungen wie Geld- und EC-Karten sind nicht anonym, da Händler die Möglichkeit besitzen, mehrere Zahlungen miteinander zu verknüpfen und mit einer Person in Verbindung zu bringen. Desweiteren haben Banken Einblick darin, welcher Kunde wann wo eingekauft hat und können anhand des Geldbetrags auch auf die erworbene Ware oder Dienstleistung schließen und somit das Konsumverhalten analysieren. (Siehe [CHL05] für eine Lösung.)

1.3.3. Bestehende Lösungen

Die Europäische Kommission fördert mit dem Siebten Rahmenprogramm (RP7) unter anderem das Projekt „Computer Aided Cryptography Engineering“ (CACE) [Pro].

Eine Arbeitsgruppe innerhalb des CACE-Projekts entwickelt den „Zero Knowledge Proof of Knowledge Compiler“, welcher vergleichbare Ziele wie unsere Lösung hat, aber anderen Ansätzen folgt.

Vom CACE-Compiler steht eine Testversion zur Verfügung, zu der allerdings kein Quellcode verfügbar ist und die nur über das Internet benutzt werden kann.

1.4. Aufbau der Arbeit

Im 2. Kapitel werden die mathematischen Konzepte erläutert, die Grundlage dieser Arbeit sind. Diese sind für das Verständnis von dem darauffolgenden Kapitel unverzichtbar.

In Kapitel 3 wird die Theorie der Zero-Knowledge-Beweise ausführlich dargestellt und an ausgewählten Protokollen und Beispielen näher erläutert.

Das 4. Kapitel enthält eine Beschreibung der entwickelten Werkzeuge und der hierfür benötigten Eingabesprache.

Ein abschließendes Fazit befindet sich in Kapitel 5.

Im Anhang findet sich eine Auflistung der verwendeten Abkürzungen und Konventionen.

2. Mathematische Grundlagen

2.1. Gruppen

2.1.1. Definition

Eine *Gruppe* ist definiert als ein Tupel $(G, \circ)$ mit einer nichtleeren Menge G und einer binären Verknüpfung $\circ : G \times G \to G$, für die folgende Axiome erfüllt sind:

- **Assoziativität:** $\forall a,b,c \in G : (a \circ b) \circ c = a \circ (b \circ c) = a \circ b \circ c$
  Ob man erst a mit b verknüpft und das Ergebnis dann mit c oder ob man a mit dem Ergebnis der Verknüpfung von b und c verknüpft, macht keinen Unterschied. Zur besseren Lesbarkeit kann auf die Klammern verzichtet werden.

- **Neutrales Element:** $\exists e \in G : \forall a \in G : e \circ a = a \circ e = a$
  Es gibt ein *neutrales Element* in G. Wird dieses mit einem beliebigen Element a verknüpft, ergibt sich wieder a. Wenn die Verknüpfung die Addition ist, heißt dieses Element 0; bei der Multiplikation heißt es 1.

- **Inverse Elemente:** $\forall a \in G : \exists a^{-1} \in G : a \circ a^{-1} = a^{-1} \circ a = e$
  Zu jedem Element a in G gibt es ein *inverses Element* a^{-1}. Die Verknüpfung der beiden Elemente ergibt das neutrale Element e.

Eine Gruppe heißt *abelsche Gruppe*, wenn zusätzlich noch das folgende Axiom erfüllt ist:

- **Kommutativität**: $\forall a,b \in G : a \circ b = b \circ a$
  Die Operanden der Verknüpfung können vertauscht werden, ohne dass sich das Ergebnis ändert.

Alle in dieser Arbeit beschriebenen Gruppen sind abelsch. Einige der noch vorzustellenden Verfahren setzen die Verwendung abelscher Gruppen voraus.

Die Verknüpfung zweier Elemente aus einer Gruppe ist stets ein Element aus derselben Gruppe, also $\forall a,b \in G : a \circ b \in G$. Diese Eigenschaft heißt *Abgeschlossenheit* (bezüglich der Verknüpfung $\circ$) und folgt unmittelbar aus der Definition der Verknüpfung.

Satz 1: Für $a,x,y \in G$ mit $x \circ a = y \circ a$ gilt $x = y$.
Beweis: $x = x \circ e = x \circ (a \circ a^{-1}) = (x \circ a) \circ a^{-1} = (y \circ a) \circ a^{-1} = y \circ (a \circ a^{-1}) = y \circ e = y$
Analog lässt sich zeigen, dass aus $a \circ x = a \circ y$ folgt: $x = y$.

Mit diesem Satz ergibt sich, dass das neutrale Element und inverse Elemente eindeutig sind:

- Seien $a, e_0, e_1 \in G$ wobei e_0 ein neutrales Element ist und $e_1 \circ a = a$ gilt. Aus $e_0 \circ a = a$ folgt damit $e_0 \circ a = e_1 \circ a$. Mit dem Satz folgt $e_0 = e_1$. Es muss also genau ein neutrales Element geben.

- Seien $a, a_0^{-1}, a_1^{-1} \in G$, wobei $a_0^{-1} \circ a = a_1^{-1} \circ a = e$ gilt. Mit dem Satz folgt $a_0^{-1} = a_1^{-1}$. Zu jedem Element gibt es also genau ein inverses Element.

Satz 2: Für $a, b \in G$ gilt: $(a \circ b)^{-1} = b^{-1} \circ a^{-1}$.
Beweis: $(a \circ b)^{-1} = (a \circ b)^{-1} \circ e = (a \circ b)^{-1} \circ (a \circ e \circ a^{-1}) = (a \circ b)^{-1} \circ (a \circ b \circ b^{-1} \circ a^{-1})$
$= (a \circ b)^{-1} \circ (a \circ b) \circ (b^{-1} \circ a^{-1}) = e \circ (b^{-1} \circ a^{-1}) = b^{-1} \circ a^{-1}$

Wenn die Verknüpfungsfunktion einer Gruppe als bekannt vorausgesetzt werden kann oder sie nicht relevant ist, kann statt $(G, \circ)$ auch nur G geschrieben werden.

2.1.2. Ordnung von Gruppen

Die *Ordnung einer Gruppe* $(G, \circ)$ ist die Anzahl der Elemente in der Menge G, also $|G|$. Wenn die Ordnung einer Gruppe endlich ist, so wird diese Gruppe als *endliche Gruppe* bezeichnet.

2.1.3. Potenzen

Für $a, b \in G, i \in \mathbb{Z}$ bezeichnet $b = a^i$ die i-te Potenz von a, was der i-fachen Verknüpfung von a mit dem neutralen Element e entspricht: $b = e \underbrace{\circ a \circ a \cdots \circ a}_{i\text{-mal}}$

Somit gelten

- $a^0 = e$,
- $a^1 = e \circ a = a$,
- $a^2 = e \circ a \circ a = a \circ a$

und so weiter.

Diese Darstellung ergibt jedoch nur dann Sinn, wenn i nicht negativ ist. Für $i < 0$ gilt $a^i = (a^{-1})^{-i}$. In diesem Fall wird also das Inverse von a $(-i)$-mal mit e verknüpft.

Die Beziehung $a^i \circ a^j = a^{i+j}$ mit $i, j \in \mathbb{Z}$ gilt ebenfalls in allen Gruppen.

Die Berechnung von Potenzen ist mit Verfahren wie „Quadrieren und Multiplizieren“ mit wenig Aufwand verbunden. Dieser verhält sich linear zur Länge des Exponenten.

2.1.4. Untergruppen

Eine Gruppe $(U, \circ)$ heißt *Untergruppe* von einer Gruppe $(G, \circ)$, wenn U eine unechte Teilmenge von G ist, die Verknüpfungsfunktion $\circ$ in beiden Gruppen identisch ist und die Untergruppe bezüglich der Verknüpfung abgeschlossen ist.

Bestimmte Untergruppen haben für kryptographische Anwendungen sehr nützliche Eigenschaften, weshalb diese häufig verwendet werden.

Von besonderem Interesse ist die Untergruppe der quadratischen Reste. Diese beinhaltet genau die Elemente, die sich als Quadrat eines Elements darstellen lassen: $QR(G) = \{a^2 \mid a \in G\}$.

2.1.5. Zyklische Gruppen

Eine Gruppe $(G, \circ)$ heißt genau dann *zyklische Gruppe*, wenn es ein Element $g \in G$, auch Generator von G genannt, gibt, für das gilt: $\forall a \in G : \exists i \in \mathbb{Z} : g^i = a$. G wird von g erzeugt: $G = \{g^i \mid i \in \mathbb{Z}\} = \langle g \rangle$. Jedes Element in G lässt sich also als eine Potenz von g darstellen. Der Exponent i darf auch negativ sein.

Satz 3: Alle zyklischen Gruppen sind kommutativ.
Beweis: Seien $a = g^i, b = g^j$ mit $i, j \in \mathbb{Z}$. Es folgt: $a \circ b = g^i \circ g^j = g^{i+j} = g^{j+i} = g^j \circ g^i = b \circ a$.

Jedes Element $a \in G$ erzeugt eine Untergruppe $U \subseteq G$. Nach dem Satz von Lagrange ist die Ordnung von U ein Teiler der Ordnung von G ([MOV01, 2.171]). Die *Ordnung eines Elements* $a \in G$ wird mit $|a|$ bezeichnet und entspricht der Ordnung der erzeugten Untergruppe $\langle a \rangle$.

Es gibt zyklische Gruppen, die nicht endlich sind. Zum Beispiel ist 1 ein Generator der unendlichen Gruppe der ganzen Zahlen, $(\mathbb{Z}, +)$. Ebenso existieren endliche Gruppen, die nicht zyklisch sind: $\mathbb{Z}_8^*$ hat keinen Generator und ist somit nicht zyklisch.

Wenn G eine endliche, zyklische Gruppe ist, so gilt $a^q = e$ für alle $a \in G$ und $q = |G|$. Die Exponenten von Potenzen dürfen für solche Gruppen statt aus der Menge $\mathbb{Z}$ auch aus der Menge $\mathbb{Z}_q$ stammen, wenn die Restklassenelemente als ganze Zahlen interpretiert werden. Ist die Ordnung der Gruppe G unbekannt, beispielsweise wenn G eine RSA-Gruppe ist, muss $\mathbb{Z}$ für Exponenten verwendet werden.

2.1.6. Diskrete Logarithmen

Die Umkehrfunktion der Potenz ist der Logarithmus. Von Interesse ist hier der *diskrete Logarithmus*, der auf endlichen, zyklischen Gruppen definiert ist.

Seien $g, x \in G$ und g ein Generator von G. Dann heißt $i = \log_g x,\ i \in \mathbb{Z}$ genau dann ein diskreter Logarithmus von x zur Basis g, wenn gilt: $g^i = x$.

Zu festen g, x existieren unendlich viele diskrete Logarithmen i: Sei $q = |G|$ und $i = \log_g x$ ein diskreter Logarithmus von x. Dann sind auch $i+q,\ i+2q,\ \ldots$ diskrete Logarithmen von x, weil $g^q = e$ und somit $g^{i+q} = g^i \circ g^q = x \circ e = x$ gelten.

Im Gegensatz zur Berechnung von Potenzen kann die Berechnung von diskreten Logarithmen sehr schwierig sein.

2.1.7. Verwendete Gruppen

In dieser Arbeit werden folgende Gruppen verwendet:

- $(\mathbb{Z},+)$ bzw. $\mathbb{Z}$ ist die unendliche, zyklische Gruppe der ganzen Zahlen mit Addition als Verknüpfung. Das neutrale Element ist $e=0$ und inverse Elemente werden als $-a$ geschrieben. Die Generatoren dieser Gruppe sind 1 und -1. Potenzen in dieser Gruppe entsprechen der Multiplikation.

- $(\mathbb{Z}_n,+)$ bzw. $\mathbb{Z}_n$ ist die Familie der endlichen, zyklischen Gruppen der Restklassen modulo $n \in \mathbb{N}$ mit modularer Addition als Verknüpfung. Das neutrale Element ist $[0]$. Das inverse Element von $a=[k]$ ist $a^{-1}=[n-k]$. Die Ordnung von $\mathbb{Z}_n$ ist n.

  Generatoren von $\mathbb{Z}_n$ sind alle zu n teilerfremden Elemente. Die Potenz entspricht der modularen Multiplikation. In diesen Gruppen lassen sich diskrete Logarithmen einfach mit dem erweiterten euklidischen Algorithmus berechnen.

  Beispiel $\mathbb{Z}_{10}$:

  - $\mathbb{Z}_{10}=\{[0],[1],[2],\ldots,[8],[9]\}$
  - $[4]+[3]=[7]$ und $[8]+[6]=[4]$
  - $[2]^{-1}=[10-2]=[8]$
  - Die Generatoren sind $[1],[3],[7],[9]$. Zum Beispiel gilt $[5]^0=[0],\ [5]^1=[5],\ [5]^2=0,\ldots$ und ist damit kein Generator. Hingegen ist $[7]^{0\ldots9}=\{[0],[7],[4],[1],[8],[5],[2],[9],[6],[3]\}$ ein Generator.
  - Der diskrete Logarithmus $i=\log_{[3]}[4]$ lässt sich mit dem erweiterten euklidischen Algorithmus berechnen. Es gilt: $3\cdot7-10\cdot2=1$, also $3\cdot7\cdot4-10\cdot2\cdot4=4$. Eine Lösung ist somit $i=7\cdot4=28$. Weitere Lösungen sind $38,18,8,-2,\ldots$.

- $(\mathbb{Z}_n,*)$ bzw. $\mathbb{Z}_n^*$ ist die Familie der endlichen Gruppen der Restklassen modulo $n\in\mathbb{N}$ mit modularer Multiplikation als Verknüpfung. In der Menge $\mathbb{Z}_n^*$ sind alle zu n teilerfremden Restklassen modulo n enthalten. Das neutrale Element ist $[1]$, inverse Elemente werden mit Hilfe des erweiterten euklidischen Algorithmus berechnet. Die Ordnung von $\mathbb{Z}_n^*$ ist $|\mathbb{Z}_n^*|=\varphi(n)$, wobei φ die eulersche φ-Funktion ist (Siehe [Big85, Theorem 4.5.1]).

  $\mathbb{Z}_n^*$ ist genau dann zyklisch (hat also Generatoren), wenn $n=2,4,p^k$ oder $2p^k$ gilt mit $k\in\mathbb{N},k\geq1$ und einer Primzahl $p\geq3$ [MOV01, 2.132].

  Von besonderem Interesse sind zwei Spezialfälle:

1. Der Modulus n ist eine *sichere Primzahl*, das heißt: n und $\frac{n-1}{2}$ sind Primzahlen. Betrachtet wird die Untergruppe der quadratischen Reste modulo n, $U = QR(\mathbb{Z}_n^*)$, die von jedem quadratischen Rest bis auf $[1]$ generiert wird. Die Ordnung der Untergruppe ist $|U| = \frac{n-1}{2}$, also eine Primzahl.

2. Der Modulus n ist das Produkt zweier sicherer Primzahlen p und q. Betrachtet wird auch hier die Untergruppe U der quadratischen Reste modulo n. Die Ordnung dieser Untergruppe ist $|U| = \frac{\varphi(n)}{4} = \frac{(p-1)(q-1)}{4}$, also das Produkt zweier Primzahlen. Die Anzahl der Generatoren ist $\varphi(|U|) = (\frac{p-1}{2} - 1)(\frac{q-1}{2} - 1)$; bei großen Primzahlen p, q ist fast jedes Element auch gleichzeitig ein Generator.

Für $\mathbb{Z}_n^*$ ist kein allgemeines Verfahren bekannt, mit dem sich diskrete Logarithmen effizient berechnen ließen.

Beispiel $\mathbb{Z}_{20}^*$:

- $\mathbb{Z}_{20}^* = \{[1], [3], [7], [9], [11], [13], [17], [19]\}$
- $[3] * [3] = [9]$ und $[9] * [13] = [17]$
- $[7]^{-1} = [3]$ (Weil $7 * 3 = 1 + 20$)
- $\mathbb{Z}_{20}^*$ hat keine Generatoren und ist somit nicht zyklisch.

Beispiel quadratische Reste in $\mathbb{Z}_{77}^*$:

- $77 = 7 \cdot 11$, was beides sichere Primzahlen sind.
- $|QR(\mathbb{Z}_{77}^*)| = 15$
- Die Generatoren von $QR(\mathbb{Z}_{77}^*)$ sind $[4], [9], [16], [25], [37], [53], [58], [60]$.

2.2. Gruppenhomomorphismen

Eine Abbildung $\phi : A \to B$ von einer Gruppe $(A, \circ)$ auf eine andere Gruppe $(B, \diamond)$ heißt ein *Gruppenhomomorphismus* genau dann, wenn gilt:

$$\forall a_0, a_1 \in A : \phi(a_0 \circ a_1) = \phi(a_0) \diamond \phi(a_1)$$

Die Abbildung heißt außerdem:

- Gruppenepimorphismus, wenn sie surjektiv ist, also auf jedes Element der Bildmenge mindestens einmal abgebildet wird.
- Gruppenmonomorphismus, wenn sie injektiv ist, also auf dasselbe Element der Bildmenge nicht öfter als einmal abgebildet wird.

- Gruppenisomorphismus, wenn sie bijektiv ist, also auf alle Elemente der Bildmenge genau einmal abgebildet wird.

Eigenschaften von Gruppenhomomorphismen:

- Das neutrale Element des Urbilds bildet auf das neutrale Element des Bilds ab: Sei e_A das neutrale Element der Gruppe $(A, \circ)$ und e_B das neutrale Element der Gruppe $(B, \diamond)$. Sei $a \in A$. Dann gilt: $\phi(a) = \phi(a \circ e_A) = \phi(a) \diamond \phi(e_A)$, es muss also $\phi(e_A) = e_B$ sein.

- Inverse Elemente im Urbild bilden auf die entsprechenden inversen Elemente im Bild ab: Seien e_A und e_B die neutralen Elemente der Gruppen $(A, \circ)$ und $(B, \diamond)$. Seien $a, a^{-1} \in A$. Dann gilt: $\phi(a) \diamond \phi(a^{-1}) = \phi(a \circ a^{-1}) = \phi(e_A) = e_B = \phi(a) \diamond \phi(a)^{-1}$, folglich muss $\phi(a^{-1}) = \phi(a)^{-1}$ gelten.

- Die Potenzierung ergibt vor und nach der Abbildung dasselbe: $\forall a \in A,\ i \in \mathbb{Z} :\ \phi(a^i) = \phi(a)^i$. Dies wird deutlich, wenn man die Potenz ausschreibt und die Elemente dann einzeln abbildet: $\phi(a^3) = \phi(a \circ (a \circ a)) = \phi(a) \diamond \phi(a \circ a) = \phi(a) \diamond \phi(a) \diamond \phi(a) = \phi(a)^3$.

Während die Berechnung des Bilds bei den meisten Homomorphismen einfach ist, kann die Berechnung des Urbilds aus einem Bild sehr schwierig sein.

2.3. Zu Grunde liegende mathematische Probleme

2.3.1. Diskretes Logarithmus-Problem

In einer Gruppe $(G, \circ)$ sind die Verknüpfung $\circ$ und Potenzen in der Regel mit geringem Aufwand berechenbar. Potenzen mit Exponent i liegen, gemessen in Verknüpfungen, in der Komplexitätsklasse $O(2 \cdot \log(i))$.

Um diskrete Logarithmen zu bestimmen ist hingegen (für klassische Computer) kein allgemeines, effizientes Verfahren bekannt. Es wird vermutet, dass so ein Verfahren nicht existiert.

Es existieren jedoch Verfahren, um in bestimmten Fällen diskrete Logarithmen berechnen zu können. Wenn beispielsweise die Ordnung der Gruppe keinen großen Primfaktor hat, lassen sich die diskreten Logarithmen mit dem Pohlig-Hellman-Verfahren (siehe [MOV01, Kapitel 3.6.4]) effizient bestimmen. Durch das Verfahren reduziert sich die Schwierigkeit einen diskreten Logarithmus zu berechnen darauf, einen diskreten Logarithmus in einer Gruppe zu berechnen, deren Ordnung der größte Primfaktor der Ordnung der eigentlichen Gruppe ist.

Deshalb ist es notwendig, Gruppen so zu wählen, dass ihre Ordnung mindestens einen großen Primfaktor hat. Im Fall von $\mathbb{Z}_n^*$ bietet sich an, für n das Produkt aus ein oder zwei großen, sicheren Primzahlen zu wählen, da dann die Ordnung der Gruppe sehr große Primfaktoren hat.

2.3.2. Faktorisierungsproblem

Gegeben sei eine Zahl $n \in \mathbb{N}$. Es ist kein Verfahren bekannt, um (auf klassischen Computern) effizient die Primfaktorzerlegung $n = \prod_i p_i^{e_i},\ e_i \in \mathbb{N},\ p_i$ prim zu bestimmen.

Spezialfall RSA-Modulus

Sei $n = p \cdot q$ das Produkt zweier zu bestimmender Primzahlen, $e \in \mathbb{Z}^*_{\varphi(n)}$ der öffentliche RSA-Exponent und $d \in \mathbb{Z}^*_{\varphi(n)}$ der geheime Exponent. Es seien $a_i, c_i \in \mathbb{Z}^*_n$ mit $c_i = a_i^e$ gegeben. Gesucht sind die $d_i \in \mathbb{Z}^*_{\varphi(n)}$ mit $a_i = c_i^{d_i}$. Wenn das diskrete Logarithmus-Problem effizient lösbar ist, findet man solche d_i. Mit hoher Wahrscheinlichkeit ist $\prod_i d_i$ dann ein Vielfaches des geheimen RSA-Exponenten d.

Es gibt einen probabilistischen Algorithmus, der effizient aus Vielfachen von $d \cdot e$ die Primfaktorzerlegung p, q bestimmt. Dieser Algorithmus lässt sich so erweitern, dass er auch eine allgemeine Primfaktorzerlegung $n = \prod_{i=0}^{k-1} p_i^{e_i}$ bestimmen kann.

Hieraus folgt, dass das Faktorisierungsproblem nicht schwerer lösbar ist, als das diskrete Logarithmus-Problem.

2.3.3. RSA-Problem

Gegeben seien eine natürliche Zahl $n = p \cdot q$, die das Produkt von zwei verschiedenen großen Primzahlen ist, eine Zahl $e \in \mathbb{Z}^*_{\varphi(n)}$ und eine Zahl $c \in \mathbb{Z}^*_n$. Es ist kein Verfahren bekannt, um ein $m \in \mathbb{Z}^*_n$ mit $m^e = c$ effizient zu finden. Ist jedoch die Faktorisierung von n bekannt, so kann die Gruppenordnung $\varphi(n)$ bestimmt und das Inverse von e berechnet werden. Dann gilt $m = c^{e^{-1}}$. Dieses Problem kann auch als Wurzel-Problem bezeichnet werden, da die e-te Wurzel aus c zu ziehen ist.

Es wird vermutet, dass aus einer Lösung des RSA-Problems auch eine Lösung des Faktorisierungsproblems folgen würde, die Probleme also äquivalent sind.

2.3.4. Praktische Nutzung der Probleme

Um die Eigenschaften der schweren Probleme zu nutzen und zu verallgemeinern, werden sie in Form von Gruppenhomomorphismen dargestellt, deren Urbilder schwierig berechenbar sind.

Sei A eine Gruppe, in der diskrete Logarithmen nur schwer zu berechnen sind und sei $a,\ \langle a \rangle = A$ ein Generator der Gruppe. Ist die Ordnung von A nicht bekannt, beispielsweise wenn $A \subseteq \mathbb{Z}^*_n$ gilt mit unbekannter Faktorisierung von n, definieren wir $\phi_0 : \mathbb{Z} \to A,\ i \mapsto a^i$, ansonsten $\phi_1 : \mathbb{Z}_{|A|} \to A,\ i \mapsto a^i$.

Wenn B eine Gruppe ist, in der es schwierig ist, Wurzeln zu ziehen, definieren wir $\phi_2 : B \to B,\ b \mapsto b^e$ mit einem festen Exponenten e.

Es können auch mehrere Gruppenhomomorphismen miteinander verknüpft werden. Seien $\phi_A : A \to C$ und $\phi_B : B \to C$ zwei Homomorphismen, in denen Urbilder praktisch nicht berechenbar sind. Dann kann $\phi : A \times B \to C,\ (a,b) \mapsto \phi_A(a) \circ \phi_B(b)$ definiert werden.

Die Schwierigkeit, Urbilder zu berechnen, basiert in diesen Gruppenhomomorphismen auf den zugrundeliegenden mathematischen Problemen. Hiermit sind die mathematischen Grundlagen geschaffen, auf denen die Sicherheit von Zero-Knowledge-Beweisen basiert. Im folgenden Kapitel wird von der Darstellung als Gruppenhomomorphismus noch weiter abstrahiert, um schwierige Probleme in der Mathematik schließlich praktisch nutzen zu können.

3. Zero-Knowledge Beweise

3.1. Einleitung

Zero-Knowledge Proofs of Knowledge (Wissensbeweise ohne Weitergabe von Wissen) dienen dazu, Aussagen über Eigenschaften eines Systems oder eines Individuums zu authentisieren, ohne die Eigenschaft selbst preiszugeben.

Die hierzu nötigen Verfahren basieren auf Commitments; diese sind Thema des nächsten Abschnitts. Danach werden die Verfahren erläutert, mit denen Zero-Knowledge-Beweise interaktiv mit Sigma-Protokollen und nichtinteraktiv mit Signaturen geführt werden können.

3.2. Commitment-Verfahren

Commitment-Verfahren sind ein Grundbaustein von Zero-Knowledge-Beweisen. Sie dienen dazu, ein Geheimnis so in einem *Commitment* (Festlegung) zu verbergen, dass es nicht mehr verändert werden kann. Diese Eigenschaft heißt *Binding* (Bindung). Die zweite Eigenschaft, dass von dem Commitment nicht auf das Geheimnis geschlossen werden kann, heißt *Hiding* (Verdeckung).

Eine weitere wichtige Eigenschaft von Commitments ist, dass man sie *öffnen* kann, also das Geheimnis offenbaren und auf Korrektheit prüfen kann. Die in dieser Arbeit beschriebenen Verfahren verwenden Commitments unterschiedlich. In manchen Fällen werden sie geöffnet, in anderen Fällen aber auch nicht, da Geheimnisse in Zero-Knowledge-Beweisen gerade nicht offengelegt werden sollen.

Eine anschauliche Beschreibung von Commitments ist diese: Das Geheimnis wird auf ein Stück Papier geschrieben, in einen Tresor gelegt und dieser mit einem Schlüssel verschlossen. Der Tresor, der dem Commitment entspricht, wird an den Empfänger übergeben. Dieser kann ihn nicht öffnen (Hiding-Eigenschaft), der Absender kann das Geheimnis nicht mehr ändern (Binding-Eigenschaft). Will der Absender das Commitment öffnen, übergibt er den Schlüssel an den Empfänger. Im Unterschied zu diesem anschaulichen Beispiel lassen sich mit mathematischen Commitments Aussagen über das verborgene Geheimnis beweisen.

Die mathematische Grundlage von Commitment-Verfahren sind Gruppenhomomorphismen, in denen Urbilder praktisch nicht zu berechnen sind. Es gibt zwei Arten von Commitment-Verfahren: die eine bildet das Geheimnis direkt mit einem Gruppenhomomorphismus auf das Commitment ab, während die andere das Geheimnis zunächst um einen Zufallswert erweitert und beides zusammen mit einem geeigneten Gruppenhomomorphismus auf das Commitment abbildet.

Die Binding-Eigenschaften lässt sich noch genauer unterteilen:

- Perfect Binding (Perfekte Bindung): Es ist unmöglich, ein zweites Urbild zu einem Commitment zu finden, weil nur ein einziges existiert. Dies ist genau dann der Fall, wenn der Homomorphismus injektiv ist.

- Computational Binding (Rechnerische Bindung): Mehrere Urbilder bilden auf dasselbe Commitment ab, doch ist es sehr schwierig, neben dem bereits bekannten Urbild ein zweites Urbild zu finden, das auf dasselbe Commitment abbildet. Hierzu müsste zum Beispiel ein diskreter Logarithmus berechnet werden.

Ebenso gibt es auch von der Hiding-Eigenschaft unterschiedliche Ausprägungen:

- Perfect Hiding (Perfekte Verdeckung): Es ist unmöglich, ausgehend von dem Commitment eine Aussage über das Geheimnis zu treffen. Dies ist genau dann der Fall, wenn jedes Geheimnis mit derselben Wahrscheinlichkeit auf jedes Commitment abbildet.

- Statistical Hiding (Statistische Verdeckung): Wenn dasselbe Geheimnis mit unterschiedlichen Wahrscheinlichkeiten auf die verschiedenen Commitments abbildet, die Unterschiede zwischen den Wahrscheinlichkeiten aber vernachlässigbar klein sind, handelt es sich um Statistical Hiding. Dieser Fall tritt zum Beispiel dann ein, wenn die Ordnung der Bildgruppe nicht bekannt ist und in der Urbildgruppe Werte verwendet werden, die die Ordnung der Bildgruppe um Größenordnungen übersteigen.

- Computational Hiding (Rechnerische Verdeckung): Es ist theoretisch, aber nicht praktisch, möglich, das eindeutige Geheimnis aus dem Commitment zu berechnen. Dieser Fall tritt genau dann ein, wenn der Gruppenhomomorphismus injektiv ist, aber Urbilder nicht effizient berechenbar sind.

Von den sechs möglichen Paarungen aus Binding und Hiding sind nur drei möglich:

	Binding		
Hiding	Perfect	Statistical	Computational
Perfect	-	-	✓
Computational	✓	✓	-

Perfect Hiding und Perfect Binding schließen sich gegenseitig aus, da ein Homomorphismus nicht gleichzeitig injektiv sein kann und mehrere Urbilder auf dasselbe Bild abbilden können. Dasselbe gilt für die Paarung aus Perfect Hiding und Statistical Binding.

Computational Binding und Computational Hiding schließen sich aus, weil ein eindeutiges Urbild nicht bestimmt werden kann, wenn es mehrere mögliche Urbilder gibt.

3.2.1. Commitments ohne Zufallskomponente

Es seien A, B Gruppen und $\phi : A \to B$ ein Gruppenhomomorphismus, in dem Urbilder schwer berechenbar sind. Ist ϕ injektiv, so hat das Commitment die Eigenschaften Perfect Binding und Computational Hiding. Anderenfalls hat dieses Commitment, bis auf die fehlende Zufallskomponente, dieselben Eigenschaften wie eines der beiden anderen Verfahren.

Mit einem Monomorphismus ϕ ergeben sich zwei grundlegende Probleme:

- Wenn zwei Commitments denselben Wert haben, folgt aus der Eigenschaft der Injektivität unmittelbar, dass auch die Geheimnisse identisch sein müssen, wodurch das Geheimnis selbst aber nicht aufgedeckt wird.
- Ist bekannt, dass das Geheimnis aus einer kleinen Menge stammt, so kann jedes Element dieser Menge probehalber in ϕ eingesetzt werden. Entspricht das Bild des Homomorphismus dem Commitment, ist das Geheimnis aufgedeckt.

Wenn eines der beiden Probleme für die Anwendung relevant ist, muss ein Verfahren zum Einsatz kommen, welches das Geheimnis um einen Zufallswert erweitert.

Beispiel für ein Verfahren ohne Zufallskomponente:

Seien $A = \mathbb{Z}_{6173},\ B = QR(\mathbb{Z}^*_{12347}),\ g \in B,\ g = [8400],\ \langle g \rangle = B,\ \phi(w) = g^w$.

A ist die additive Gruppe der Restklassen modulo 6173, B ist die Gruppe der quadratischen Reste modulo 12347. Die Ordnung beider Gruppen ist 6173. g ist ein Generator von B. Hieraus folgt, dass ϕ ein Gruppenisomorphismus und somit auch injektiv ist.

Gegeben sind die Commitments $x_0 = \phi(w_0 = [1000]) = [6681],\ x_1 = \phi(w_1) = [9130]$,
$x_2 = \phi(w_2) = [9130],\ x_3 = \phi(w_3) = [5602]$.

Es fällt auf, dass $x_1 = x_2$ gilt, deshalb muss auch $w_1 = w_2$ gelten (es ist der Wert $[100]$).

Von x_3 sei bekannt, dass es das Geburtsjahr des Autors ist, es kommen also nur Werte zwischen $[1890]$ und $[2010]$ in Frage, wobei der Wert wohl zwischen $[1970]$ und $[1990]$ liegen dürfte. Der geneigte Leser möge den korrekten Wert zur Übung ermitteln.

3.2.2. Pedersen-Commitments

Das Pedersen-Commitment ist ein 1991 durch Pedersen beschriebenes Verfahren[Ped92], welches die beiden Probleme des vorigen Verfahrens dadurch löst, dass eine Zufallskomponente eingeführt wird. Das Verfahren von Pedersen garantiert, dass aus einem Commitment keinerlei Rückschlüsse auf das Geheimnis möglich sind. Es hat somit die Eigenschaft Perfect Hiding. Die Binding-Eigenschaft ist aber nicht Perfect, sondern Computational, da theoretisch weitere Urbilder berechnet werden können.

Es seien $(A, \circ), (B, \diamond), (C, \star)$ Gruppen mit derselben bekannten Ordnung q und $\phi_0 : A \to C,\ \phi_1 : B \to C$ seien zwei Gruppenisomorphismen, in denen Urbilder nicht effizient zu berechnen sind.

A ist die Gruppe der Geheimnisse, B ist die Gruppe der Zufallskomponenten und C die Gruppe der Commitments.

Ein dritter Gruppenhomomorphismus sei definiert durch $\phi : A \times B \to C,\ \phi(a,b) \mapsto \phi_0(a) \star \phi_1(b)$. Dieser Homomorphismus ist surjektiv, aber nicht injektiv. Dadurch, dass ϕ_0 und ϕ_1 bijektiv sind, bildet jedes Geheimnis in A auf jedes Commitment in C genau einmal und mit derselben Wahrscheinlichkeit ab. Damit hat das Pedersen-Verfahren Perfect Hiding.

Soll ein Commitment zu $w \in A$ erstellt werden, muss zunächst ein Zufallselement $p \in B$ gewählt werden. Das Commitment ist dann $x = \phi(w,p) = \phi_0(w) \star \phi_1(p)$.

Ein Sicherheitsproblem ergibt sich, wenn eine der Funktionen $f : A \to B$ mit $\phi_0(a) = \phi_1(f(a))$ oder $g : B \to A$ mit $\phi_1(b) = \phi_0(g(b))$ bekannt und effizient berechenbar ist, denn dann gilt:
$\phi(a,b) = \phi_0(a \circ g(b))$ oder $\phi(a,b) = \phi_1(f(a) \diamond b)$.

Dann können zu einem bekannten Urbild weitere Urbilder mit anderen Geheimnissen gefunden werden, die auf dasselbe Commitment abbilden. Dadurch würde die Binding-Eigenschaft nicht mehr gelten, weshalb diese Funktionen nicht bekannt sein dürfen.

Beispiel für Pedersen-Commitments:

Seien $(A,\circ) = (B,\diamond) = \mathbb{Z}_{6173},\ (C,\star) = QR(\mathbb{Z}^*_{12347}),\ g,h \in C,\ g = [8400],\ h = [1110], \langle g \rangle = \langle h \rangle = C,$
$\phi_0(w) = g^w,\ \phi_1(p) = h^p,\ \phi(w,p) = g^w \star h^p$.

Es seien Commitments $x_0 = \phi(w_0 = [1000], p_0 = [2881]) = [6664],\ x_1 = \phi(w_1 = [100], p_1 = [77]) = [4252],\ x_2 = \phi(w_2 = [100], p_2 = [88]) = [8878],\ x_3 = \phi(w_3 = [1234], p_3 = [5005]) = [6664]$ gegeben.

Hier sind $x_0 = x_3$ aber $w_0 \neq w_3$. Ebenso ist $w_1 = w_2$ aber $x_1 \neq x_2$.

Die Funktion $f(a)$ ist: $f(a) = a^{\log_h(g)} = a^{2840}$. Zur Bestimmung dieser Funktion musste der diskrete Logarithmus vom Generator g zur Basis h berechnet werden, was mit großen Zahlen jedoch nicht praktikabel wäre.

Gesucht sei $p_3 \in B$ mit $\phi([1000],[2881]) = \phi([1234],p_3) = [6664]$. Forme um zu:
$\phi_1(f([1000]) \diamond [2881]) = \phi_1(f([1234]) \diamond p_3)$ und weiter zu $[1000]^{2840} \diamond [2881] = [1234]^{2840} \diamond p_3$.
Umgestellt ergibt sich: $p_3 = [1234]^{-2840} \diamond [1000]^{2840} \diamond [2881] = [1704] \diamond [420] \diamond [2881] = [5005]$.
Dies ist genau die Zufallskomponente von Commitment x_3. (Hinweis: $\diamond$ ist die modulare Addition, weshalb die Potenz der modularen Multiplikation entspricht!)

3.2.3. Damgård-Fujisaki-Commitments

Mit dem Pedersen-Commitment ist man auf q (Ordnung der verwendeten Gruppen) verschiedene Geheimnisse beschränkt. In manchen Situationen ist es jedoch notwendig, als Geheimnis eine beliebige ganze Zahl $w \in \mathbb{Z}$ zu wählen. Damgård und Fujisaki beschreiben in [DF02] ein Verfahren hierzu.

Es wird ein Modulus $n = p \cdot q$ benötigt mit zwei natürlichen Zahlen $p,q \in \mathbb{N}$ für die gilt:

Die Kongruenz $p \equiv q \equiv 3 \pmod 4$ ist erfüllt, $\text{ggT}(p-1, q-1) = 2$ und die Zahlen $p-1, q-1$ haben nicht zu viele kleine Primfaktoren. ([DF02, S. 3])

Für p, q können zwei sichere Primzahlen gewählt werden. Es ist aber aufwendig, diese zu erzeugen. Des Weiteren werden zwei Generatoren g, h für eine große Untergruppe von $\mathbb{Z}_n^*$ benötigt. Sind p, q sichere Primzahlen, sind fast alle quadratischen Reste in $\mathbb{Z}_n^*$ geeignete Generatoren. Sind die Primfaktoren p, q nicht bekannt, ist es sehr schwierig zu prüfen, ob ein Generator tatsächlich die Gruppe der quadratischen Reste erzeugt. Daher müssen die Generatoren von derselben Partei gewählt werden, die auch den Modulus berechnet hat.

Die Faktorisierung von n und die diskreten Logarithmen $\log_g(h),\ \log_h(g)$ dürfen dem Erzeuger von Commitments nicht bekannt sein; letzteres wurde bereits im Rahmen der Pedersen-Commitments begründet.

Die Gruppe $(A, +)$ der Geheimnisse und die Gruppe der Zufallskomponenten $(B, +)$ sind $\mathbb{Z}$, also alle ganzen Zahlen. Die Gruppe der Commitments ist $(C, \cdot) = \langle g \rangle = \langle h \rangle \subsetneq \mathbb{Z}_n^*$. Der Homomorphismus zur Erzeugung von Commitments ist ein Spezialfall des Pedersen-Commitments:
$\phi : A \times B \to C,\ (w, p) \mapsto g^w \cdot h^p$

Um ein Commitment zu einer Zahl $w \in \mathbb{Z}$ zu berechnen, muss ein Zufallswert $p \in [0, 2^{D+k}]$ erzeugt werden mit $k = \lfloor \log_2(n) \rfloor$ und mit einem Sicherheitsparameter $D \in \mathbb{N}$, der polynomiell von k abhängen muss, wobei $D = k$ gewählt werden kann. Das Commitment berechnet sich dann durch $x = g^w \cdot h^p$.

Diese Commitments haben Statistical Hiding und Computational Binding.

Erstes Beispiel:

Sei $n = 13393$ das Produkt zweier sicherer Primzahlen und $D = k = \lfloor \log_2(n) \rfloor = 13$. Die zwei Generatoren seien $g = [5738]$, $h = [11497]$.

Es werden die Commitments $x_0 = \phi(w_0 = 731, p_0 = 48391495) = [2910],\ x_1 = \phi(w_1 = 40055, p_1 = 22975) = [2910]$ berechnet.

Die Commitments x_0 und x_1 sind identisch, obwohl weder $w_0 = w_1$ noch $p_0 = p_1$ gilt. Jedoch sind $40055 - 731 = 39324$ und $48391495 - 22975 = 48368520$ beides Vielfache der Gruppenordnung $\frac{58 \cdot 226}{4} = 3277$. Ist ein Vielfaches der Gruppenordnung bekannt, kann hieraus die Faktorisierung von n berechnet werden, weshalb die Sicherheit des Damgård-Fujisaki-Verfahrens auch von der Schwierigkeit des Faktorisierungsproblems abhängt.

Zweites Beispiel:

Seien $n = 55,\ k = 5$ und die Generatoren seien $g = [14],\ h = [49]$. D sei 1 und damit zu niedrig. Es sei bekannt, dass das geheime w aus dem Intervall $[1, 3]$ stammt.

Diese Tabelle zeigt die Häufigkeiten der Commitments x in Abhängigkeit vom gewählten Geheimnis w, wenn die Zufallskomponente $p \in [0, 64)$ gleichverteilt gewählt wird:

		Commitment x									
		[1]	[4]	[9]	[14]	[16]	[26]	[31]	[34]	[36]	[49]
w	1	7	6	7	7	6	7	6	6	6	6
	2	6	6	6	7	7	6	7	7	6	6
	3	6	7	6	6	6	6	7	6	7	7

Hieran lässt sich erkennen, dass das Verfahren Statistical Hiding hat. Ist beispielsweise $x = [34]$, so ist $w = 2$ wahrscheinlicher als $w = 1$ oder $w = 3$. Werden jedoch ein größerer Modulus, mehr mögliche Geheimnisse und insbesondere ein größerer Sicherheitsparameter D gewählt, verliert der statistische Unterschied an Bedeutung.

Für eine ausführliche Abhandlung über Commitment-Verfahren sei auf [DN08] verwiesen.

3.3. Proof of Knowledge

Ein *Proof of Knowledge* (Wissensbeweis) ist ein Protokoll zwischen zwei Parteien: dem Prover $\mathcal{P}$ (Beweiser) und dem Verifier $\mathcal{V}$ (Prüfer). Der Prover versucht, den Verifier davon zu überzeugen, dass er ein oder mehrere Geheimnisse kennt, die bestimmte Eigenschaften erfüllen. Wenn der Verifier durch dieses Protokoll keine neuen Erkenntnisse über diese Geheimnisse gewinnt, so handelt es sich um einen *Zero-Knowledge Proof of Knowledge* (Wissensbeweis ohne Wissen).

Die beweisbaren Eigenschaften der Geheimnisse umfassen:

- Das Geheimnis ist das Urbild eines Commitments, der Prover kann das Commitment also öffnen.
- Mehrere Geheimnisse stehen in einer linearen, multiplikativen oder polynomiellen Beziehung zueinander.
- Ein Geheimnis liegt in einem Intervall.
- Von mehreren Geheimnissen sind alle bekannt.
- Von mehreren Geheimnissen ist mindestens eines bekannt.

Sowohl Prover als auch Verifier können ehrlich (honest) oder arglistig (malicious) sein:

- Ein ehrlicher Prover kennt das Geheimnis, während der arglistige Prover es nicht kennt und auf anderem Weg versucht, den Verifier zu überzeugen.
- Ist der Verifier ehrlich, hält er sich strikt an das Protokoll, darf aber trotzdem versuchen, aus den übertragenen Informationen Erkenntnisse zu gewinnen. Ein arglistiger Verifier kann vom Protokoll in beliebiger Weise abweichen. Zum Beispiel kann er die Challenge (mehr dazu später) gezielt wählen, anstatt sie zufällig zu erzeugen. Hierdurch könnte er in bestimmten Situationen das Geheimnis berechnen. ([S$^+$09])

Die in dieser Arbeit beschriebenen interaktiven Verfahren gehen davon aus, dass der Verifier ehrlich ist. Nichtinteraktive Beweise sind auch dann sicher, wenn der Verifier arglistig ist.

Zero-Knowledge Proofs of Knowledge zeichnen sich formal gesehen durch drei Eigenschaften aus:

- Vollständigkeit (completeness): Wenn der Prover das Geheimnis kennt, kann er den Verifier hiervon fast immer überzeugen. Die in dieser Arbeit verwendeten Verfahren sind so gestaltet, dass $\mathcal{V}$ immer überzeugt wird.
- Korrektheit (soundness): Wenn der Prover den Verifier fast immer überzeugen kann, so muss er das Geheimnis mit nicht zu vernachlässigender Wahrscheinlichkeit tatsächlich kennen oder er kann es zumindest berechnen.
- Zero-Knowledge: Der Verifier erhält durch den Beweis keine neuen Informationen über das Geheimnis. Dies lässt sich in perfektes und statistisches Zero-Knowledge unterteilen. Letzteres offenbart vernachlässigbar wenig Informationen über das Geheimnis, während das Erstere gar nichts verrät.

3.4. Camenisch-Stadler-Notation

Um formal aufzuschreiben, was mit einem Zero-Knowledge-Beweis gezeigt werden soll, hat sich die durch Camenisch und Stadler[CS97] geprägte Notation durchgesetzt:

$$\text{ZPK}\Big[(\text{Variablenliste}) : \text{Prädikat}\Big]$$

Die Variablenliste zwischen den runden Klammern ist eine Liste von geheimen Variablen, die mit kleinen griechischen Buchstaben bezeichnet werden. Diese können mit Indizes, Strichen und Ähnlichem modifiziert werden.

Das Prädikat hinter dem Doppelpunkt ist ein Ausdruck, der abhängig von der Belegung der Variablen wahr oder falsch sein kann. Das Prädikat hat keine wohldefinierte Form, folgt in der Regel aber der üblichen logischen und mathematischen Notation. Alle Bestandteile des Prädikats, die nicht in der Variablenliste auftauchen, sind sowohl dem Prover als auch dem Verifier bekannt.

Die Aufgabe des Provers ist es, den Verifier davon zu überzeugen, dass er eine Belegung der geheimen Variablen kennt, mit der das Prädikat wahr ist.

Beispiel:

$$\text{ZPK}\Big[(\alpha,\beta) : x = g^{\alpha}h^{\beta} \ \wedge \ y = \phi(\alpha) \ \wedge \ \beta \in [100,200]\Big]$$

In diesem Beispiel muss der Prover zeigen, dass er eine Belegung der beiden geheimen Variablen α, β kennt, so dass $x = g^{\alpha}h^{\beta}$ und $y = \phi(\alpha)$ wahr sind. Außerdem muss er zeigen, dass β im Intervall $[100,200]$ liegt. Die Werte von x, y, g, h, der Homomorphismus ϕ, die Intervallgrenzen, usw. sind dem Prover und dem Verifier bekannt.

Dieses Beispiel ist noch sehr einfach; in der Literatur sind Beweise zu finden, deren Umfang mehr als eine Seite beträgt.

Ein Ausdruck wie $\mathrm{ZPK}\left[(\xi) : \xi > 3\right]$ macht keinen Sinn, denn jeder kennt eine größere Zahl als 3. Hier gibt es nichts zu beweisen. Deshalb müssen die geheimen Variablen immer an Commitments oder Ähnliches gebunden sein.

3.5. Sigma-Protokoll

Mit der Camenisch-Stadler-Notation kann spezifiziert werden, was in Zero-Knowledge bewiesen werden soll. Die gebräuchliche Methode, um diese Zero-Knowledge-Beweise durchzuführen, ist das Sigma-Protokoll. Das Sigma-Protokoll wurde bereits in [S$^+$09] zusammengefasst. Dieser und die folgenden Abschnitte sind hieran angelehnt, folgen aber in vielerlei Hinsicht eigenen Ideen.

Das Sigma-Protokoll ist ein interaktives Verfahren zwischen zwei Parteien, dem Prover $\mathcal{P}$ und dem Verifier $\mathcal{V}$. Während der Durchführung des Sigma-Protokolls werden drei Nachrichten im Wechsel zwischen den Parteien ausgetauscht, weshalb es interaktiv heißt. Durch eine Modifikation ist es möglich, das Sigma-Protokoll nichtinteraktiv durchzuführen, wodurch nur noch eine Nachricht ausgetauscht werden muss und somit die Interaktion wegfällt.

Mit dem Sigma-Protokoll kann der Prover in Zero-Knowledge beweisen, dass er eine Belegung von Variablen kennt, unter der ein Prädikat gilt. Es gibt verschiedene Arten von Prädikaten und für jede Art wird eine spezielle Variante des Sigma-Protokolls benötigt.

Jede Variante des Sigma-Protokolls besteht aus einer Reihe von Komponenten. Während die konkrete Ausgestaltung der Komponenten für jede Variante des Sigma-Protokolls separat betrachtet werden muss, werden die Eigenschaften der gemeinsamen Komponenten im Folgenden kurz erläutert:

Im Sigma-Protokoll muss ein Sicherheitsparameter $c^+ \in \mathbb{N},\ c^+ \geq 2$ festgelegt werden, der die obere Grenze für den Wert der Challenge (Herausforderung) c angibt.

Die geheime Belegung der Variablen wird mit w bezeichnet. Mit x wird ein öffentlicher Wert bezeichnet, der zu w in Beziehung steht, beispielsweise wenn x ein Commitment zu w ist.

Im Laufe des Protokolls werden drei Nachrichten ausgetauscht:

- r wird als erste Nachricht von $\mathcal{P}$ an $\mathcal{V}$ gesendet; dies ist meistens ein Commitment zu einem Zufallswert, weshalb r als Commitment bezeichnet wird.
- $c \in \mathbb{N},\ c \in [0, c^+)$ ist die Challenge und wird von $\mathcal{V}$ an $\mathcal{P}$ gesendet. In einigen Teilen der Literatur ist c^+ im Intervall enthalten. Hierdurch werden aber die Notation von einigen Formeln und die Implementierung aufwendiger.
- s ist die Response (Antwort) auf die Challenge und wird als letzte Nachricht von $\mathcal{P}$ an $\mathcal{V}$ geschickt.

Diese drei Nachrichten werden auch häufig als Tripel (r,c,s) dargestellt. So ein Tripel heißt *korrekt* genau dann, wenn der Algorithmus V es akzeptiert.

Im Sigma-Protokoll ist eine Reihe von Algorithmen definiert:

- $P_1(x) \to (state, r)$ wird nur von $\mathcal{P}$ verwendet. Die Eingabe ist der öffentliche Wert x. $\mathcal{P}$ erzeugt die Nachricht r und merkt sich im Status $state$ zum Beispiel ein geheimes Urbild von r.
- $P_2(state, w, c) \to (s)$ berechnet aus dem Status, dem Geheimnis w und der Challenge c die Response s. Wie P_1 wird auch P_2 ausschließlich von $\mathcal{P}$ benutzt.
- $S(x,c) \to (r,s)$ ist ein Simulator. Wenn $\mathcal{P}$ die Challenge c kennt oder errät, so kann er anstatt P_1 und P_2 diesen Algorithmus benutzen, um ein korrektes Tripel (r,c,s) zu berechnen, auch wenn $\mathcal{P}$ das Geheimnis w überhaupt nicht bekannt ist. Hierzu erzeugt $\mathcal{P}$ eine zufällige Response s und berechnet $R(x,c,s) \to (r)$.
- $R(x,c,s) \to (r)$ ist ein Algorithmus, der r so aus (x,c,s) berechnet, dass ein korrektes Tripel (r,c,s) entsteht. Dieser Algorithmus wird als Teil des Simulators gebraucht, aber auch um nichtinteraktive Beweise auszuführen. Letzteres wird in Kapitel 3.9 erläutert.
- $C() \to (c)$ generiert eine zufällige Challenge $c \in_R [0, c^+)$.
- $V(x,r,c,s)$ wird von $\mathcal{V}$ verwendet, um ein Tripel auf Korrektheit zu prüfen. Schlägt die Überprüfung fehl, so kann $\mathcal{V}$ davon ausgehen, dass $\mathcal{P}$ keine Variablenbelegung kennt, unter der das Prädikat wahr ist.

Ein nicht simulierter, normaler Protokollablauf ist im folgenden Diagramm zu sehen. Die Eingaben von $\mathcal{P}$ und $\mathcal{V}$ sind (x,w) respektive (x). Die Zeit verläuft von oben nach unten, die beschrifteten Pfeile stellen eine Kommunikation zwischen $\mathcal{P}$ und $\mathcal{V}$ dar.

$\mathcal{P}[x,w]$	Normaler Ablauf	$\mathcal{V}[x]$
$P_1(x) \to (state, r)$	$(r) \longrightarrow$	
		$C() \to (c),\ c \in_R [0, c^+)$
	$\longleftarrow (c)$	
$P_2(state, w, c) \to (s)$	$(s) \longrightarrow$	
		$V(x,r,c,s) \stackrel{?}{=} \#\, t$

Kennt $\mathcal{P}$ den geheimen Wert w nicht, muss er das Protokoll simulieren:

$\mathcal{P}[x,c]$	Simulierter Ablauf	$\mathcal{V}[x]$
$S(x,c) \to (r,s)$	$(r) \longrightarrow$	
		$C() \to (c),\ c \in_R [0, c^+)$
	$\longleftarrow (c)$	
	$(s) \longrightarrow$	
		$V(x,r,c,s) \stackrel{?}{=} \#\, t$

Die drei Eigenschaften von Zero-Knowledge-Beweisen erfüllen Sigma-Protokolle wie folgt:

- **Vollständigkeit:** Ein ehrlicher Prover kann immer P_1 und P_2 benutzen, um korrekte Tripel (r,c,s) zu berechnen. Deshalb sind Sigma-Protokolle vollständig.

- **Korrektheit:** Ein arglistiger Prover, dem es dennoch mit hoher Wahrscheinlichkeit gelingt, den Verifier zu überzeugen, kann das Geheimnis w berechnen. Wie dies konkret aussieht, ist in den entsprechenden Protokollen beschrieben.

- **Zero-Knowledge:** Um zu zeigen, dass ein Verifier durch das Sigma-Protokoll keine neuen Erkenntnisse erhält, wird ein Simulator benötigt, der den aktuellen Zustand des Verifiers auf einen vorherigen zurücksetzen kann. Der Simulator verwendet den folgenden Algorithmus:

  1. Erzeuge eine zufällige Challenge $c' \in [0, c^+)$.

  2. Führe den Simulator $S(x,c') \rightarrow (r,s)$ aus und sende r an den Verifier.

  3. Speichere den Zustand des Verifiers so ab, dass er später wiederherzustellen ist.

  4. Der Verifier führt den Algorithmus $C() \rightarrow (c)$ aus und schickt c an den Simulator. Wenn $c \neq c'$ gilt, so stelle den vorherigen Zustand wieder her und wiederhole diesen 4. Schritt. Gilt $c = c'$, so fahre mit dem nächsten Schritt fort.

  5. Schicke s an den Verifier. (r,c,s) ist ein korrektes Tripel und wird von V akzeptiert.

  Dieser Ablauf entspricht aus Sicht des Verifiers dem normalen Ablauf des Sigma-Protokolls. Da der Prover nicht involviert ist, kann er sein Geheimnis nicht verraten und der Verifier erhält durch das Protokoll keine neuen Informationen.

  Wird das Sigma-Protokoll statt mit dem Simulator mit einem echten, ehrlichen Prover ausgeführt, wird auch hier ein korrektes Tripel (r,c,s) erstellt.

  In beiden Fällen (Simulator, ehrlicher Prover) wird ein korrektes Nachrichtentripel erstellt. Anhand der Nachrichten ist es dem Verifier jedoch nicht möglich, zwischen diesen beiden Fällen zu unterscheiden. Daraus folgt, dass das Sigma-Protokoll die Zero-Knowledge-Eigenschaft erfüllt.

Wenn der Prover die Challenge c errät, bevor er r sendet, kann er mit dem Simulator S korrekte Tripel (r,c,s) berechnen, die vom Verifier akzeptiert werden, ohne dass er das geheime w kennt. Somit hat ein arglistiger Prover eine Chance von $\frac{1}{c^+}$, erfolgreich zu betrügen.

Wird das Sigma-Protokoll k-mal ausgeführt, so verringert sich die Betrugswahrscheinlichkeit auf $\left(\frac{1}{c^+}\right)^k$. Deshalb muss das Protokoll je nach Wahl von c^+ und anderen Parametern mehrfach hintereinander ausgeführt werden, um die Betrugswahrscheinlichkeit auf einen gewünschten Wert zu senken.

3.5.1. Sigma-Phi (Σ^ϕ)

Das Σ^ϕ-Protokoll ist die einfachste Variante des Sigma-Protokolls. Mit diesem kann der Prover gegenüber dem Verifier beweisen, dass er ein Urbild in einem Homomorphismus ϕ kennt bzw. dass er ein Commitment öffnen kann. In Camenisch-Stadler-Notation entspricht dies: $\text{ZPK}\Big[(\alpha) : x = \phi(\alpha)\Big]$.

Damit das Σ^ϕ-Protokoll sicher ist, muss ϕ ein Gruppenepimorphismus zwischen zwei endlichen, zyklischen Gruppen sein. Die Zufallselemente aus der Urbildmenge müssen so gezogen werden können, dass die Bilder gleichmäßig verteilt sind.

Gegeben seien also zwei endliche zyklische Gruppen $(A, \circ)$ und $(B, \diamond)$ sowie ein Gruppenepimorphismus $\phi : A \to B$, mit dem Commitments berechnet werden können, also Urbilder schwierig zu berechnen sind. Der geheime Wert $w \in A$ sei nur dem Prover bekannt, während das öffentliche Commitment $x = \phi(w),\ x \in B$ sowohl dem Prover als auch dem Verifier bekannt ist.

Der Prover will den Verifier davon überzeugen, dass er eine Belegung der Variablen α kennt, mit der das Prädikat $x = \phi(\alpha)$ wahr ist. Ein ehrlicher Prover kann $\alpha = w$ einsetzen.

Die Komponenten des Sigma-Protokolls definiert Σ^ϕ wie folgt:

- $w, k, s \in A$
- $x, r \in B$
- P_1: $k \in_R A,\ r = \phi(k),\ state = (k)$
- P_2: $s = k \circ w^c$
- S: $s \in_R A,\ r = \phi(s) \diamond x^{-c}$
- R: $r = \phi(s) \diamond x^{-c}$
- V: $\phi(s) \stackrel{?}{=} r \diamond x^c$

Der Protokollablauf ist demnach:

$\mathcal{P}[x, w]$		$\mathcal{V}[x]$
$P_1 :\ k \in_R A,\ r = \phi(k)$		
	$\xrightarrow{\quad (r) \quad}$	
		$c \in_R [0, c^+)$
	$\xleftarrow{\quad (c) \quad}$	
$P_2 :\ s = k \circ w^c$		
	$\xrightarrow{\quad (s) \quad}$	
		$V :\ \phi(s) \stackrel{?}{=} r \diamond x^c$

Die für Zero-Knowledge-Beweise notwendigen Eigenschaften sind erfüllt:

- **Vollständigkeit:** Mit $x = \phi(w)$ gilt: $\phi(s) = \phi(k \circ w^c) = \phi(k) \diamond \phi(w^c) = \phi(k) \diamond \phi(w)^c = r \diamond x^c$. Kennt der Prover dieses w, kann er immer und ohne zu raten korrekte Tripel (r, c, s) berechnen, die vom Verifier akzeptiert werden.

- **Korrektheit:** Wenn der Prover meistens korrekte Werte an den Verifier schickt, kann der Prover auch zwei korrekte Tripel $(r, c_0, s_0), (r, c_1, s_1)$ berechnen mit $c_1 = c_0 + 1$ und demselben r. Dann gilt: $x = \phi(w = s_0^{-1} \circ s_1)$, er kann also das Geheimnis w berechnen.

  Beweis: Es gelten $\phi(s_0) = r \diamond x^{c_0}$ und $\phi(s_1) = r \diamond x^{c_0+1}$.
  Es folgt: $\phi(s_0^{-1} \circ s_1) = \phi(s_0)^{-1} \diamond \phi(s_1) = (r \diamond x^{c_0})^{-1} \diamond (r \diamond x^{c_0+1}) = (x^{-c_0} \diamond r^{-1}) \diamond (r \diamond x^{c_0} \diamond x) = x$.

  Wenn ϕ nicht injektiv ist, können neben $s_0^{-1} \circ s_1$ noch weitere Lösungen existieren; da x aber ein Commitment ist, ist es schwierig, eine andere Lösung zu finden. Zu vermuten ist daher, dass der Prover das „richtige“ w kennt.

  Der Prover muss darauf achten, den Wert k auch wirklich zufällig zu wählen. Ansonsten kann der Verifier auf demselben Weg das Geheimnis w berechnen.

- **Zero-Knowledge:** Aus r können keine Rückschlüsse über w gezogen werden, da w in die Berechnung von r nicht eingeht. Aus s kann man ebenfalls keine Informationen über w gewinnen. Zwar fließt w direkt in die Berechnung von s mit ein, aber da k gleichverteilt zufällig gewählt ist, ist auch s gleichverteilt. Ansonsten gilt hier die Argumentation, die für das Sigma-Protokoll getroffen wurde.

Beispiel:

Gegeben seien die endlichen, zyklischen Gruppen $(A, +) = \mathbb{Z}_{11}$ mit modularer Addition und $(B, *) = QR(\mathbb{Z}_{23}^*)$ mit modularer Multiplikation sowie ein Generator $g \in B,\ g = [3],\ B = \langle g \rangle$.

Der Homomorphismus sei $\phi : A \to B,\ a \mapsto g^a$ und $c^+ = 11$ sei gleich der Gruppenordnung.

$\mathcal{P}$ hat das Geheimnis $w \in A,\ w = [6]$ und berechnet das Commitment $x = \phi(w) = [3]^{[6]} = [16]$, welches auch $\mathcal{V}$ bekannt gemacht wird.

Um gegenüber $\mathcal{V}$ zu beweisen, dass er das Commitment x öffnen kann, führt $\mathcal{P}$ mit $\mathcal{V}$ das Σ^ϕ-Protokoll aus:

- P_1: $\mathcal{P}$ wählt ein zufälliges $k \in A$, z. B. $k = [8]$. $\mathcal{P}$ berechnet $r = \phi(k) = [3]^{[8]} = [6]$.
- $\mathcal{P}$ überträgt $r = [6]$ an $\mathcal{V}$.
- C: $\mathcal{V}$ erzeugt eine zufällige Challenge $c = 4$.
- $\mathcal{V}$ sendet $c = 4$ zu $\mathcal{P}$.
- $\mathcal{P}$ berechnet $s = k + w^c = [8] + [6]^4 = [10]$.
- $\mathcal{P}$ überträgt $s = [10]$ an $\mathcal{V}$.

- $\mathcal{V}$ prüft, ob $\phi(s) = r * x^c$ gilt: $\phi(s) = [3]^{[10]} = [8] = [6] * [16]^4$. Dies gilt, also kann sich $\mathcal{V}$ mit ca. 91%iger Wahrscheinlichkeit $\left(100 \cdot \left(1 - \frac{1}{c^+}\right)\right)$ sicher sein, dass $\mathcal{P}$ das Geheimnis kennt. $\mathcal{V}$ ist damit noch nicht zufrieden und die beiden Parteien wiederholen das Protokoll.

- $\mathcal{P}$ wählt zufällig wieder $k_1 = [8]$, berechnet $r_1 = [6]$ und überträgt dies an $\mathcal{V}$. Diesmal hält sich $\mathcal{V}$ nicht an das Protokoll und sendet, anstatt eines Zufallwerts, $c_1 = [5]$ an $\mathcal{P}$.

- $\mathcal{P}$ berechnet $s_1 = k_1 + w^c = [5]$ und schickt dies an $\mathcal{V}$. Dieser kann nun $w = -s + s_1 = [-10] + [5] = [-5] = [6]$ berechnen.

Der Verifier konnte mit Hilfe des Σ^ϕ-Protokolls das Geheimnis w berechnen. Die Ursache dieses Problems ist aber weder das Σ^ϕ-Protokoll noch ein Fehler beim Prover, sondern ist auf die verwendete Gruppengröße zurückzuführen. Werden Gruppen verwendet, deren Ordnung um Größenordnungen höher ist (z. B. 2^{1024}), ist die Wahrscheinlichkeit dafür, zwei identische Nachrichten r, r_1 zu generieren, vernachlässigbar klein.

3.5.2. Sigma-GSP (Σ^{GSP})

Mit dem Σ^{GSP}-Protokoll (Generalized Schnorr Proof) kann ein Prover zeigen, dass er ein Urbild in einem Homomorphismus ϕ kennt, also ein Commitment öffnen kann. Dies ist in Camenisch-Stadler-Notation: $\text{ZPK}\left[(\alpha) : x = \phi(\alpha)\right]$. Das Schnorr-Protokoll entspricht weitestgehend dem Σ^ϕ-Protokoll.

Die Unterschiede des Σ^{GSP}-Protokolls zum Σ^ϕ-Protokoll bestehen in den verwendeten Gruppen und in dem zugrundeliegenden Commitment-Verfahren. Während das Σ^ϕ-Protokoll Computational oder Perfect Hiding voraussetzt, benötigt das Σ^{GSP}-Protokoll ein Commitment-Verfahren mit Statistical Hiding.

Die Bildgruppe des Homomorphismus ϕ hat eine unbekannte Ordnung (zum Beispiel $\mathbb{Z}_n^*$ mit zusammengesetztem n) und die Urbildgruppe muss ein Isomorphismus von $\mathbb{Z}^t$, dem kartesischen Produkt aus t ganzen Zahlen, sein. Die Komponenten der Urbildgruppe müssen als Exponenten verwendet werden.

Seien $(D, \circ) = QR(\mathbb{Z}_n^*),\ g, h \in D,\ \langle g \rangle = \langle h \rangle = D$. Dann eignen sich beispielsweise die folgenden Gruppen und Homomorphismen für das Σ^{GSP}-Protokoll:

- $E = \mathbb{Z},\ \phi_E : E \to D,\ w \mapsto g^w$

- $F = \mathbb{Z}^2,\ \phi_F : F \to D,\ (w_0, w_1) \mapsto g^{w_0} \circ h^{w_1}$

- $G = (\mathbb{Z}, \mathbb{Z}^2),\ \phi_G : G \to D^2,\ (v, (w_0, w_1)) \mapsto (g^v, h^{w_0 + w_1})$

Es seien nun $(A, \circ) = \mathbb{Z}^t$ die unendliche Gruppe der Geheimnisse und $(B, \diamond)$ eine endliche zyklische Gruppe mit unbekannter Ordnung für die Commitments. $\phi : A \to B$ sei ein Gruppenepimorphismus mit Exponentation (siehe vorheriges Beispiel).

Operationen auf Elementen von A werden auf deren Komponenten ausgeführt. So gilt für $a,\ b \in A$: $a \circ b = (a_0 \circ b_0, a_1 \circ b_1, \ldots, a_{t-1} \circ b_{t-1})$. Ebenso verhält es sich mit Inversen, dem neutralen Element und auch mit Intervallen: Seien $L, R \in A$ und $L_i \leq R_i$ für $0 \leq i < t$. Ein Element $a \in A$ ist genau dann in dem Intervall $[L,R]$ enthalten, wenn für jede Komponente von a gilt: $a_i \in [L_i, R_i]$.

Das Geheimnis $w \in A$ liege in einem Intervall $[L,R]$ und es sei definiert: $m = L \circ R^{-1}$. Das Commitment $x = \phi(w)$ sei sowohl dem Prover als auch dem Verifier bekannt.

Zusätzlich sei ein Sicherheitsparameter $l \in \mathbb{N}$ gegeben, der angibt, wie dicht die Verteilung der Commitments an der Gleichverteilung liegt. Übliche Werte liegen zwischen 80 und 100.

Genau wie beim Σ^{ϕ}-Protokoll will der Prover den Verifier davon überzeugen, dass er eine Belegung der Variablen α kennt, mit der das Prädikat $x = \phi(\alpha)$ wahr ist. Auch hier kann der ehrliche Prover $\alpha = w$ einsetzen.

Die wesentlichen Unterschiede gegenüber dem Σ^{ϕ}-Protokoll liegen in den Algorithmen P_1, P_2, S, R, V. Die Komponenten von Σ^{GSP} sind wie folgt definiert:

- $w, k, s \in A$
- $x, r \in B$
- P_1: $k \in_R [m^{-2^l \cdot c^+}, m^{2^l \cdot c^+}],\ r = \phi(k),\ state = (k)$
- P_2: $s = k \circ (w \circ L^{-1})^c$
- S: $a \in_R [m^{-2^l \cdot c^+}, m^{2^l \cdot c^+}],\ b \in_R [L,R],\ s = a \circ (b \circ L^{-1})^c.\ r = \phi(s \circ L^c) \diamond x^{-c}$.
- R: $r = \phi(s \circ L^c) \diamond x^{-c}$
- V: $s \stackrel{?}{\in} [m^{-2^l \cdot c^+}, m^{2^l \cdot c^+ + c}] \ \wedge\ \phi(s \circ L^c) \stackrel{?}{=} r \diamond x^c$

Die Eigenschaften von Zero-Knowledge-Beweisen gelten:

- **Vollständigkeit**: Es gilt $\phi(s \circ L^c) = \phi(k \circ (w \circ L^{-1})^c \circ L^c) = \phi(k + w^c) = r \diamond x^c$. Es ist auch einfach zu sehen, dass s im richtigen Intervall liegt, wenn sich der Prover an das Protokoll hält.
- **Korrektheit:** Ebenso wie beim Σ^{ϕ}-Protokoll kann hier der Prover zwei korrekte Tripel (r, c_0, s_0) und (r, c_1, s_1) mit $c_1 = c_0 + 1$ berechnen. Es gilt: $x = \phi(w = s_0^{-1} \circ s_1 \circ L)$.

  Beweis: Es gelten $\phi(s_0 \circ L^{c_0}) = r \diamond x^{c_0}$ und $\phi(s_1 \circ L^{c_0+1}) = r \diamond x^{c_0+1}$.
  Es folgt: $\phi(s_0^{-1} \circ s_1 \circ L) = \phi(s_0^{-1} \circ (L^{-c_0} + L^{c_0}) \circ s_1 \circ L) = \phi((L^{-c_0} \circ s_0^{-1}) \circ (s_1 \circ L^{c_0} \circ L)) = \phi(L^{-c_0} \circ s_0^{-1}) \diamond \phi(s_1 \circ L^{c_0+1}) = \phi(s_0 \circ L^{c_0})^{-1} \diamond \phi(s_1 \circ L^{c_0+1}) = (r \diamond x^{c_0})^{-1} \diamond (r \diamond x^{c_0+1}) = x^{-c_0} \diamond r^{-1} \diamond r \diamond x^{c_0} \diamond x = x$.

  Neben diesem w gibt es unendlich viele weitere Urbilder von x. Da aber x ein Commitment ist, ist es schwierig, eine weitere Lösung zu finden. Eine weitere Möglichkeit, Urbilder zu berechnen, ergäbe sich, wenn die Ordnung der Bildgruppe bekannt wäre. Dann könnte die Ordnung zu

einem w_i addiert werden, um dasselbe Commitment zu erhalten. Die Schwierigkeit, die Ordnung zu ermitteln, hängt aber zum Beispiel vom Faktorisierungsproblem ab.

Daher kann man auch bei dem Σ^{GSP}-Protokoll davon ausgehen, dass ein erfolgreicher Prover das „richtige“ Geheimnis kennt.

- **Zero-Knowledge:** Die Nachricht r hängt nicht von w ab, daher können hieraus keine Informationen über w gewonnen werden. Aus s kann aber mit vernachlässigbar kleiner Wahrscheinlichkeit auf w geschlossen werden: Kein Paar (w,c) kann auf jedes s abbilden. Ist s am „Rand“ des möglichen Intervalls, kann auf w geschlossen werden (siehe Beispiel). Dasselbe Problem ergibt sich aber auch im Fall des Simulators, wie er für das Sigma-Protokoll definiert wurde. Da das simulierte s nicht vom „echten“ s zu unterscheiden ist, gilt für Σ^{GSP} die Zero-Knowledge-Eigenschaft.

Beispiel:

Gegeben seien die Gruppen $(A,+) = Z \times Z$ und $(B,*) = QR(\mathbb{Z}_{77}^*)$, zwei Generatoren $g,h \in B,\ g = [9],\ h = [37]$ und der Homomorphismus $\phi : A \to B,\ (w,p) \mapsto g^w * h^p$. Das Geheimnis $(w,p) \in A$ liegt im Intervall $[L,R]$ mit $L = (3,0),\ R = (5,4096)$ und $m = R \circ L^{-1} = (2,4096)$. Die Sicherheitsparameter seien $c^+ = 2$ und $l = 1$.

Das Geheimnis sei $w = 5$. Nach den Regeln des Damgård-Fujisaki-Commitments wählt $\mathcal{P}$ einen Zufallswert $p = 2731$ und berechnet dann $x = g^w * h^p = [9]^5 * [37]^{2731} = [67] * [37] = [15]$.

Das Σ^{GSP}-Protokoll zwischen $\mathcal{P}$ und $\mathcal{V}$ läuft wie folgt ab:

- $P_1 : \mathcal{P}$ wählt $k \in [(-8,-16384),\ (8,16384)]$, z. B. $k = (8,12345)$. $\mathcal{P}$ berechnet $r = \phi(k) = [25]$ und überträgt $r = [25]$ an $\mathcal{V}$.
- $C : \mathcal{V}$ erzeugt eine zufällige Challenge $c = 1$ und sendet $c = 1$ an $\mathcal{P}$.
- $\mathcal{P}$ berechnet $s = k + ((w,p) + L^{-1})^c = (8,12345) + ((5,2731) + (-3,0))^1 = (10,15076)$ und überträgt $s = (10,15076)$ an $\mathcal{V}$.
- $V : \mathcal{V}$ stellt fest, dass $s \in [(-8,-16384),\ (10,20480)]$ und $\phi(s + L^c) = \phi(13,15076) = [67] = [25] * [15]^1$ gelten und akzeptiert.

$\mathcal{V}$ kann nun direkt das Geheimnis w nennen. Die folgende Tabelle zeigt, bei welchen Geheimnissen 3,4,5 welche Responses (1. Komponente) möglich sind. Bei $s_0 = 10$ ist nur $w = 5$ möglich.

		Response s_0																		
		-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10
w	3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-
	4	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-
	5	-	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Wenn der Sicherheitsparameter l größer gewählt wird, ist die Wahrscheinlichkeit für so einen Fall vernachlässigbar niedrig. Beim Σ^{GSP}-Protokoll muss zwischen Sicherheit und Kommunikationsaufwand abgewogen werden.

3.6. Logische Verknüpfung von Sigma-Protokollen

Mit Σ^{ϕ} und Σ^{GSP} kann $\mathcal{P}$ nur zeigen, dass er Urbilder von einzelnen Homomorphismen kennt. Es ist aber auch möglich, die Kenntnis von mehreren Urbildern gleichzeitig zu zeigen oder zu zeigen, dass man von mehreren Urbildern mindestens eines kennt.

3.6.1. Und-Verknüpfung

Soll bewiesen werden, dass von mehreren *unabhängigen* Prädikaten jedes gilt, kann das Σ^{AND}-Protokoll verwendet werden. Zwei Prädikate sind unabhängig, wenn sie nicht dieselben geheimen Variablen verwenden.

In Camenisch-Stadler-Notation entspricht dies:
$\text{ZPK}\Big[((\alpha_0,\beta_0,\gamma_0,\ldots),(\alpha_1,\beta_1,\gamma_1,\ldots),\ldots) : \text{Präd}_0(\alpha_0,\beta_0,\gamma_0,\ldots) \wedge \text{Präd}_1(\alpha_1,\beta_1,\gamma_1,\ldots) \wedge \ldots\Big]$

Zum Beispiel sind die Prädikate in $\text{ZPK}\Big[(\alpha,\beta) : x_0 = \phi_0(\alpha) \wedge x_1 = \phi_1(\beta)\Big]$ voneinander unabhängig, während die Prädikate in $\text{ZPK}\Big[(\alpha) : x_0 = \phi_0(\alpha) \wedge x_1 = \phi_1(\alpha)\Big]$ abhängig sind. Würde man versuchen, letzteres mit Σ^{AND} zu beweisen, würde lediglich bewiesen werden, dass $\mathcal{P}$ Urbilder von x_0 und x_1 kennt. Es würde aber nicht bewiesen werden, dass die Urbilder identisch sind. Der Beweis wäre korrekt als $\text{ZPK}\Big[(\alpha) : (x_0,x_1) = \phi_2(\alpha)\Big]$ zu schreiben, wobei $\phi_2(w) = (\phi_0(w),\phi_1(w))$ gilt. Dies kann dann zum Beispiel mit Σ^{ϕ} bewiesen werden.

Es gibt Beweise, bei denen scheinbar abhängige Prädikate auch unabhängig mit Σ^{AND} bewiesen werden können, dies sind jedoch Ausnahmen.

In Σ^{AND} wird jedes Prädikat als eigenes Sigma-Protokoll betrachtet. Wird ein Algorithmus am Σ^{AND}-Protokoll ausgeführt, so wird der entsprechende Algorithmus an allen Teilprotokollen ausgeführt. Dies gilt für P_1, P_2, S, R.

Die Nachrichten $r = (r_0, r_1, \ldots)$ und $s = (s_0, s_1, \ldots)$ sind Tupel der Teilnachrichten. Die Challenge hingegen ist kein Tupel. Sie wird wie auch bei den anderen Protokollen vom Verifier erzeugt und an den Prover übertragen. Das Σ^{AND}-Protokoll überträgt c intern weiter an die Teilprotokolle.

Der Algorithmus V gibt genau dann „korrekt“ zurück, wenn in jedem Unterprotokoll „korrekt“ ausgegeben wird.

Die Eigenschaften Vollständigkeit, Korrektheit und Zero-Knowledge sind genau dann erfüllt, wenn jedes Unterprotokoll sie erfüllt.

Beispiel:

Für $\text{ZPK}\Big[(\alpha,\beta,\gamma) : x_0 = \phi_0(\alpha,\beta) \ \wedge \ x_1 = \phi_1(\gamma)\Big]$ werden zwei Σ^{ϕ}-Protokolle gestartet, diese seien mit Σ_0 und Σ_1 bezeichnet.

- $P_1 : \mathcal{P}$ ruft P_1 an Σ_0 und Σ_1 auf und übermittelt $r = (r_0, r_1)$ an $\mathcal{V}$.
- $\mathcal{V}$ leitet r_0 an Σ_0 und r_1 an Σ_1 weiter.
- $C : \mathcal{V}$ erzeugt eine zufällige Challenge c und überträgt sie auf Σ_0, Σ_1 und $\mathcal{P}$.
- $\mathcal{P}$ leitet c ebenfalls an Σ_0 und Σ_1 weiter.
- $P_2 : \mathcal{P}$ ruft P_2 an Σ_0 und Σ_1 auf und übermittelt $s = (s_0, s_1)$ an $\mathcal{V}$.
- $\mathcal{V}$ leitet s_0 an Σ_0 und s_1 an Σ_1 weiter.
- $V : \mathcal{V}$ ruft V an Σ_0 und Σ_1 auf. Geben beide „korrekt“ zurück, so wird ebenfalls „korrekt“ ausgegeben, anderenfalls ist die Ausgabe „falsch“.

3.6.2. Oder-Verknüpfung

Mit dem Σ^{OR}-Protokoll kann der Prover zeigen, dass von n Prädikaten mindestens eines gilt, ohne jedoch zu offenbaren, welches Prädikat dies ist. Vom Aufbau her ähnelt das Σ^{OR}-Protokoll dem Σ^{AND}-Protokoll. Ein Unterschied ist, dass die Prädikate beim Σ^{OR}-Protokoll abhängige Variablen verwenden dürfen.

In Camenisch-Stadler-Notation entspricht dies:
$\text{ZPK}\Big[(\alpha,\beta,\gamma,\ldots) : \text{Präd}_0(\alpha,\beta,\gamma,\ldots) \ \vee \ \text{Präd}_1(\alpha,\beta,\gamma,\ldots) \ \vee \ \ldots\Big]$

Ebenso wie in Σ^{AND} wird in Σ^{OR} jedes Prädikat als eigenes Sigma-Protokoll betrachtet. Die Algorithmen unterscheiden sich aber von denen in Σ^{AND}.

Die Nachricht $r = (r_0, r_1, r_2, \ldots)$ ist das Tupel der entsprechenden Teilnachrichten. Die Response $s = ((s_0, s_1, s_2, \ldots), (c_1, c_2, \ldots))$ beinhaltet zusätzlich zu den einzelnen Teilresponses noch zu jedem Unterprotokoll, außer dem ersten, eine Challenge, die vom Prover gewählt wurde.

Von den n Prädikaten sei Präd_k jenes, zu dem der Prover eine passende Variablenbelegung kennt.

Die Algorithmen von Σ^{OR} sind wie folgt definiert:

- P_1 : Für Präd_k wird P_1 ausgeführt. In allen anderen Unterprotokollen wird der Simulator S mit einer zufälligen Challenge $c_i \in_R [0, c^+)$ ausgeführt. Diese Challenges werden auch in $state$ gesichert.
- P_2 : Es wird $c_k = c - \sum_{i=0,\ i\neq k}^{n-1} \pmod{c^+}$ berechnet und P_2 für Präd_k ausgeführt.

- S: Für die Unterprotokolle $1,\ldots,n-1$ werden zufällige Challenges c_i gewählt und $c_0 = c - \sum_{i=1}^{n-1} c_i$ berechnet. Dann werden alle Simulatoren ausgeführt.

- R: Hier wird an jedem Unterprotokoll R ausgeführt.

- V: Es wird geprüft, ob $c_1, c_2, \ldots, c_{n-1}$ im Intervall $[0, c^+)$ liegen. Dann wird $c_0 = c - \sum_{i=1}^{n-1} c_i$ berechnet und an jedem Unterprotokoll der Algorithmus V ausgeführt. Nur wenn alle Teilbeweise erfolgreich sind, ist auch V von Σ^{OR} erfolgreich.

Die für Zero-Knowledge-Beweise notwendigen Eigenschaften sind erfüllt:

- **Vollständigkeit:** Der Prover kann für alle Prädikate, die er nicht erfüllen kann, den Simulator einsetzen, da er die Challenges selbst bestimmt. Die Challenge für das verbleibende Prädikat hängt von der Challenge ab, die der Verifier sendet. Der Prover kennt für dieses Prädikat aber die geheime Belegung der Variablen. Deshalb kann der ehrliche Prover den Gesamtbeweis immer erfolgreich führen.

- **Korrektheit:** Wenn ein Prover häufig korrekte Tripel (r,c,s) erstellt, so sind die Tripel für mindestens einen der Teilbeweise auch häufig richtig. Alles weitere folgt dann aus der Korrektheit dieses Teilbeweises.

- **Zero-Knowledge:** Der Verifier kann nicht unterscheiden, für welches Prädikat der Prover die Belegung kennt, da aus Sicht des Verifiers alle Prädikate richtig sind. Deshalb ist die Zero-Knowledge-Eigenschaft in Bezug auf die Besonderheit des Σ^{OR}-Protokolls erfüllt.

Beispiel:

Für $\text{ZPK}\Big[(\alpha,\beta) : x_0 = \phi_0(\alpha) \ \vee \ x_1 = \phi_1(\alpha,\beta) \ \vee \ x_2 = \phi_2(\beta)\Big]$ werden die drei Sigma-Protokolle $\Sigma_0, \Sigma_1, \Sigma_2$ gestartet. Dem Prover sei eine Belegung von α, β bekannt, mit der das mittlere Prädikat $x_1 = \phi_1(\alpha,\beta)$ gilt. Der Protokollablauf ist:

- $P_1 : \mathcal{P}$ wählt zufällige Challenges $c_0, c_2 \in [0, c^+)$, führt S an Σ_0, Σ_2 und P_1 an Σ_1 aus. Die Nachricht $r = (r_0, r_1, r_2)$ wird an $\mathcal{V}$ übertragen.

- $C : \mathcal{V}$ wählt eine Challenge $c \in [0, c^+)$ und überträgt sie an $\mathcal{P}$.

- $P_2 : \mathcal{P}$ berechnet $c_1 = c - c_0 - c_2 \pmod{c^+}$ und führt P_2 an Σ_1 aus.

- $\mathcal{P}$ überträgt die Nachricht $s = ((s_0, s_1, s_2), (c_1, c_2))$ an $\mathcal{V}$.

- $V : \mathcal{V}$ berechnet $c_0 = c - c_1 - c_2 \pmod{c^+}$ und führt V an $\Sigma_0, \Sigma_1, \Sigma_2$ aus. Nur wenn alle „korrekt“ ausgeben, gibt V selbst „korrekt“ aus.

3.7. Arithmetische Abhängigkeiten

3.7.1. Lineare Abhängigkeiten zwischen Geheimnissen

Lineare Abhängigkeiten haben die Form: $\alpha = b \circ \bigcirc_i \beta_i^{e_i}$. b ist eine Konstante, α und β_i sind geheime Variablen (in Camenisch-Stadler-Notation) und $e_i \in \mathbb{Z}$ sind konstante Exponenten. Die Beschreibung „linear“ ist offensichtlicher, wenn die Geheimnis-Gruppe additiv ist, da die Form dann $\alpha = b + \sum_i \beta_i \cdot e_i$ ist.

Die linearen Abhängigkeiten lassen sich durch Umformungen zeigen: Es werden solange die abhängigen Variablen ersetzt, bis alle Abhängigkeiten aufgelöst sind. Anschließend können noch weitere Anpassungen nötig sein; dies hängt von dem konkreten Beweis ab.

Beispiel:

$$\text{ZPK}\Big[(\alpha_{\{0,1,2\}}, \beta_{\{0,1,2\}}) : x_0 = \phi_0(\alpha_0, \beta_0) \ \wedge \ x_1 = \phi_1(\alpha_1, \beta_1) \ \wedge \ x_2 = \phi_2(\alpha_2, \beta_2) \ \wedge \ \alpha_0 = 3 \circ \alpha_1 \circ \alpha_2^2 \ \wedge \ \alpha_1 = 4 \circ \alpha_2\Big]$$

Zunächst werden die Abhängigkeiten nacheinander in der Spezifikation des Beweises ersetzt.

Aus $\text{ZPK}\Big[(\alpha_{\{0,1,2\}}, \beta_{\{0,1,2\}}) : x_0 = \phi_0(\alpha_0, \beta_0) \ \wedge \ x_1 = \phi_1(\alpha_1, \beta_1) \ \wedge \ x_2 = \phi_2(\alpha_2, \beta_2) \ \wedge \ \alpha_0 = 3 \circ \alpha_1 \circ \alpha_2^2 \ \wedge \ \alpha_1 = 4 \circ \alpha_2\Big]$ wird zunächst:

$$\text{ZPK}\Big[(\alpha_{\{0,2\}}, \beta_{\{0,1,2\}}) : x_0 = \phi_0(\alpha_0, \beta_0) \wedge x_1 = \phi_1(4 \circ \alpha_2, \beta_1) \wedge x_2 = \phi_2(\alpha_2, \beta_2) \wedge \alpha_0 = 3 \circ 4 \circ \alpha_2 \circ \alpha_2^2\Big]$$

und dann: $\text{ZPK}\Big[(\alpha_2, \beta_{\{0,1,2\}}) : x_0 = \phi_0(3 \circ 4 \circ \alpha_2^3, \beta_0) \ \wedge \ x_1 = \phi_1(4 \circ \alpha_2, \beta_1) \ \wedge \ x_2 = \phi_2(\alpha_2, \beta_2)\Big]$

Anschließend müssen noch die Konstanten so umgeformt werden, dass die benötigte Homomorphieeigenschaft wiederhergestellt wird:

$$\text{ZPK}\Big[(\alpha_2, \beta_{\{0,1,2\}}) : x_0 \diamond \phi_0(3 \circ 4, e)^{-1} = \phi_0(\alpha_2^3, \beta_0) \wedge x_1 \diamond \phi_1(4, e)^{-1} = \phi_1(\alpha_2, \beta_1) \wedge x_2 = \phi_2(\alpha_2, \beta_2)\Big]$$

Dieser Beweis ist äquivalent zum ursprünglichen Beweis, es wurden aber α_0 und α_1 eliminiert.

Da die linearen Abhängigkeiten durch Umformungen aufgelöst werden, hat dies keinen Einfluss auf die drei Eigenschaften Vollständigkeit, Korrektheit und Zero-Knowledge.

3.7.2. Multiplikative Abhängigkeiten zwischen Geheimnissen

Abhängigkeiten der Form $\text{ZPK}\Big[(w_0, w_1, w_2, p) : x = \phi(w_0, w_1, w_2, p) \ \wedge \ w_2 = w_0^{w_1}\Big]$ werden als multiplikativ bezeichnet, da die Gruppe von w_i in der Regel additiv ist und die Potenz in dieser Gruppe dann der Multiplikation entspricht.

Um Abhängigkeiten dieser Form zu beweisen, werden Commitments x_i zu den einzelnen w_i benötigt.

Beispiel:

Seien $\phi : A \times B \to C, \ (a,b) \mapsto g^a h^b$ und $\phi_x : A \times B \to C, \ (a,b) \mapsto x^a h^b$ zwei Homomorphismen, wobei der zweite einen variablen Generator x verwendet.

$\text{ZPK}\left[(w_0, p_0, w_1, p_1, w_2, p_2) : \bigwedge_{i=0}^{2} x_i = \phi(w_i, p_i) \ \wedge \ w_2 = w_0 \cdot w_1\right]$ ist dann äquivalent zu:

$\text{ZPK}\left[(w_0, p_0, w_1, p_1, w_2, p_2) : \bigwedge_{i=0}^{2} x_i = \phi(w_i, p_i) \ \wedge \ x_2 = \phi_{x_0}(w_1, q)\right]$ mit $q = p_2 - w_1 p_0$.

Beweis der Vollständigkeit:

$\phi_{x_0}(w_1, q) = x_0^{w_1} \cdot h^q = (g^{w_0} \cdot h^{p_0})^{w_1} \cdot h^{p_2 - w_1 \cdot p_0} = g^{w_0 \cdot w_1} \cdot h^{p_2} = g^{w_2} \cdot h^{p_2} = x_2$.

3.8. Intervalle

In manchen Situationen muss gezeigt werden, dass ein Geheimnnis in einem bestimmten Intervall $[a, b]$ liegt. Hierzu gibt es verschiedene Verfahren, die unterschiedlich effizient sind. Je nach Anforderungen ist zu entscheiden, welches Verfahren am geeignetsten ist.

Es sei $(A, +)$ das kartesische Produkt aus Geheimnissen und Zufallswerten, $(B, \circ)$ sei eine endliche zyklische Gruppe und $\phi : A \to B$ ein Homomorphismus, um Commitments zu berechnen. Zu einem Geheimnis w und einem Zufallswert p mit $(w, p) \in A$ sei das Commitment $x = \phi(w, p)$.

Zu beweisen ist: $\text{ZPK}\left[(\alpha, \beta) : x = \phi(\alpha, \beta) \ \wedge \ \alpha \in [a, b]\right]$

3.8.1. Bit Commitments

In [CGH07] wird ein Verfahren vorgestellt, um zu zeigen, dass ein festgelegter nichtnegativer Wert w nicht kleiner ist als ein bestimmter öffentlicher Wert a. Hierzu wird w als Binärzahl der Länge l dargestellt:

$w = w_0 \cdot 2^0 + w_1 \cdot 2^1 + \ldots + w_{l-1} \cdot 2^{l-1}$ mit $w_i \in \{0, 1\}$. Zu jedem Bit w_i wird ein Commitment x_i berechnet.

Anschließend wird mit einem Zero-Knowledge-Beweis gezeigt, dass das erste Bit von w größer als das erste Bit von a ist, oder dass in beiden Zahlen das erste Bit identisch ist und das zweite größer ist, und so weiter.

Dieses Verfahren lässt sich auch so umformen, dass gezeigt wird, dass der festgelegte Wert w nicht größer als ein öffentlicher Wert b ist. Führt man beide Verfahren aus, kann man damit überprüfen, dass w in dem Intervall $[a, b]$ liegt.

Dieses Verfahren eignet sich nur für kleine Intervalle. Je länger das Intervall $[a, b]$ ist, desto mehr Commitments müssen erzeugt werden und desto komplexer wird der Beweis.

3.8.2. Lipmaa-Verfahren

Helger Lipmaa beschreibt in [Lip03] ein Verfahren, um die Behauptung, eine ganze Zahl liege in einem Intervall $[a,b]$, zu beweisen.

Für den Beweis werden zunächst zwei Hilfsgeheimnisse $\bar{w} = w + a^{-1},\ \hat{w} = b + w^{-1}$ eingeführt. w liegt genau dann im Intervall $[a,b]$, wenn $\bar{w}$ und $\hat{w}$ nicht negativ sind. Desweiteren muss die Bildgruppe des Homomorphismus, aus dem x stammt, eine unbekannte Ordnung haben, da ansonsten die Ordnung zu einem negativen $\bar{w}$ oder $\hat{w}$ addiert werden könnte.

Mit ein paar arithmetischen Umformungen gelangt man zu dieser Darstellung des Intervall-Beweises:
$\mathrm{ZPK}\Big[(\alpha,\beta,\gamma,\delta) : \big(x \circ \phi(a^{-1},e) = \phi(\alpha,\beta) \wedge \alpha \geq 0\big) \wedge \big(x^{-1} \circ \phi(b,e) = \phi(\gamma,\delta) \wedge \gamma \geq 0\big)\Big]$
Ein ehrlicher Prover kann $\alpha = \bar{w},\ \beta = p,\ \gamma = \hat{w},\ \delta = p^{-1}$ einsetzen.

Der Intervallbeweis wird darauf reduziert, zu zeigen, dass ein Geheimnis nicht negativ ist. Nach dem Satz von Lagrange lässt sich jede natürliche Zahl als Summe von vier Quadratzahlen schreiben (siehe [Wik10]). Lipmaa gibt einen probabilistischen Algorithmus an, um eine beliebige natürliche Zahl so zu zerlegen.

Lässt sich beweisen, dass sich die Geheimnisse $\bar{w}, \hat{w}$ jeweils als Summe von vier Quadratzahlen darstellen lassen, entspricht dies also dem Beweis, dass $w \in [a,b]$ gilt.

Hierfür muss wiederum bewiesen werden, dass eine Zahl das Quadrat einer anderen Zahl ist. Zu diesen beiden Zahlen wird jeweils ein Commitment benötigt.

Die Quadratzahlen seien mit $\bar{w}'_i$ und $\hat{w}'_i$ bezeichnet, die Wurzeln davon mit $\bar{w}_i$ und $\hat{w}_i$ (jeweils für $i \in [0,3]$). Dies sind insgesamt 16 Geheimnisse, dazu kommen 16 Commitments.

Die Commitments seien $\bar{x}_i = \phi_C(\bar{w}_i, \bar{p}_i),\ \bar{x}'_i = \phi_C(\bar{w}'_i, \bar{p}'_i),\ \hat{x}_i = \phi_C(\hat{w}_i, \hat{p}_i),\ \hat{x}'_i = \phi_C(\hat{w}'_i, \hat{p}'_i)$ für $i \in [0,3]$.

Es bleibt zu zeigen, dass $\bar{w} = \bar{w}'_0 + \bar{w}'_1 + \bar{w}'_2 + \bar{w}'_3 \ \wedge\ \hat{w} = \hat{w}'_0 + \hat{w}'_1 + \hat{w}'_2 + \hat{w}'_3$ und $\bar{w}'_i = \bar{w}_i^2 \wedge \hat{w}'_i = \hat{w}_i^2$ für $i \in [0,3]$ gelten.

Der Intervall-Beweis lässt sich nun wie folgt ausdrücken:

$$\mathrm{ZPK}\Big[((\alpha_i,\beta_i,\gamma_i,\delta_i,\varepsilon_i,\kappa),(\lambda_i,\mu_i,\xi_i,\pi_i,\rho_i,\tau)) : \bigwedge_{i=0}^{3} (\\ (x \circ \phi(a^{-1},e) = \phi(\gamma_0+\gamma_1+\gamma_2+\gamma_3,\kappa) \wedge \bar{x}_i = \phi_C(\alpha_i,\beta_i) \wedge \bar{x}'_i = \phi_C(\gamma_i,\delta_i) \wedge \bar{x}'_i = \phi_{\bar{x}_i}(\alpha_i,\varepsilon_i)) \wedge \\ (x^{-1} \circ \phi(b,e) = \phi(\xi_0+\xi_1+\xi_2+\xi_3,\tau) \wedge \hat{x}_i = \phi_C(\lambda_i,\mu_i) \wedge \hat{x}'_i = \phi_C(\xi_i,\pi_i) \wedge \hat{x}'_i = \phi_{\hat{x}_i}(\lambda_i,\rho_i)))\Big]$$

Dieser Beweis lässt sich mit dem Σ^{AND}-Protokoll aus zwei Σ^{GSP}-Protokollen darstellen. Der Homomorphismus im Σ^{GSP} bildet jeweils 21 Variablen auf 9 Commitments ab. Die Darstellung des kompletten Protokoll-Ablaufs ist ihm Rahmen dieser Arbeit nicht zu präsentieren. Eine Implementierung befindet sich auf der beiliegenden Compact Disc.

Das Lipmaa-Verfahren ist im Gegensatz zu den Bit-Commitments sehr kompliziert, zeichnet sich aber dadurch aus, dass die Komplexität nicht von der Größe des Intervalls abhängt. Bei großen Intervallen ist das Lipmaa-Verfahren also effizienter.

3.9. Nichtinteraktive Beweise und Signaturen

Mit Zero-Knowledge Proofs of Knowledge kann man Signaturen einer beliebigen Nachricht m erstellen. Das Protokoll hierzu ähnelt den interaktiven Sigma-Protokollen. Es wird aber zusätzlich ein Zufallsorakel oder eine kryptographische Hashfunktion benötigt, das anstelle des Verifiers die Challenge erzeugt. Hierdurch kann auch ein arglistiger Verifier keinen Einfluss mehr auf den Prover nehmen.

3.9.1. Zufallsorakel

Ein *Zufallsorakel* ist eine surjektive Funktion $H : A \to B$, die von einer endlichen oder unendlichen Menge A auf eine endliche Menge B abbildet. Das Bild jedes Urbildes ist zufällig und gleichverteilt über der Bildmenge B gewählt und stochastisch unabhängig von allen anderen Urbildern. Eine Umkehrabbildung existiert nicht.

Man kann sich ein Zufallsorakel auch als eine Black Box vorstellen, die jede Anfrage $a \in A$ mit einem zufälligen $b \in B$ beantwortet und sich diese Antwort merkt; wird dieselbe Anfrage mehrfach gestellt, wird jedesmal dieselbe Antwort geliefert.

Das Zufallsorakel ist ein theoretisches Modell, welches in der Praxis nur mit unverhältnismäßigem Aufwand umzusetzen wäre. Hierzu bräuchte man beispielsweise einen perfekten Zufallszahlengenerator, der auf physikalischen Prozessen beruht, und einen großen Speicher, um sich die zufälligen Antworten zu merken. Ferner müsste jede Anwendung, die dieses Zufallsorakel verwenden will, mit diesem kommunizieren können, wodurch sich wieder andere Probleme ergäben. Deshalb werden in der Praxis sogenannte kryptographische Hashfunktionen, zum Beispiel SHA-2, verwendet, die scheinbar zufällige Antworten deterministisch berechnen. In [CGH98] wurde jedoch bewiesen, dass Anwendungen denkbar sind, die mit echten Zufallsorakeln sicher sind, aber durch den Einsatz von Hashfunktionen unsicher werden. Für viele praktische Anwendungen haben diese Ergebnisse allerdings keine Relevanz.

Die Urbildmenge von üblichen kryptographischen Hashfunktionen ist $\{0,1\}^*$, also verschieden lange Folgen von Bits. Die Urbildmenge von einigen Hashfunktionen ist zudem endlich: MD5, SHA-1 und SHA-2 unterstützen Längen von bis zu $2^{64}-1$ Bits, was für praktische Anwendungen ausreicht.

Wird eine Hashfunktion $H : A \to B$ benötigt, die Hashwerte von Elementen einer beliebigen Menge A berechnet, aber nur eine Funktion $H' : C \to D$ zur Verfügung steht, sind eine injektive Abbildung $f : A \to C$ und eine bijektive Funktion $g : D \to B$ nötig. Es gilt dann $H(a) = g(H'(f(a)))$.

Beispiel:

SHA-1 ist eine Funktion $H : \{0,1\}^* \to \{0,1\}^{160}$ und wir benötigen $H' : (\mathbb{Z}^2, \mathbb{Z}_{23}^*) \to \mathbb{Z}_{2^{160}}$. Die Funktionen f, g sind entsprechend $f : (\mathbb{Z}^2, \mathbb{Z}_{23}^*) \to \{0,1\}^*$ und $g : \{0,1\}^{160} \to \mathbb{Z}_{2^{160}}$.

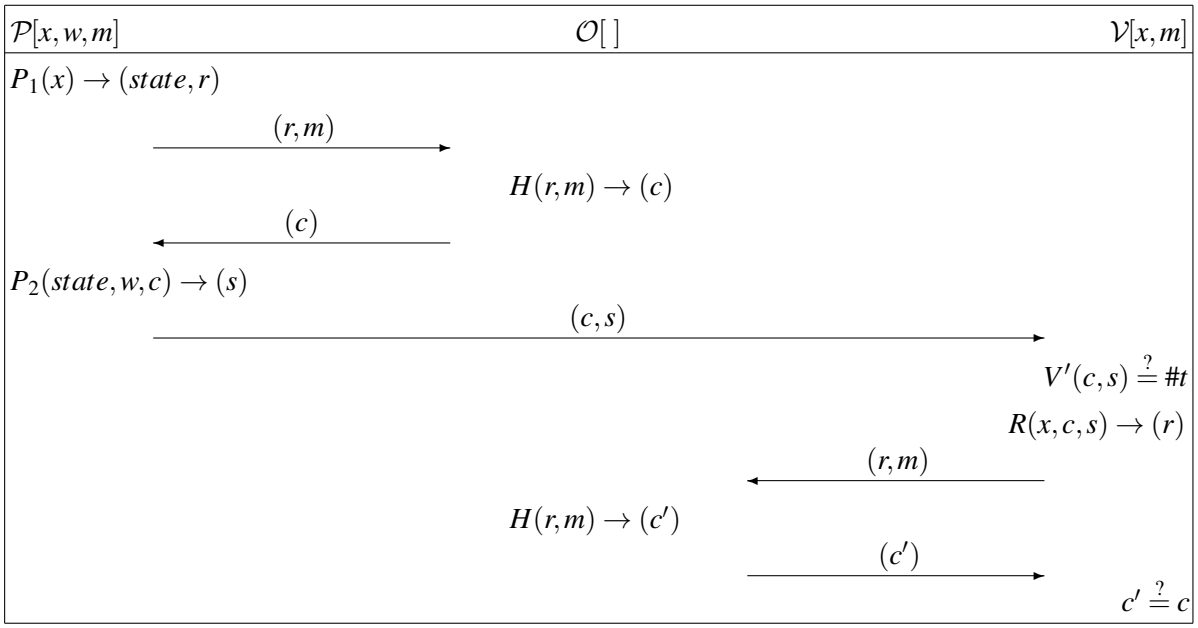
3.9.2. Signaturverfahren

Um eine Signatur von einer Nachricht m zu erstellen, wendet der Prover den Algorithmus $P_1(x) \rightarrow (state, r)$ an, berechnet mit Hilfe des Orakels (oder einer Hashfunktion) die Challenge $c = H(r,m)$ und führt anschließend $P_2(state, w, c) \rightarrow (s)$ aus. Die Signatur zu der Nachricht m ist das Tupel (c,s).

Der Verifier prüft mit $V'(c,s)$, ob z. B. Intervallgrenzen in den Nachrichten eingehalten wurden. Ist dies der Fall, rekonstruiert er mit der Funktion $R(x,c,s) \rightarrow r$ die erste Nachricht des Sigma-Protokolls und die Challenge $c' = H(r,m)$. Gilt $c = c'$, ist (c,s) eine korrekte Signatur der Nachricht m.

Soll keine spezielle Nachricht m signiert, sondern ein Zero-Knowledge-Beweis nichtinteraktiv ausgeführt werden, kann anstatt m der öffentliche Wert x verwendet werden.

Da ein Orakel bzw. eine Hashfunktion niemals arglistig ist, ist bei den nichtinteraktiven Verfahren die Zero-Knowledge-Eigenschaft auch dann sichergestellt, wenn der Verifier arglistig ist. ([S+09, S. 48,50])



4. Zero-Knowledge Compiler

Ein wichtiges Ziel der Arbeit war es, ein Rahmenwerk zu schaffen, um Zero-Knowledge-Beweise auf Rechnern einsetzen zu können.

Um einen konkreten Zero-Knowledge-Beweis zu implementieren, muss dieser zunächst in einer domänenspezifischen Sprache angegeben werden, welche im nächsten Abschnitt erläutert wird. Im darauffolgenden Abschnitt wird auf Konzept und Implementierung des Rahmenwerks eingegangen, mit dem Zero-Knowledge-Beweise dann tatsächlich benutzbar gemacht werden. Im dritten Abschnitt dieses Kapitels wird detailliert erläutert, welche Schritte nötig sind, um aus einem Beweis in Camenisch-Stadler-Notation ein lauffähiges Programm zu erstellen.

4.1. Eingabesprache

4.1.1. Motivation

Um einen Zero-Knowledge-Beweis auf einem Rechner zu implementieren, muss man ihn in einer geeigneten Weise angeben. Dies kann einfach ein Programm in einer schon vorhandenen Programmiersprache (z. B. C oder Java) sein, oder man definiert eine domänenspezifische Sprache, in der sich die Zero-Knowledge-Beweise mit sehr viel weniger Aufwand angeben lassen.

Letzteres hat auch den Vorteil, dass dieselbe Eingabe auf verschiedene Arten verwendet werden kann. So könnten aus derselben Eingabe Implementierungen des Beweises in verschiedenen herkömmlichen Programmiersprachen, als integrierter Schaltkreis oder als Programm für eine Grafikkarte erzeugt werden. Auch könnte Dokumentation in verschiedenen Formaten erzeugt werden.

Deshalb wurde für diese Arbeit eine domänenspezifische Sprache für Zero-Knowledge-Beweise entwickelt; die Anforderungen waren Kompaktheit, große Abdeckung des Problembereichs und die einfache Implementierung der Sprache.

4.1.2. Rahmenbedingungen

Die Eingabe muss in Form einer Textdatei vorliegen. Die Zeichenkodierung der Eingabe ist von dem Betriebssystem abhängig, auf dem die Eingabe verarbeitet wird. Häufige Kodierungen sind ASCII und UTF-8. Wird eine Eingabedatei auf andere Systeme, die zum Beispiel auf EBCDIC basieren, übertragen, so muss eine entsprechende Konvertierung durch den Anwender erfolgen.

Eine gültige Eingabe (`input`) besteht aus einer Folge von Anweisungen (`statement`). Jede Anweisung wird durch ein Semikolon (`;`) abgeschlossen.

In der Eingabe können Kommentare vorkommen, die vom Parser dann nicht weiter verarbeitet werden. Genau wie in der Programmiersprache C sind Kommentare alles zwischen `/*` und dem nächsten `*/` und alles von `//` bis zum Ende der Zeile (`\n`).

Desweiteren werden alle Leerräume (`' '`, `\t`, `\v`, `\f`, `\n`, `\r`, also Leerzeichen, Tabulatoren, Seitenumbrüche, Zeilenumbrüche) ignoriert. Diese Zeichen können und sollten trotzdem zur besseren Lesbarkeit verwendet werden.

Es gibt fünf verschiedene Anweisungstypen, die in den folgenden Abschnitten ausführlich erläutert werden. Am Ende des Kapitels findet sich eine formale Spezifikation der Grammatik.

4.1.3. Bezeichner und Namensräume

In der Eingabe können verschiedene Bezeichner definiert werden. Bezeichner müssen mit einem Buchstaben des englischen Alphabets (`A` bis `Z` und `a` bis `z`) beginnen. Danach folgen beliebig viele Buchstaben, Dezimalziffern (`0` bis `9`) und Unterstriche (`_`). In der Grammatik sind alle Bezeichner vom Typ `ASTRING`.

Bezeichner sind auf unterschiedliche Namensräume aufgeteilt. In einem Namensraum kann derselbe Bezeichner nicht mehrfach definiert werden, in verschiedenen Namensräumen können jedoch mehrere identische Bezeichner existieren. Durch die Syntax der Eingabesprache ist eindeutig festgelegt, in welchem Namensraum ein benutzter Bezeichner zu suchen ist.

Die Namensräume entsprechen den verschiedenen Anweisungstypen. Es gibt Namensräume für Gruppen, Variablen, Homomorphismen und Sigma-Protokolle.

Aus Gründen der Lesbarkeit ist zu empfehlen, sich bei der Wahl von Bezeichnern an eine Konvention zu halten. In dieser Arbeit werden Gruppen mit großen Buchstaben und Ziffern geschrieben, Variablen werden mit kleinen Buchstaben geschrieben, Homomorphismen haben als Präfix `Phi` und Sigma-Protokolle das Präfix `Sigma`.

4.1.4. Gruppen

In der Eingabe muss als erstes definiert werden, in welchen Gruppen gerechnet wird. Es gibt zwei Kategorien von Gruppen: atomare Gruppen (`atomicgroup`) und kartesische Produkte von Gruppen (`tuplegroup`).

Jede Gruppe hat einen Namen, mit dem die Gruppe später referenziert werden kann. Bevor eine Gruppe benutzt werden kann, beispielsweise als Teil eines kartesischen Produkts oder zur Definition von Variablen, muss diese bereits definiert worden sein.

Atomare Gruppen

Syntax: `GruppenName = Gruppentyp ( Parameter 1, Parameter 2, ... );`

Atomare Gruppen haben einen Gruppentyp, der die Ausgestaltung der Gruppenoperationen festlegt.

Die Anzahl und Art der Parameter richten sich nach dem jeweiligen Gruppentyp.

Die folgenden Gruppentypen sind derzeit implementiert:

- `Z` – ganze Zahlen mit Addition

  Syntax: `GruppenName = Z (Minimum, Maximum);`

  Diese Gruppen entsprechen der mathematischen Gruppe $(\mathbb{Z},+)$, also den ganzen Zahlen mit „gewöhnlicher“ Addition als Operation. Es sind zwei Parameter, Minimum und Maximum, anzugeben, die festlegen, aus welchem Intervall Zufallselemente gewählt werden. Trotzdem kann ein Element dieser Gruppe beliebig große ganze Zahlen als Wert haben, weil diese Eigenschaft von einigen Protokollen (zum Beispiel Σ^{GSP}) benötigt wird.

  Beispiel: `G0 = Z (1900, 2010);` definiert die Gruppe $\texttt{G0} = (\mathbb{Z},+)$, deren Elemente ganze Zahlen sind. Die Operation entspricht der Addition, Zufallszahlen werden aus dem Intervall $[1900,2010]$ gleichverteilt gewählt.

- `Z_add_n` – Restklassengruppen mit Addition

  Syntax: `GruppenName = Z_add_n (Modulus);`

  Gruppen mit diesem Typ sind die additiven Restklassengruppen modulo einer natürlichen Zahl n, also $\mathbb{Z}_n$ mit $n \in \mathbb{N}, n \geq 1$. Für diese Gruppe ist genau ein Parameter notwendig; dieser gibt den verwendeten Modulus an. Elemente dieser Gruppen sind die Restklassen $[0],\cdots,[n-1]$ modulo n, die mit den natürlichen Zahlen 0 bis $n-1$ repräsentiert werden. Das Ergebnis jeder Operation liegt daher auch in diesem Intervall $[0,n)$.

  Beispiel: `G1 = Z_add_n (509);` definiert die Gruppe $\texttt{G1} = \mathbb{Z}_{509}$, deren Elemente die Restklassen $[0],\cdots,[508]$ sind.

- `Z_mul_n` – Restklassengruppen mit Multiplikation

  Syntax: `GruppenName = Z_mul_n (Modulus, Untergruppe);`

  Hierbei handelt es sich um die multiplikativen Restklassengruppen modulo einer natürlichen Zahl n, also $(\mathbb{Z}_n^*,\cdot)$ mit $n \in \mathbb{N}, n \geq 2$. Der erste Parameter gibt den verwendeten Modulus an. Der zweite Parameter beschreibt, welche Untergruppe von $\mathbb{Z}_n^*$ benutzt werden soll. Der Wert des zweiten Parameters muss `qr` (Quadratische Reste) oder `default` (ganze Gruppe) sein. Er bestimmt, aus welcher Menge Zufallselemente gewählt werden.

  Beispiel: `G2 = Z_mul_n (2021, default); G3 = Z_mul_n (1019, qr);` definiert zwei Gruppen, $\texttt{G2} = \mathbb{Z}_{2021}^*$ und $\texttt{G3} = QR(\mathbb{Z}_{1019}^*)$.

Wenn der Modulus n nicht prim ist und die Faktorisierung nicht bekannt ist, ist es in der Regel schwierig zu bestimmen, ob ein Element aus $\mathbb{Z}_n^*$ auch in der konkreten Untergruppe ist, welche zum Beispiel durch einen Generator $g \in \mathbb{Z}_n^*$ angegeben wurde. Wenn der Empfänger des Elements dies wissen muss, kann der Sender es mit Hilfe eines Zero-Knowledge-Beweises zeigen.

Von besonderem Interesse ist die Untergruppe der quadratischen Reste modulo n, weshalb diese speziell implementiert wurde.

Kartesische Produkte / Tupelgruppen

Syntax: `GruppenName = ( GruppenName, GruppenName, GruppeName, ... );`

Kartesische Produkte von Gruppen benötigt man unter anderem dann, wenn man mit Gruppenhomomorphismen der Form $\phi : A \times A \to B,\ (a_0, a_1) \mapsto g_0^{a_0} \diamond g_1^{a_1}$ arbeitet.

Beispiel: `GT0 = (G0, G1, G2, G3);` definiert $\texttt{GT0} = \mathbb{Z} \times \mathbb{Z}_{509} \times \mathbb{Z}_{2021}^* \times \mathrm{QR}(\mathbb{Z}_{1019}^*)$

Elemente dieser Gruppe sind entsprechende Tupel. Die Operationen auf Tupeln werden komponentenweise ausgeführt. Es gelten für $a = (a_0, a_1, a_2, a_3) \in \texttt{GT0},\ b = (b_0, b_1, b_2, b_3) \in \texttt{GT0}$ zum Beispiel $a^c = (a_0^c,\ a_1^c,\ a_2^c,\ a_3^c)$ und $a \circ b = (a_0 \diamond b_0,\ a_1 \bullet b_1,\ a_2 \star b_2,\ a_3 \heartsuit b_3)$.

Grundsätzlich können keine zwei gleichen kartesischen Produkte definiert werden. Ist bereits die Gruppe `GT0 = (G0, G1, G2, G3);` definiert, wäre danach die Anweisung `GT1 = (G0, G1, G2, G3);` ungültig, da eine Tupelgruppe mit den Komponenten `(G0, G1, G2, G3)` bereits existiert. Diese Einschränkung ist notwendig, da Gruppentypen in Homomorphismen automatisch ermittelt werden. Gäbe es mehrere Tupelgruppen mit denselben Komponenten, könnte eine eindeutige Gruppe nicht ermittelt werden.

4.1.5. Variablen

Syntax: `GruppenName : Var0, Var1 = Wert, Var2 = (Wert0, Wert1, ...), ...;`

Um Berechnungen durchführen zu können, ist es notwendig, Variablen zu verwenden. Eine Variable ist ein Speicherplatz, in dem ein Element einer Gruppe gespeichert ist. Der Wert einer Variablen kann verändert werden.

Jede Variable hat als Typ eine Gruppe; der Wert einer Variable wird als n-Tupel von ganzen Zahlen (`mpz_t`) dargestellt. Die Bedeutung der jeweiligen Werte hängt von der Gruppe ab, zu der die Variable gehört. Hierzu muss eine injektive Abbildung von der Gruppe auf $\mathbb{Z}^n$ existieren. Sowohl die Abbildung als auch die Umkehrabbildung müssen einfach berechenbar sein. In den implementierten Gruppen ist diese Abbildung trivial: Beispielsweise wird die Restklasse $[17]$ durch das Tupel (17) repräsentiert.

Es gibt zwei Arten von Variablen: benannte Variablen und anonyme Variablen. Letztere werden für temporäre Werte bei Berechnungen genutzt; auf diese wird später noch eingegangen.

Benannte Variablen müssen definiert werden, bevor sie verwendet werden können. Bei der Definition kann man sie auch gleichzeitig initialisieren, also einen Wert setzen. Wurde einer Variablen kein Wert zugewiesen, darf sie solange nicht für Berechnungen verwendet werden, bis ein Wert zugewiesen ist.

Wenn eine Variable während der Deklaration initialisiert werden soll, muss der Wert in der internen Repräsentation als n-Tupel $(w_0, w_1, w_2, \ldots, w_{n-1})$ angegeben werden. Ist $n = 1$, kann auf die Klammern verzichtet werden.

Beispiel: `G0: a, b = 1989;` definiert die Variablen a und b in der Gruppe `G0`. a wird kein Wert zugewiesen, b hat den Wert 1989.

Beispiel: `GT0: c = (2003, 423, 31, 337);` definiert die Variable c in der Gruppe `GT0` $= \mathbb{Z} \times \mathbb{Z}_{509} \times \mathbb{Z}^*_{2021} \times \mathrm{QR}(\mathbb{Z}^*_{1019})$. Der Anfangswert ist $(2003, [423], [31], [337])$.

4.1.6. Homomorphismen

Syntax: `HomName [ QuellGruppe -> ZielGruppe ] = Ausdruck;`

Homomorphismen bilden den bei weitem komplexesten Teil der Eingabesprache. Sie bilden Werte einer Gruppe auf Werte einer anderen Gruppe ab. Sie dienen vorrangig dazu, die von den Sigma-Protokollen benötigten Homomorphismen zu spezifizieren. Im mathematischen Sinne entsprechen die hier definierbaren Abbildungen aber Relationen, da das Bild einer Abbildung nicht ausschließlich von der Eingabe abhängen muss und auch die Homomorphiebedingung verletzt werden kann.

`HomName` gibt den Bezeichner des Homomorphismus an, der Werte von `QuellGruppe` auf `ZielGruppe` abbildet.

Ein *Ausdruck* kann zum Beispiel eine einzelne Variable sein, aber auch eine komplexere Berechnung, in welcher mehrere Ausdrücke mit Hilfe von Operatoren zu einem größeren Ausdruck verknüpft werden. Jeder (Teil-)Ausdruck hat als Typ eine Gruppe, was für bestimmte Operatoren von Bedeutung ist. Der Typ des gesamten Ausdrucks muss die `ZielGruppe` sein.

Die verschiedenen Operatoren bzw. Ausdrucksarten sind bis auf `:` linksassoziativ. Die Liste der Operatoren (mit absteigender Präzedenz) ist:

- `(Ausdruck)`
  Ausdrücke können geklammert werden, um die Assoziativität der Operatoren explizit festzulegen, wie man es auch aus vielen anderen Sprachen gewohnt ist. Weitere Auswirkungen hat die Klammerung nicht.

- Variable
  Dies muss der Name einer Variablen sein, die vorher in der Eingabe definiert wurde. Der Typ ist die Gruppe dieser Variablen.

- `$`
  Auf den Eingabewert des Homomorphismus wird mit `$` zugegriffen. Dies ist eine spezielle Variable, die überall in der Homomorphismusdefinition verfügbar ist. Deren Typ ist `QuellGruppe`.

- `#`, `##`, `###`, ...
  In einer Ausdrucks-Folge (siehe unten) kann mit `#` auf den Wert des vorigen Glieds, mit `##` auf den Wert des Glieds vor jenem, und so weiter zugegriffen werden. Diese spezielle Variable darf deshalb nur verwendet werden, wenn es diese Glieder gibt. Der Typ ist der jeweilige Typ des Ausdrucks.
  Beispiel: `x : (a + $ : g ^ #, #) : - # - b : ###`; hier steht das erste `#` für `a + $`, das zweite steht für `x` und das dritte für den gesamten Klammerausdruck. `###` steht für `x`.

- `?Gruppe`
  Erzeugt ein Zufallselement aus der Gruppe `Gruppe`. Der Ergebnistyp ist `Gruppe`.

- `<Gruppe`
  Entspricht dem Minimalwert in der Gruppe `Gruppe`. Der Ergebnistyp ist `Gruppe`.

- `>Gruppe`
  Entspricht dem Maximalwert in der Gruppe `Gruppe`. Der Ergebnistyp ist `Gruppe`.

- `~Gruppe`
  Entspricht der Identität in der Gruppe `Gruppe`. Der Ergebnistyp ist `Gruppe`.

- `Gruppe {32,-11,...}`
  Mit dieser Syntax lassen sich Konstanten in einer bestimmten Gruppe definieren. Stattdessen könnte man auch eine Variable definieren und initialisieren und diese als Ausdruck verwenden.

- `[Ausdruck, Ausdruck, ...]`
  Erstellt ein Tupel mit den Resultaten der entsprechenden Ausdrücke. Der Typ dieses Ausdrucks ist das kartesische Produkt, das sich aus den einzelnen Typen zusammensetzt.

- `HomName(Ausdruck)`
  Wendet den Homomorphismus `HomName` auf das Ergebnis des Ausdrucks `Ausdruck` an. Jener muss den Typ der Quellgruppe des Homomorphismus haben. Der Ergebnistyp entspricht dem des Homomorphismus. Der referenzierte Homomorphismus muss bereits definiert sein und darf auch nicht derselbe sein, zu dem dieser Ausdruck gehört.

- `Ausdruck.123`
  Wenn ein Ausdruck `Ausdruck` ein Tupel mit n Gliedern ist, kann mit `Ausdruck.x` auf die einzelnen Tupelglieder zugegriffen werden, wobei $x \in [0, n)$ gilt. Ist dies wiederum ein Tupel, kann diese Prozedur fortgesetzt werden, z. B. `Ausdruck.3.2.0`

- `- Ausdruck`
  Der unäre Minus-Operator entspricht der Inversenbildung in der Gruppe. Der Typ dieses inversen Ausdrucks ist derselbe wie der des Ausdrucks.

- `<Gruppe> Ausdruck`
  Dies ist der Cast-Operator. Der Typ von `Ausdruck` wird explizit zu `Gruppe` verändert. Dies wird zum Beispiel benötigt, wenn mit Gruppen von unendlicher Ordnung gerechnet wird und sich durch die Berechnungen die Wertegrenzen verändern. Die ursprüngliche und die neue Gruppe müssen jeweils dieselbe Anzahl an atomaren Komponenten enthalten.
  Beispiel: `a + b` wobei `a,b` in $[0,10]$ liegen. Die Summe liegt jedoch im Intervall $[0,20]$, wozu eine neue Gruppe definiert werden muss und das Ergebnis zu dieser umzuwandeln ist.

- `Ausdruck ^ Exponent`
  Mit dem Potenz-Operator wird `Ausdruck` mit dem `Exponenten` potenziert. Der Exponent ist entweder eine ganze Zahl in Dezimaldarstellung oder ein atomarer Ausdruck.

- `Ausdruck + Ausdruck`
  Mit dem Plus-Operator wird die Gruppenoperation mit den beiden Ausdrücken ausgeführt. Diese müssen in derselben Gruppe sein.

- `Ausdruck - Ausdruck`
  Mit dem binären Minus-Operator wird die Gruppenoperation mit dem ersten Ausdruck und dem Inversen des zweiten Ausdrucks durchführt. Die beiden Ausdrücke müssen in derselben Gruppe sein.

- `Ausdruck : Ausdruck`
  Mit diesem Operator können mehrere Ausdrücke nacheinander ausgeführt werden (von links nach rechts). Als einziger Operator ist dieser rechtsassoziativ, d. h. `A0 : A1 : A2 : A3` entspricht `(A0 : (A1 : (A2 : A3)))`. Mit Hilfe des #-Operators kann auf das Ergebnis der vorigen Ausdrücke zugegriffen werden (siehe oben).

Die Mächtigkeit dieser Ausdrücke ist begrenzt, da keinerlei Rekursion oder Schleifen möglich sind. Dies hat den Nachteil, dass sich hiermit nicht alles berechnen lässt, was in der Theorie berechenbar wäre. Der große Vorteil ist jedoch, dass sich jeder Homomorphismus als lineares Programm darstellen lässt, wodurch Implementierungen der Sprache sehr viel einfacher sind.

Beispiele:

- `PhiId [ G -> G ] = $;` entspricht: $\phi_{Id} : G \to G,\ a \mapsto a$.
- `PhiSwap [ AB -> BA ] = [$.1,$.0];` entspricht: $\phi_{Swap} : A \times B \to B \times A,\ (a,b) \mapsto (b,a)$.
- `PhiDup [ A -> AA ] = [$,$];` entspricht: $\phi_{Dup} : A \to A \times A,\ a \mapsto (a,a)$.
- `PhiFoo [ B -> AA ] = PhiSwap([x,$]) : PhiDup(#.1);` entspricht:
  $\phi_{Foo} : B \to A \times A,\ b \mapsto (x,x)$ mit einer Variablen $x \in A$.

- `PhiRnd [ M -> A ] = ?A;` entspricht $\phi_{Rnd} : M \to A,\ m \mapsto a$ mit $a \in_R A$. Dies gibt eine Zufallszahl zurück und ist somit kein Homomorphismus im mathematischen Sinne.
- `PhiComm [ W -> CWP ] = ?P : [ [g^$, h^#] : #.0 + #.1, [$, #]];` berechnet Commitments mit einer Zufallskomponente: $\phi_{Comm} : W \to C \times (W \times P),\ w \mapsto (g^w \cdot h^p, (w, p))$ mit $p \in_R P$. Auch dies ist kein Homomorphismus im mathematischen Sinne.

4.1.7. Sigma-Protokolle

Syntax: `SigmaName = Protokollname [ Parameter 1, Parameter 2, ... ];`

Sigma-Protokolle verwenden eine ähnliche Syntax wie die atomaren Gruppen, nur werden hier eckige anstatt runder Klammern verwendet. Ein nicht-syntaktischer Unterschied ist der Protokollname: Versucht man, eine Gruppe mit einem Sigma-Protokoll oder andersherum zu definieren, führt dies zu einem Fehler, weil Gruppen und Sigma-Protokolle verschiedene Namen haben.

Der Protokollname steht für das verwendete Sigma-Protokoll, also beispielsweise `SigmaPhi` oder `SigmaAND`. Die Parameter hängen vom konkreten Protokoll ab.

SigmaPhi

Syntax: `SigmaName = SigmaPhi [ HomName, PublicVar, SecretVar, c+ ];`

`HomName` steht für einen Homomorphismus; `PublicVar` und `SecretVar` für Variablen, die die öffentlichen beziehungsweise geheimen Werte speichern. `c+` steht für das Intervall $C = [0, c^+)$, aus dem Challenges gewählt werden können.

Mit diesem Protokoll beweist der Prover gegenüber dem Verifier, dass sein Wert in `SecretVar` die Gleichung `PublicVar = HomName (SecretVar)` erfüllt.

Aus Sicherheitsgründen muss `SecretVar` zu einer endlichen Gruppe gehören, in der Zufallselemente gleichverteilt erzeugt werden können.

Beispiel: `SigmaD = SigmaPhi [ PhiDup, aa, a, 713 ];` entspricht: $\mathrm{ZPK}\left[(\alpha) : aa = \phi_{Dup}(\alpha)\right]$.

SigmaGsp

Syntax: `SigmaName = SigmaGsp [ HomName, PublicVar, SecretVar, c+, l ];`

Die Syntax und Bedeutung der Parameter entsprechen SigmaPhi. Es ist jedoch ein weiterer Wert `l` anzugeben; dieser legt einen Sicherheitsparameter fest, der bestimmt, wie dicht die Wahrscheinlichkeitsverteilung von Zufallselementen an der Gleichverteilung liegt. Übliche Werte liegen zwischen 80 und 100.

Im Gegensatz zu SigmaPhi darf `SecretVar` zu einer unendlichen Gruppe gehören. Die höhere Sicherheit des GSP-Protokolls wird aber mit einem höheren Aufwand erkauft: die Algorithmen sind komplizierter, die verwendeten Zahlen sind um Größenordnungen höher und der Kommunikationsaufwand ist ein Vielfaches.

SigmaAND

Syntax: `SigmaName = SigmaAND [ SigmaName, SigmaName, SigmaName, ... ];`

Als Parameter wird eine nichtleere Liste von bereits definierten Sigma-Protokollen erwartet. Wird ein Beweis mit diesem Sigma-Protokoll durchgeführt, so werden alle Unterbeweise durchgeführt.

Der Beweis ist genau dann erfolgreich, wenn er für **alle** Unterbeweise erfolgreich geführt wird.

Im Unterschied zu Σ^{ϕ},Σ^{GSP}gibt es hier keine Angabe von c^+. Als c^+ wird das Minimum aller c^+ in den Unterbeweisen verwendet; die Unterbeweise verwenden auch nur dieses Minimum. Der Anwender hat darauf zu achten, dass Beweise oft genug wiederholt werden.

Beispiel: `SigmaX = SigmaAnd [ SigmaD, SigmaU, Sigma2 ];` entspricht:
$\text{ZPK}\left[(\alpha,\beta,\gamma) : aa = \phi_{Dup}(\alpha) \wedge \text{Präd}_U(\beta) \wedge \text{Präd}_2(\gamma)\right]$

SigmaOR

Syntax: `SigmaName = SigmaOR [ SigmaName, SigmaName, SigmaName, ... ];`

Als Parameter wird genau wie bei SigmaAND eine nichtleere Liste von bereits definierten Sigma-Protokollen erwartet. Wird ein Beweis mit dem SigmaOR-Protokoll geführt, wird einer der Unterbeweise normal geführt und alle anderen simuliert.

Ein Beweis ist genau dann erfolgreich, wenn er für **mindestens einen** Unterbeweis erfolgreich geführt wird. Aus Sicht von $\mathcal{V}$ müssen jedoch alle Beweise korrekt geführt werden; dieser kann nicht unterscheiden, welcher Beweis korrekt war.

4.1.8. Grammatik

In den Abschnitten zuvor wurde die Grammatik stückweise und nicht formal angegeben. Die Grammatik der Eingabesprache ist in Abbildung 4.1 noch einmal in EBNF dargestellt. Das Startsymbol ist `input`.

Die Anzahl und Art der Parameter für die atomaren Gruppen und Sigma-Protokolle sind nicht Teil der Grammatik; diese werden durch die entsprechenden Gruppen und Protokolle festgelegt, um die Erweiterbarkeit der Sprache zu vereinfachen.

```
input = { statement, ";" };
statement = atomicgroup | tuplegroup | varlist | sigma | hom;
atomicgroup = ASTRING, "=", ASTRING, "(", stringlist, ")";
tuplegroup = ASTRING, "=", "(", astringlist, ")";
varlist = ASTRING, ":", var, { var };
sigma = ASTRING, "=", ASTRING, "[", stringlist, "]";
hom = ASTRING, "[", ASTRING, "-", ">", ASTRING, "]", "=", expr;
var = ASTRING, [ "=", ( signednum | "(", signednumlist, ")" ) ];
expr = expr, ":", expr | addexpr;
addexpr = addexpr, ( "+" | "-" ), addexpr | powexpr;
powexpr = powexpr, "^", ( powexpr | NUMBER ) | castexpr;
castexpr = "<", ASTRING, ">", castexpr | unaryexpr;
unaryexpr = "-", unaryexpr | dotexpr;
dotexpr = dotexpr, ".", NUMBER | atomexpr;
atomexpr = "(", expr, ")" | "$" | "#", { "#" } | ASTRING |
           ASTRING, "(", expr, ")" | '[', expr, { expr }, ']' |
           ASTRING, "{", NUMBER, { NUMBER }, "}" |
           ( "?" | "<" | ">" | "~" ), ASTRING;
signednumlist = signednum, { ",", signednum };
astringlist = ASTRING, { ",", ASTRING };
stringlist = string, { ",", string };
string = STRING | ASTRING | signednum;
signednum = NUMBER | "-", NUMBER;
NUMBER = digit, { digit };
STRING = "_", { digit | alpha | "_" } |
         digit, { digit | alpha | "_" }, ( alpha | "_" ),
                { digit | alpha | "_" };
ASTRING = alpha, { digit | alpha | "_" };
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
alpha = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
        "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
        "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"       |
        "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
        "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
        "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";
```

Abbildung 4.1.: Grammatik der Eingabesprache in EBNF

EBNF

Die Erweiterte Backus-Naur-Form (EBNF) ist ein nach ISO/IEC 14977:1996(E) [ISO96] standardisiertes Format, um kontextfreie Grammatiken formal anzugeben. Eine Grammatik gibt an, welche Wörter zu einer Sprache gehören.

Ausgehend von einem Startsymbol werden rekursiv alle syntaktisch gültigen Wörter der Sprache mit Hilfe von Produktionsregeln abgeleitet. Nach jedem Ableitungsschritt besteht das aktuelle Wort aus einer Folge von Symbolen. Symbole können terminal und nichtterminal sein. Das Startsymbol ist ein Nichtterminalsymbol.

Die gültigen Wörter der Sprache bestehen aus einer Folge von Terminalsymbolen; dies sind Symbole die nicht weiter abgeleitet werden können.

Nichtterminalsymbole müssen solange rekursiv zu Folgen anderer Symbole abgeleitet werden, bis das Wort nur noch aus Nichtterminalsymbolen besteht.

Grammatiken bestehen aus einer Folge von Produktionsregeln. Jede Produktionsregel leitet ein Nichtterminalsymbol zu einer Folge/Konkatenierung von Symbolen (terminal oder nichtterminal) ab.

Produktionsregeln haben die Form:

```
Nichtterminalsymbol = Definition ;
```

Definitionen bestehen aus diesen Elementen:

- `"a"` gibt das Terminalsymbol ‘a’ an.
- `xyz` gibt ein Nichtterminalsymbol an.
- `def0, def1` ist die Konkatenierung von zwei Definitionen. `"a", "b"` steht also für die Zeichenkette ‘ab’.
- `def0 | def1` definiert eine Alternative. `"a" | "b"` ist entweder ‘a’ oder ‘b’.
- `[ def ]` ist eine Option. `"a", [ "b" ], "c"` ist eine der Zeichenketten ‘abc’, ‘ac’.
- `{ def }` ist der Wiederholungsoperator. `"a", { "b" }, "c"` steht für die Zeichenketten ‘ac’, ‘abc’, ‘abbc’, ‘abbbc’ und so weiter.
- `( def )` gruppiert mehrere Definitionen. Dies wird benötigt, weil der Konkatenierungsoperator `,` eine höhere Bindungsstärke als der Alternativen-Operator `|` hat. `"a", "b" | "c"` ist somit äquivalent zu `( "a", "b" ) | "c"`, was für ‘ab’ oder ‘c’ steht. `"a", ( "b" | "c" )` hingegen ist ‘ab’ oder ‘ac’.

Eine ausführliche Beschreibung der EBNF ist im Standard[ISO96] nachzulesen.

4.2. Implementierung

Die Referenzimplementierung besteht aus einem LALR(1)-Parser[1], der die Eingabesprache verarbeitet und hieraus eine Datenstruktur erzeugt, und einem Interpreter, der anhand dieser Datenstruktur die Beweise führt und dazu einige Hilfsfunktionen beinhaltet.

Der Parser ist mit Hilfe von Bison in ANSI C89 geschrieben; der Interpreter ist in C++ implementiert.

[1] Look Ahead Left-to-right, Untermenge der kontextfreien Sprachen

4.2.1. Verwendete Software

Bison

Um eine Sprache zu verarbeiten, wird ein Parser benötigt. Diesen vollständig per Hand zu programmieren, ist sehr aufwendig und fehlerträchtig. Zudem müsste für jede Spracherweiterung der Parser neu geschrieben werden.

Bison ist ein *Parser Generator*, der aus einer kontextfreien Grammatik in Backus-Naur-Form einen LALR(1)-Parser konstruiert. Der erzeugte Parser kann als C-Modul in andere Projekte eingebunden werden.

Es handelt sich bei Bison um eine zu yacc (**Y**et **A**nother **C**ompiler **C**ompiler) kompatible Entwicklung des GNU-Projekts. Bison darf nach den Regeln der GNU General Public License, Version 3, genutzt werden; der erzeugte Parser hingegen unterliegt keiner vorgegebenen Lizenz und darf vom Autor der Grammatik beliebig verwendet werden.

Offizielle Webseite: `http://www.gnu.org/software/bison/`

GMP (The GNU Multiple Precision Arithmetic Library)

Kryptographische Anwendungen benötigen für viele Berechnungen sehr große Zahlen. Übliche Rechner können nur Zahlen von Größen bis maximal 32 bzw. 64 Bits verarbeiten, benötigt werden aber auch Zahlen mit deutlich mehr als 1000 Bits. Hierzu gibt es mehrere Bibliotheken, die die Verarbeitung beliebig großer Zahlen unterstützen.

Die Wahl fiel auf die GMP (GNU Multiple Precision Arithmetic Library). Diese Bibliothek stellt alle Grundoperationen (Addition, Subtraktion, Multiplikation, Division, schnelle Exponentation, Modulo-Rechnung, etc.) bereit, um mit beliebig großen Zahlen zu rechnen. Zudem enthalten sind auch Funktionen zur Erzeugung von Zufallszahlen sowie wichtige zahlentheoretische Algorithmen wie die Berechnung des größten gemeinsamen Teilers, probabilistische Primzahlentests und Inversenbildung von multiplikativen Restklassengruppen. Die Bibliothek ist auf vielen Plattformen einsetzbar und enthält für viele Prozessoren optimierten Code.

Wie Bison ist auch die GMP Teil des GNU-Projekts; die GMP steht unter der GNU Lesser General Public License, wodurch auch Binärpakete verbreitet werden dürfen, die die GMP-Bibliothek verwenden.

Offizielle Webseite: `http://gmplib.org/`

CMake / CTest

CMake ist ein plattformübergreifendes *build system*, das vergleichbare Programme wie GNU Autoconf, Automake und Libtool ersetzt. Es kann unter anderem Makefiles für POSIX-Systeme und Projektdateien für Microsoft Visual C++ erzeugen. Es stellt damit ein automatisiertes Verfahren bereit, um aus dem Quellcode ausführbare Dateien zu erzeugen.

Mit dem dazugehörigen CTest lassen sich Testabläufe automatisieren.

Entwickelt wird CMake von Kitware, Inc. Die Lizenz von CMake entspricht der „3-Klausel-BSD-Lizenz“.

Offizielle Webseite: `http://www.cmake.org/`

Doxygen

Zur automatischen Erzeugung von Dokumentation wird Doxygen benutzt. Doxygen analysiert den Quellcode und spezielle im Quellcode enthaltene Kommentare und generiert hieraus unter anderem die API-Dokumentation im HTML-Format, Klassendiagramme, etc.

Doxygen steht unter der GNU General Public License und wird von Dimitri van Heesch et al. entwickelt.

Offizielle Webseite: `http://www.doxygen.org/`

Subversion

Als System zur Versionsverwaltung kommt Apache Subversion zum Einsatz. Da an dem Projekt mehrere Personen beteiligt waren, ist eine Versionsverwaltung zur Koordinierung hilfreich.

Subversion unterliegt der Apache License, Version 2.0 und gilt damit wie die andere Software auch als freie Software.

Offizielle Webseite: `http://subversion.apache.org/`

4.2.2. Datenstruktur

In der Eingabe werden Gruppen, Variablen, Homomorphismen und Sigma-Protokolle definiert; die durch den Parser erzeugte Datenstruktur besteht aus vier Arrays, in denen die formalen Beschreibungen eben dieser Definitionen enthalten sind.

Für jede Gruppe wird in die Struktur ein Objekt gespeichert, welches angibt, wie die Gruppe heißt und ob sie eine atomare Gruppe oder eine Tupelgruppe ist. Von atomaren Gruppen werden zudem der Typ (z. B. `Z_add_n`) und die Parameter hinterlegt. Von Tupelgruppen werden die Teilgruppen gespeichert.

Pro definierter Variable wird ein Objekt in die Struktur gespeichert, die den Namen, die Gruppe und die Initialisierung der Variable angibt. Eine Definition wie `G:a,b,c; H:d,e;` würde also fünf Objekte erzeugen.

Die Objekte für Sigma-Protokolle enthalten den Typ, den Namen und die Parameter des Protokolls.

Homomorphismen werden mit ihrem Namen und den Bild- und Urbildgruppen in die Struktur übertragen. Außerdem ist noch eine lineare Liste von Befehlen enthalten. Jeder Befehl gibt an, in welcher Gruppe mit welchen Variablen/Werten welche Operation auszuführen ist, wobei nur atomare Gruppen Verwendung finden. Sollte die Operation eigentlich auf einem Tupel ausgeführt werden, wird stattdessen für jede Komponente ein Befehl erzeugt. Jeder Befehl hat genau eine Ausgangsvariable und null bis zwei Eingangsvariablen. Die Liste von erzeugten Befehlen ist in etwa mit einem Assembler-Programm vergleichbar, welches keinerlei Sprünge und Verzweigungen enthält.

Die möglichen Operationen sind:

- `id` – Setzt die Ausgangsvariable auf das neutrale Element der Gruppe.
- `min` – Setzt die Ausgangsvariable auf das kleinste Element der Gruppe.
- `max` – Setzt die Ausgangsvariable auf das größte Element der Gruppe.
- `inv` – Berechnet das Inverse der Eingangsvariable und speichert das Ergebnis in der Ausgangsvariablen.
- `op` – Wendet die Gruppenoperation auf den beiden Eingangsvariablen an.
- `iop` – Wendet die Gruppenoperation auf der ersten Eingangsvariablen und dem Inversen der zweiten an.
- `pow` – Berechnet die Potenz der ersten Einangsvariablen mit der zweiten Variablen als Exponent.
- `ipow` – Dasselbe wie `pow`, nur dass der Exponent negiert wird.
- `rnd` – Erzeugt einen Zufallswert in der Ausgangsvariablen.
- `set` – Kopiert die Eingangsvariable in die Ausgangsvariable.
- `cast` – Kopiert die Eingangsvariable in die Ausgangsvariable. Die Gruppen dürfen jedoch verschieden sein.

4.2.3. Parser

Die Aufgabe des Parsers ist es, die Eingabedatei in die interne Datenstruktur zu übersetzen, die dann von anderen Modulen (zum Beispiel dem Interpreter) weiterverarbeitet wird.

Der Parser selbst lässt sich noch in mehrere Teile untergliedern.

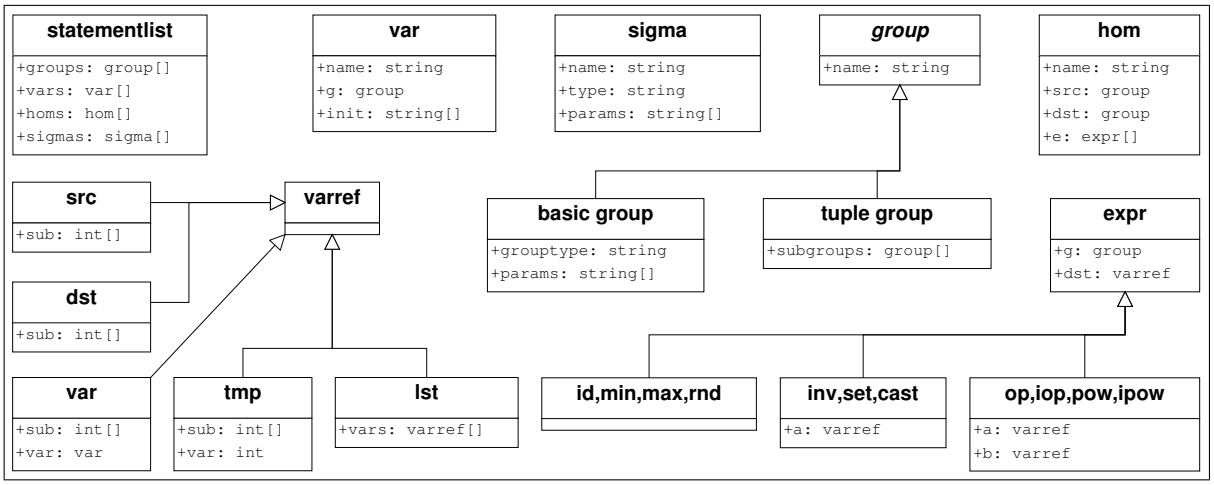


Abbildung 4.2.: Datenstruktur

Grammatik / LALR-Parser

In der Quellcode-Datei `zkparser_grammar.y` ist die Grammatik der Eingabesprache in einer der EBNF ähnlichen Notation angegeben. Darüber hinaus enthält diese Datei Anweisungen zur Speicherverwaltung und zu jeder Produktionsregel ein kleines Codefragment.

Bison übersetzt die Datei `zkparser_grammar.y` in ein C-Modul, welches dann den eigentlichen LALR(1)-Parser enthält. Das Herzstück hiervon ist die Funktion `zkparse`, die einen endlichen Kellerautomaten implementiert.

Dieser Automat liest mit dem Lexer eine Folge von Terminalsymbolen. Immer wenn alle für eine Produktionsregel nötigen Symbole eingelesen sind und entschieden ist, welche Produktionsregel anzuwenden ist, wird die Symbolfolge zu einem einzelnen Nichtterminalsymbol zusammengefasst. Wie genau dies geschieht, ist durch die kurzen Codefragmente in der Grammatik angegeben. Viele Produktionsregeln müssen hierzu komplexere Hilfsfunktionen aufrufen, die dann für das Zusammenfassen der Symbole zu einem neuen Nichtterminal zuständig sind.

Von besonderem Interesse sind die fünf Produktionsregeln, die letztlich die Anweisungen (Definitionen von atomaren Gruppen, Tupelgruppen, Variablen, Homomorphismen, Sigma-Protokollen) verarbeiten und zur Datenstruktur hinzufügen, welche schließlich als Resultat ausgegeben wird.

Lexer

Der Lexer (kurz für lexikalischer Scanner) liest einzelne Zeichen der Eingabe, entfernt Kommentare aus der Eingabe und gruppiert mehrere zusammenhängende Eingabezeichen zu Terminalsymbolen. Jeder Aufruf des Lexers gibt genau ein Terminalsymbol oder `EOF`[2] zurück. Realisiert ist der Lexer mit zwei hintereinandergeschalteten deterministischen endlichen Mealy-Automaten; einer filtert Kommentare aus und der andere fügt Folgen von Zeichen zu Terminalsymbolen zusammen.

[2] End Of File / Ende der Eingabe

Jedes Terminalsymbol hat einen Typ und einen optionalen Inhalt. Die möglichen Typen sind:

- `NUMBER` – Zeichenkette aus Dezimalziffern,
- `ASTRING` – Zeichenfolge aus einem Buchstaben (a-z, A-Z) gefolgt von beliebig vielen Buchstaben, Dezimalziffern und Unterstrichen,
- `STRING` – Alle Zeichenketten aus Dezimalziffern, Buchstaben und Unterstrichen, die weder `NUMBER` noch `ASTRING` sind,
- `INVALID` – Ungültige Eingabe; tritt auf bei unerwartetem EOF in einem Kommentar oder bei Eingabe eines Null-Zeichens.
- sonstige – Alle anderen Zeichen (Satzzeichen, Sonderzeichen, etc.), die nicht Teil von Zeichenketten sind, haben das Zeichen selbst als Typ.

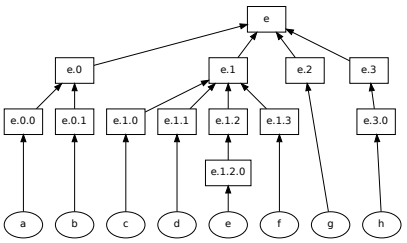
Der Inhalt der drei Zeichenketten-Typen `NUMBER`, `ASTRING` und `STRING` ist jeweils die Zeichenkette.

Konvertierung der Homomorphismen

Die meisten Hilfsfunktionen implementieren triviale Listenoperationen, Konstruktoren und Destruktoren, auf die hier nicht weiter eingegangen werden soll.

Von Interesse ist jedoch die Verarbeitung der Homomorphismen. Zunächst aber ein zum Verständnis notwendiger Exkurs:

Exkurs: (Geordnete) Tupel in der Mathematik lassen sich als Graphen darstellen. Zum Beispiel entspricht dem Tupel $((a,b),(c,d,(e),f),g,(h))$ der nebenstehende Graph. Die elliptischen Knoten stehen für die atomaren Komponenten in dem Tupel, während die rechteckigen Knoten zusammen einen Baum bilden, der die Struktur des Tupels wiedergibt.



Jeder (Teil-)Ausdruck in einem Homomorphismus wird als ein Tupel betrachtet und lässt sich dementsprechend auch in dieser Form als Graph darstellen. Die atomaren Komponenten der Ausdrücke können jedoch deutlich komplexer sein. Jede dieser Komponenten wird als gerichteter, zyklenfreier Graph dargestellt, dessen Wurzelknoten mit dem entsprechenden rechteckigen Knoten des Baums verbunden ist. Die Knoten des Graphs stehen entweder für eine Variable oder für eine der Gruppenoperationen. Jeder Knoten hat somit null bis zwei eingehende Kanten. Eine Besonderheit ist hier, dass nur in atomaren Gruppen gerechnet wird. Soll in Tupelgruppen gerechnet werden, müssen die Komponenten des Tupels aufgeteilt werden. Wenn ein Tupel mehr als eine Komponente hat, können die Graphen der verschiedenen Komponenten miteinander verbunden sein. Hierdurch kann beispielsweise derselbe Zufallswert in verschiedene Rechnungen eingehen, wie sich in den nächsten Abbildungen erkennen lässt.

Wenn eine Variable atomar ist, ist der dazugehörige Graph entsprechend klein. Tupel hingegen müssen in die einzelnen Komponenten aufgeteilt werden. Zufallselemente, Minima, Maxima und die Identität werden ähnlich dargestellt. Hier wird jeweils die entsprechende Operation mit jeder atomaren Gruppe ausgeführt.

Konstanten der Form `Gruppe {32,-11,...}` werden als anonyme Variablen angelegt und dann als Variable weiterverwendet.

Andere Operationen, zum Beispiel die Gruppenoperation, benötigen mehr Aufwand: Hier müssen mehrere Graphen zusammengefügt werden. Für die Gruppenoperation + werden komponentenweise, also für je einen Knoten beider Graphen, neue +-Knoten eingefügt, die als Eingänge diese beiden Knoten haben. Für die Tupel-Syntax `[Ausdruck, Ausdruck, ...]` werden die Graphen einfach mit einer neuen Wurzel zusammengefügt. Die Umkehrung hiervon, z. B. `v.1`, wird dadurch erreicht, dass ein Teil des Graphen wieder gelöscht wird.

Die Verkettung von Ausdrücken mittels : und # geschieht dadurch, dass beim Auftreten des :-Operators der Graph des linksstehenden Ausdrucks in einer Liste gespeichert wird und mit `#, ##, ...` auf die Listenelemente zugegriffen werden kann. Der Baum (rechteckige Knoten) wird dupliziert, während der restliche Graph lediglich referenziert wird. Hierdurch können dieselben Werte mehrfach verwendet werden.

Zum Aufruf weiterer Homomorphismen wird der gesamte Graph des aufgerufenen Homomorphismus kopiert und hierbei die die Variable `$` durch den Eingangsparameter ersetzt. Dies ist notwendig, damit bei zwei Aufrufen desselben Homomorphismus die berechneten Werte voneinander unabhängig sind.

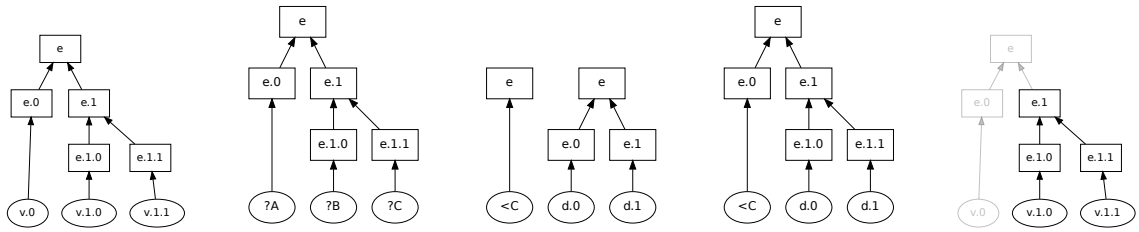
Die Graphen in Abbildung 4.3 sollen diese Konzepte noch einmal veranschaulichen.

Nachdem der gesamte Ausdruck in einen einzigen Graphen überführt wurde, könnten an dieser Stelle Optimierungen durchgeführt werden. So könnten zum Beispiel gleiche Knoten zusammengefügt werden, aber auch die Reihenfolge von Operationen verändert werden, solange das Ergebnis dasselbe bleibt. Anhand der Graphenstruktur könnte auch eine automatische Parallelisierung der Berechnungen erfolgen. In der aktuellen Version sind diese Optimierungen jedoch noch nicht vorhanden.

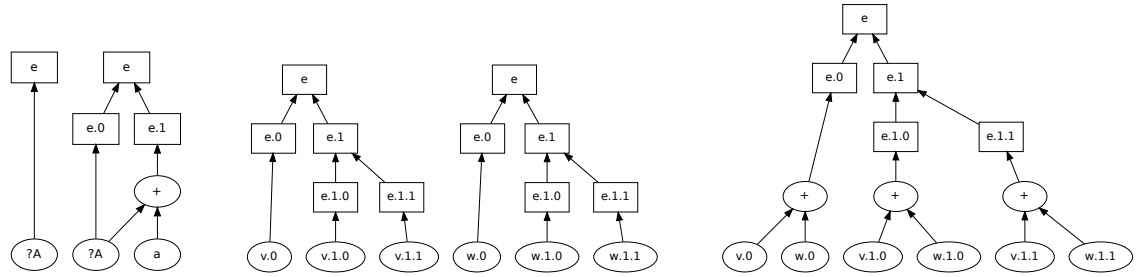
Als letzter Schritt soll der erzeugte Graph noch in Linearform gebracht werden, also in eine Abfolge von einzelnen Anweisungen. Der Ansatz hier ist, Zwischenergebnisse in temporäre Variablen zu speichern, wobei jede Variable genau einmal geschrieben wird (SSA-Form). Hierzu wird für jedes Blatt des Baums ein rekursiver Algorithmus aufgerufen, der einen Knoten linearisiert:

Wurde ein Knoten bereits lineariert, wird dieser kein zweites mal verarbeitet. Ansonsten werden zunächst all die Knoten rekursiv verarbeitet, die zum aktuellen Knoten hinführen. Danach wird geprüft, ob eine temporäre Variable benötigt wird oder ob das Ergebnis des Knoten Teil der Ausgabe des Homomophismus ist. Zuletzt wird die dem Knoten entsprechende Anweisung, bestehend aus Operation, den Eingangsvariablen und der Ausgangsvariablen, in einer Liste gespeichert.

Seien A, B, C atomare Gruppen und $D = A \times B,\ G = C \times D$. Ferner seien Variablen $a \in A,\ v, w \in G$ und $d \in D$ gegeben.



Variable `v` | Zufallswert `?G` | Ausdrücke `<C` und `d` | Tupel `[<C,d]` | Komponente `v.1`



`?A` und `?A:[#,#+a]` | Variablen `v` und `w` | Operation `v + w`

Abbildung 4.3.: Beispiele für Ausdrucks-Graphen

Schnittstellen

Die Ausgabe des Parsers ist ein Objekt der Klasse `statementlist` (siehe Abbildung 4.2). Diese Datenstruktur kann nun weiterverwendet werden, beispielsweise in unserem Fall durch den Interpreter, oder durch Codegeneratoren und zur Ausgabe von Dokumentation.

Die Klasse `statementlist` stellt neben dem Konstruktor, der mit dem Parser aus der Eingabedatei die Datenstruktur erzeugt, vier weitere Methoden zum komfortablen Zugriff auf die einzelnen Anweisungen der Eingabe bereit:

- `zkparser_get_group`
- `zkparser_get_var`
- `zkparser_get_hom`
- `zkparser_get_sigma`

Diese Methoden erwarten als Parameter den Namen einer Gruppe, einer Variablen, eines Homomorphismus oder einer Sigma-Definition. Zurückgegeben wird ein Zeiger auf das angefragte Objekt.

4.2.4. Interpreter

Der zweite Teil der Referenzimplementierung ist der Interpreter. Dieser erzeugt keinen eigenständigen Code, sondern kann Zero-Knowledge-Beweise direkt ausführen.

Der Interpreter besteht aus einer Reihe von Klassen:

- **Parser:** Die Parser-Klasse ist der zentrale Punkt des Interpreters. Sie bildet die Schnittstelle zum Parser[3] und erstellt alle nötigen Objekte. Außerdem enthält sie Schnittstellen, um auf die Elemente der Datenstruktur zuzugreifen.

  Wird ein neues Parser-Objekt erstellt, wird anhand der Datenstruktur für jede definierte Gruppe (auch anonyme) ein Group-Objekt erzeugt, ebenso wird für jede definierte Variable ein Element-Objekt erzeugt.

  Homomorphismen benötigen mehr Aufwand. Diese werden so konvertiert, dass die Hom-Klasse sie verarbeiten kann: Es müssen Referenzen zu Variablen und Gruppen aufgelöst werden. Danach wird auch für jeden definierten Homomorphismus ein Hom-Objekt erzeugt.

  Für Sigma-Protokolle werden noch keine Objekte erzeugt, da von demselben Protokoll mehrere Objekte zur selben Zeit benötigt werden können. Diese werden dann auf Anfrage erzeugt.

- **Group:** Dies ist eine abstrakte Klasse, die nur die Schnittstellen zu den Gruppen definiert. Die Schnittstellen müssen dann von den echten Gruppen implementiert werden.

  Gruppen bündeln alle Gruppenoperationen, die in Kapitel 4.2.2 definiert wurden. Zusätzlich werden die Operationen `identity`, `min`, `max`, `cmp`, `inGroup` bereitgestellt, die eine Variable auf das neutrale Element, das Minimum oder Maximum setzen beziehungsweise zwei Elemente vergleichen oder prüfen, ob ein Element überhaupt in der Gruppe enthalten ist.

- **Element:** Die Element-Klasse repräsentiert ein Element einer beliebigen Gruppe. Hierzu hält jedes Element eine Referenz auf die Gruppe und speichert den Wert des Elements als Array von ganzen Zahlen (`mpz_t`). Zudem hat jedes Element die Information gespeichert, ob ihm bereits ein Wert zugewiesen wurde. Wird versucht, mit einem Element zu rechnen, das noch keinen Wert hat, führt dies zu einem Laufzeitfehler.

  Um Gruppen-Operationen auszuführen muss die entsprechende Operation an dem Element aufgerufen werden, dem das Ergebnis zugewiesen werden soll. Hierbei wird geprüft, ob die Parameter aus den richtigen Gruppen sind und ob sie initialisiert sind.

- **Hom:** Jeder Homomorphismus wird als ein Objekt der Klasse Hom dargestellt. Dieses hat nur eine relevante Operation: map(). Diese bildet ein Element der Urbildgruppe auf ein Element der Bildgruppe ab. Hierzu wird die Liste der Anweisungen, wie sie der Parser berechnet hat, seriell ausgeführt.

[3] Mit Parser ist der Parser aus Kapitel 4.2.3 gemeint. Wird die Klasse des Interpreters oder ein Objekt dieser Klasse gemeint, steht im Text Parser-Klasse oder Parser-Objekt.

- **Sigma:** Die Sigma-Klasse ist eine abstrakte Klasse, die das Sigma-Protokoll implementiert. Definiert sind unter anderem die Funktionen P_1, P_2, V, S, R, C, wie sie in Kapitel 3.5 beschrieben wurden. Enthalten sind auch Funktionen, um die Kommunikation zwischen $\mathcal{P}$ und $\mathcal{V}$ zu ermöglichen. Mit Hilfe der Convert-Klasse können die Nachrichten r, c, s in Form von Byte-Arrays gelesen und gesetzt werden.

  Für die Unterstützung von Nichtinteraktiven Beweisen sind die Funktionen *sign* und *verify* enthalten.

  Die Sigma-Klasse muss von konkreten Sigma-Klassen implementiert werden. Dies sind zum jetzigen Zeitpunkt *Phi*, *Gsp*, *And* und *Or*.

- **Convert:** Mit der Convert-Klasse können beliebige Elemente, Zahlen und Binärblöcke zu Nachrichten serialisiert und deserialisert werden, um sie beispielsweise über ein Netzwerk übertragen zu können.

- **Hash:** Die Klasse Hash stellt eine Abstraktion von kryptographischen Hash-Funktionen dar. Konkrete Hash-Funktionen wie SHA-1 implementieren diese Klasse, um über eine gemeinsame Schnittstelle benutzbar zu sein. Dies gewährleistet eine unkomplizierte Austauschbarkeit, wenn neue Hash-Funktionen benötigt werden.

- **Random:** Um Zufallszahlen zu erzeugen, kann die Random-Klasse verwendet werden. Hier finden sich Funktionen, um zufällige Zahlen aus einem bestimmten Intervall oder mit bestimmter Bitlänge zu wählen. Es können auch Primzahlen und sichere Primzahlen mit gewünschter Größe erzeugt werden.

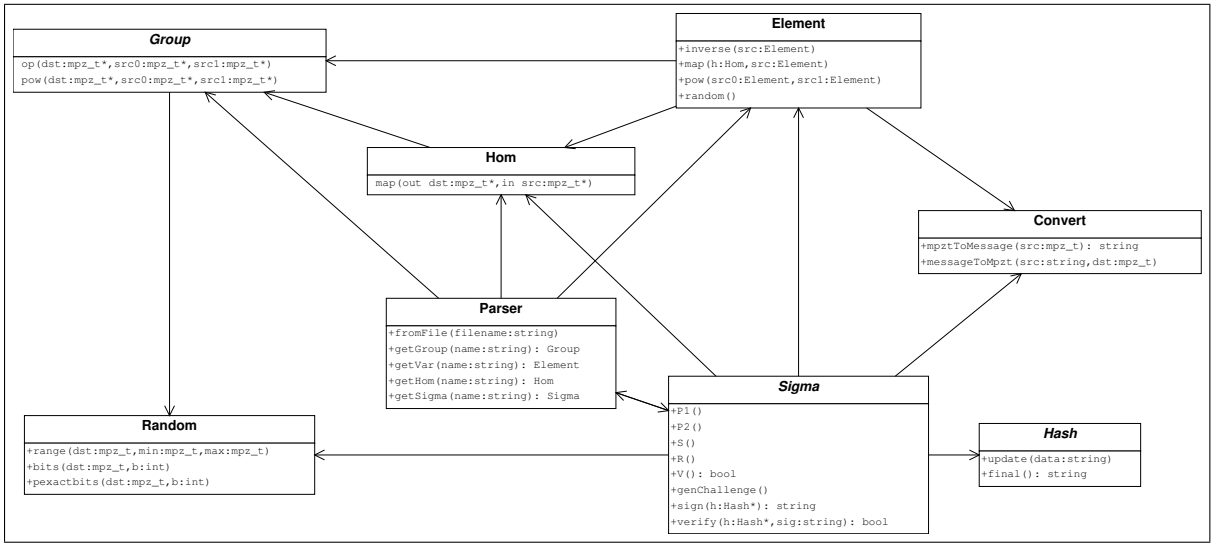


Abbildung 4.4.: Klassendiagramm, Auswahl der verfügbaren Methoden

4.3. Benutzung

Um einen Beweis, der in Camenisch-Stadler-Notation vorliegt, zu implementieren, müssen erst einmal einige Schritte zur Vorbereitung unternommen werden. Anschließend muss eine Eingabe für den Parser geschrieben werden, als letztes ein Programm, welches den Parser benutzt.

4.3.1. Vorbereitung

Um den Beweis durchzuführen, müssen zunächst einige Vorbereitungen getroffen werden:

1. Der Beweis muss so umgestaltet werden, dass Intervallprüfungen, polynomielle, multiplikative und lineare Abhängigkeiten nicht mehr vorkommen. Wie dies im Einzelnen möglich ist, ist in Kapitel 3 beschrieben.

2. Die logischen Abhängigkeiten (Und, Oder) müssen in eine disjunktive Normalform gebracht werden. In den Und-Termen dürfen keine abhängigen Variablen vorkommen (siehe Seite 28); diese müssen entsprechend zu Homomorphismen umgewandelt werden. Die Spezifikation des Beweises sollte jetzt nur noch mit Und und Oder verknüpfte „Urbild“-Beweise enthalten.

3. Es muss definiert werden, in welchen Gruppen die Berechnungen stattfinden; dies wird in der Regel in der Literatur definiert, die auch den Beweis enthält.

4. Die konkreten Systemparameter, Moduli, Generatoren, Sicherheitsparamter, etc. sind zu wählen.

4.3.2. Eingabe

Die Eingabe muss erstellt und als Text-Datei gespeichert werden (Datei-Endung per Konvention: `.zk`).

Der Inhalt der Datei ist wie folgt zusammengestellt:

1. Die Eingabe startet mit den Gruppendefinitionen.

2. Als zweites werden die Variablen und Generatoren angegeben. Die Generatoren können gleich initialisiert werden. Es ist zu beachten, dass jeder Σ^{ϕ}- und Σ^{GSP}-Beweis zwei Variablen benötigt: eine für das Geheimnis und eine für das Commitment.

3. Zu jedem „Urbild“-Beweis aus der Spezifikation muss ein Homomorphismus definiert werden, es sei denn, zwei Homomorphismen sind identisch.

4. Die Sigma-Protokolle für die „Urbild“-Beweise werden spezifiziert.

5. Es werden die Sigma-Protokolle für Σ^{AND}-Beweise und zuletzt die Σ^{OR}-Beweise angegeben.

4.3.3. Programm

Es gibt (in der Regel) zwei Programme: eines für den Prover und eines für den Verifier. Die ersten Schritte sind noch identisch:

1. Der Zufallszahlengenerator muss initialisiert werden: `Random::init();`
2. Es muss ein Parser-Objekt erstellt werden, der die Eingabe liest:
   `Parser *p = Parser::fromFile("beweis.zk");`
3. Öffentliche Variablen (Generatoren, Commitments (x)) werden gesetzt mit:
   `p->getVar("x")->set("123");`
4. $\mathcal{P}$ setzt genauso seine geheimen Variablen.
5. Ein Sigma-Objekt wird erstellt. Mit diesem wird das Sigma-Protokoll ausgeführt:
   `Sigma *s = p->getSigma("sigma");`

Die weiteren Schritte unterscheiden sich zwischen $\mathcal{P}$ und $\mathcal{V}$.

Der Prover sieht (vereinfacht) so aus:

1. `s->P1(); msg = s->getCommitment();`
2. Die Nachricht `msg` an den Verifier übertragen und auf die Challenge warten.
3. `s->setChallenge(msg); s->P2(); msg = s->getResponse();`
4. Danach muss die Response an den Verifier gesendet werden. $\mathcal{P}$ ist fertig.

Der Verifier könnte so aussehen:

1. Auf das Commitment vom Prover warten.
2. `s->setCommitment(msg); s->genChallenge(); msg = s->getChallenge();`
3. Die Challenge `msg` an den Prover schicken und auf die Response warten.
4. `s->setResponse(msg); if(s->V()) { /* success */ } else {/* fail */}`
5. An dem Ergebnis von `V()` sieht $\mathcal{V}$, ob der Beweis erfolgreich war. $\mathcal{V}$ ist damit auch fertig.

Das Protokoll kann nun wiederholt werden, um die Betrugswahrscheinlichkeit weiter abzusenken.

Am Ende müssen noch die Objekte gelöscht und der Zufallszahlengenerator beendet werden:
`delete s; delete p; Random::free();`

4.4. Beispiel

$\mathcal{P}$ zeigt, dass er ein Commitment x öffnen kann.

Eingabe für den Beweis:

```
G = Z_add_n (509);              // Gruppe G = Z_509
GxG = (G, G);                   // Gruppe GxG = G × G
H = Z_mul_n (1019, qr);         // Gruppe H = QR(Z*_1019)
GxG: w;                         // Variable w ∈ G × G
H: x, g = 452, h = 311;         // Variablen x,g=452,h=311 ∈ H
phi [ GxG -> H ] = (g ^ $.0,    // Homomorphismus
       h ^ $.1) : #.0 + #.1;    // φ((a_0,a_1)) = g^a_0 · h^a_1
sigma = SigmaPhi[phi, x, w, 20];
/* ZPK[(w_0,w_1) : x = g^w_0 · h^w_1] */
```

Programmcode, um den Beweis auszuführen:

```
Parser *p = Parser::fromFile("beweis.zk"); // Parser-Objekt für P erstellen
Parser *v = Parser::fromFile("beweis.zk"); // Auch ein Objekt für Verifier
p->getVar("w")->random();                  // Zufälliges w wählen
p->map("x","phi","w");                     // Commitment x berechnen
    /* x von P zu V übertragen */
p->P1();                                   // Prover benutzt P1
v->setCommitment(p->getCommitment());      // Nachricht r übertragen
v->genChallenge();                         // Verifier wählt Challenge
p->setChallenge(v->getChallenge());        // Nachricht c übertragen
p->P2();                                   // Prover benutzt P2
v->setResponse(p->getResponse());          // Nachricht s übertragen
if(v->V()) { /* erfolgreich */ }           // Verifier prüft Beweis
```

5. Fazit

5.1. Ergebnis der Arbeit

Das Ziel dieser Arbeit war es, ein Rahmenwerk zu entwickeln, um die Theorie der Zero-Knowledge Proofs of Knowledge in der Praxis einsetzen zu können.

Um dieses Ziel zu erreichen, habe ich die Theorie, aufbauend auf der Arbeit des CACE-Projekts [S+09], weiter formalisiert, wobei ich mich auf die grundlegenden Protokolle und Techniken beschränkt habe.

Um kommenden Prüflingen den Einstieg in das sehr komplexe Themengebiet zu erleichtern, habe ich zudem die mathematischen Grundlagen zusammengefasst, auf denen Zero-Knowledge-Beweise basieren. Dieser Teil ist notwendig, um die grundlegende Funktionsweise von Zero-Knowledge Proofs of Knowledge zu verstehen und bietet gleichzeitig die Möglichkeit, diesen Teil der Zahlentheorie in einer wissenschaftlichen Anwendung zu sehen.

Aufbauend auf der Formalisierung habe ich eine formale Sprache entwickelt, die es erlaubt, Zero-Knowledge Proofs of Knowledge mit allen Systemparametern vollständig zu spezifizieren. Dadurch hatte ich auch einen tiefen Einblick in die Funktionsweise und die logischen Hintergründe der Beweise.

Die Sprache erlaubt es, verwendete Gruppen genau anzugeben und falls nötig neue Familien von Gruppen zu implementieren. Eine weitere Stärke der Sprache ist die Möglichkeit, mit kartesischen Produkten in beliebiger Komplexität umzugehen. Selbstverständlich besteht die Möglichkeit, Variablen zu benutzen, wobei ein striktes Typsystem verwendet wird, um Fehlberechnungen auszuschließen. Mit Hilfe der Homomorphismen lassen sich alle Berechnungen spezifizieren, die für Zero-Knowledge-Beweise notwendig sind. Da hier keinerlei Rekursion oder Schleifen möglich sind, ist auf der einen Seite zwar die Mächtigkeit der Berechnungen eingeschränkt, auf der anderen Seite ermöglicht dies erst die unkomplizierte Übersetzung der Beweise in andere Programmiersprachen, Logikschaltkreise und Ähnliches. Zudem vereinfacht die Einschränkung die Analyse und Optimierungsmöglichkeiten der Programme sehr stark. Mit den Sigma-Protokollen lassen sich Beweise, die in Camenisch-Stadler-Notation vorliegen, auf einfache Weise formal spezifizieren. Außerdem können neu entwickelte Sigma-Protokolle auf einfachem Weg in die Sprache integriert werden.

Ein Großteil dieser Arbeit wurde auf die Erstellung eines Compilers für Zero-Knowledge Proofs of Knowledge verwendet. Mit diesem ist es möglich, die oben erwähnte Formalisierung der Zero-

Knowledge-Beweise, welche sich in Form der Eingabesprache zeigt, in eine Datenstruktur zu überführen und die Beweise mittels eines Interpreters auszuführen.

Der entwickelte Compiler besteht aus zwei maßgeblichen Komponenten: Einem Parser und einem Interpreter:

Der Parser ist dafür zuständig, die Eingabesprache in eine Datenstruktur umzusetzen. Er wurde mittels Bison entwickelt und ist aufgrund seiner Ausgabe modular zu nutzen. Desweiteren kann durch definierte Schnittstellen das Grundkonstrukt an Beweisen erweitert werden. Ein weiterer Vorteil dieses Compilers gegenüber bestehenden Lösungen ist die Verfügbarkeit des Quellcodes und die damit verbundende Vertrauenswürdigkeit, Erweiterbarkeit und Wartbarkeit.

Der Interpreter verwendet die durch den Parser erzeugte Datenstruktur und erlaubt die tatsächliche Ausführung der Beweise. Hierzu enthält der Interpreter Implementierungen der verwendeten Gruppen, Elemente und Homomorphismen, sowie Funktionen zur Erzeugung von Zufallszahlen und weiteren Hilfsfunktionen. Die Sigma-Protokolle, die in den Zero-Knowledge Proofs of Knowledge einen zentralen Baustein darstellen, wurden von Alexander Klein entwickelt, der im Rahmen seiner Diplomarbeit an dem Projekt beteiligt war.

Durch die Implementierung entstand auch ein tieferes Verständnis von der Anwendbarkeit und Nutzung von Zero-Knowledge-Beweisen in praktischen Anwendungen und den damit verbundenen Schwierigkeiten.

5.2. Ausblick

Die in diesem Projekt erarbeiteten Lösungen bieten eine breite Grundlage, um praktische Anwendungen umzusetzen. Je nach Spezialfall und Einsatzgebiet lässt sich das Projekt durch zusätzliche Module erweitern.

Da es auf dem Gebiet der Zero-Knowledge-Beweise und Kryptographie ständig Fortschritte gibt, kann es notwendig sein, den Compiler entsprechend zu erweitern. Es können dafür weitere Gruppen implementiert werden, zum Beispiel elliptische Kurven über einem endlichen Körper. Durch den modularen Aufbau des Compilers ist dies mit wenig Aufwand verbunden. Wenn veraltete Hash-Funktionen durch verbesserte Funktionen ersetzt werden, können auch diese auf einfachem Wege implementiert werden.

Eine Schwierigkeit bei der Verwendung der Eingabesprache besteht darin, dass Beweise, die in der Camenisch-Stadler-Notation vorliegen, zunächst manuell umgeformt werden müssen, so dass sie als Eingabe für den Parser verwendet werden können. Eventuell könnte diese Umformung durch ein Programm unterstützt werden.

Ein für praktische Anwendungen sehr wichtiger Aspekt ist die Kommunikation zwischen Prover und Verifier, wenn die beiden Parteien nicht auf demselben Rechner arbeiten. Die hierzu auszutauschenden

Nachrichten liegen bereits in serialisierter Form vor. Es müssen aber noch Kommunikationsmodule geschaffen werden, die diese Nachrichten beispielsweise über TCP/IP, Bluetooth und Near Field Communication transportieren.

Dadurch dass der Compiler modular aufgebaut ist, kann er um weitere Module zur Erzeugung von Dokumentation aus den Beweisen und zur Generierung von Programmcode in verschiedenen Sprachen erweitert werden. Diese können direkt auf die durch den Parser erzeugte Datenstruktur zugreifen oder auf dem Interpreter so aufbauen, dass dieser, anstatt die Beweise auszuführen, sie in anderer Form ausgibt. Zu diesem Thema hat bereits ein weiterführendes Projekt am Arbeitsbereich SVA der Technischen Universität Hamburg-Harburg begonnen.

Der vorliegende Compiler wurde unter GNU/Linux und Windows entwickelt und auf diesen Plattformen ausgiebig getestet. Die Portierung auf weitere Plattformen, zum Beispiel Android-Mobiltelefone, wäre ein weiteres interessantes Projekt.

Zu guter Letzt gibt es immer die Möglichkeit, mit Hilfe des Compilers praktische Anwendungen zu verwirklichen, die Zero-Knowledge-Beweise benötigen. Als Beispiele sind hier Gruppensignaturen, anonyme Zahlungsverfahren und sichere Zutrittskontrollsysteme zu nennen.

A. Abkürzungen, Notation und Konventionen

In der vorstehenden Arbeit wurden die folgenden Abkürzungen und Notationen verwendet:

$\circ, \diamond, \star$	Verschiedene binäre Gruppenoperationen
$\stackrel{?}{=}$	Prüfung auf Gleichheit
$\stackrel{?}{\in}$	Prüfung auf Mitgliedschaft
$\mathbb{N}$	Menge der natürlichen Zahlen $\{0,1,2,3,\ldots\}$
$QR(G)$	Untergruppe von G, enthält alle quadratischen Reste
$\mathbb{Z}$	Menge der ganzen Zahlen $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
$\mathbb{Z}_n$	Additive Restklassengruppe modulo n
$\mathbb{Z}_n^*$	Multiplikative Restklassengruppe modulo n
$[3]$	Element in einer Restklassengruppe
$a \in_R A$	a ist Zufallselement aus der Gruppe A
a^{-1}	Inverses von a
$[a,b]$	Intervall der ganzen Zahlen von a bis b, $\{a,a+1,a+2,\ldots,b-2,b-1,b\}$
$[a,b)$	Intervall der ganzen Zahlen von a bis $b-1$, $\{a,a+1,a+2,\ldots,b-2,b-1\}$
$\lfloor a \rfloor$	Größte ganze Zahl $\leq a$ (Gaußklammer)
$\lceil a \rceil$	Kleinste ganze Zahl $\geq a$
$\langle a \rangle$	Durch a erzeugte Gruppe
c^+	Obere Grenze von Challenges; $c \in_R [0,c^+)$
e	Neutrales Element einer Gruppe
ggT	Größter gemeinsamer Teiler
$\mathcal{P}$	Prover (Beweiser)
`#t`	true
$\mathcal{V}$	Verifier (Prüfer)

In dieser Arbeit wurden intuitive Begriffe wie „sehr schwer“, „praktisch nicht berechenbar“, „einfach“, „vernachlässigbar“, „effizient“ und so weiter verwendet. Aus Gründen der Lesbarkeit wurde an denselben Stellen keine formale Definition dieser Begriffe gegeben.

Eine Funktion $\varepsilon(l)$ ist vernachlässigbar in Bezug auf l, wenn für jedes Polynom p und für fast alle l gilt: $\varepsilon(l) \leq \frac{1}{p(l)}$ ([DN08])

Ein Problem ist einfach zu berechnen oder effizient lösbar, wenn es einen probabilistischen Algorithmus gibt, der das Problem mit nicht vernachlässigbarer Wahrscheinlichkeit löst mit einer Ausführungsdauer, die schlimmstenfalls polynomiell von der Länge der Eingabe abhängt. Bei üblichen Eingabelängen und auf üblichen Rechnern liegt die Berechnungsdauer weit unter einer Sekunde.

Sehr schwer oder praktisch nicht berechenbar ist ein Problem dann, wenn die Laufzeit von jedem probabilistischen Algorithmus, der dieses Problem mit nicht vernachlässigbarer Wahrscheinlichkeit löst, mindestens exponentiell von der Länge der Eingabe abhängt. Bei Eingaben hinreichender Länge und auf sehr schnellen Rechnern würden die Berechnungen viele tausend Jahre dauern.

Danksagung

Ich danke dem Institut für Sicherheit in verteilten Anwendungen an der Technischen Universität Hamburg-Harburg und insbesondere Alexander Klein und meinem Betreuer Tobias Jeske für die Bereitstellung der für die Arbeit notwendigen Infrastruktur, das spannende Thema und die gute Zusammenarbeit in den vergangenen Monaten.

Desweiteren danke ich meiner Mutter, Barbara, Christian und Mare für die moralische Unterstützung und die Gespräche über meine Diplomarbeit in den letzten Wochen.

Literaturverzeichnis

[ACJT00] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *CRYPTO '00: Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, pages 255–270, London, UK, 2000. Springer-Verlag.

[Big85] Norman L. Biggs. *Discrete Mathematics*. Oxford Science Publications, 1985.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 209–218, New York, NY, USA, 1998. ACM.

[CGH07] Sébastian Canard, Aline Gouget, and Emeline Hufschmitt. Handy compact e-cash system. In *Proceedings of the 2nd Conference on Security in Network Architectures and Information Systems - SAR-SSI*, 2007.

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *In EUROCRYPT, volume 3494 of LNCS*, pages 302–321. Springer-Verlag, 2005.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 410–424, London, UK, 1997. Springer-Verlag.

[DF02] Ivan Damgård and Eiichiro Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *ASIACRYPT '02: Proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security*, pages 125–142, London, UK, 2002. Springer-Verlag.

[DN08] Ivan Damgård and Jesper Buus Nielsen. Commitment schemes and zero-knowledge protocols (2008), 2008.

[ISO96] ISO. ISO/IEC 14977:1996(E), Extended Backus-Naur Form. Technical report, International Organization for Standardization, 1996.

[Lip03] Helger Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. In *Advances on Cryptology — ASIACRYPT 2003*, pages 398–415. Springer-Verlag, 2003.

[MOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 129–140, London, UK, 1992. Springer-Verlag.

[Pro] CACE Project. `http://www.cace-project.eu/`.

[S+09] Nigel P. Smart et al. Final report on unified theoretical framework of efficient zero-knowledge proofs of knowledge. Technical report, Computer Aided Cryptography Engineering, 2009.

[Wel09] Welt. Kreditkarten-datenklau trifft alle deutschen banken. `http://www.welt.de/finanzen/article5251635.html`, November 2009.

[Wik10] Wikipedia. Vier-quadrate-satz — wikipedia, die freie enzyklopädie. `http://de.wikipedia.org/w/index.php?title=Vier-Quadrate-Satz&oldid=77258224`, 2010. [Online; Stand 31. Juli 2010].

Erklärung

Ich, Jörn Heissler, versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Ich bin mit einer Einstellung in den Bestand der Bibliotheken der Universität Hamburg und der Technischen Universität Hamburg-Harburg einverstanden.

Hamburg, den 31. Juli 2010

Jörn Heissler